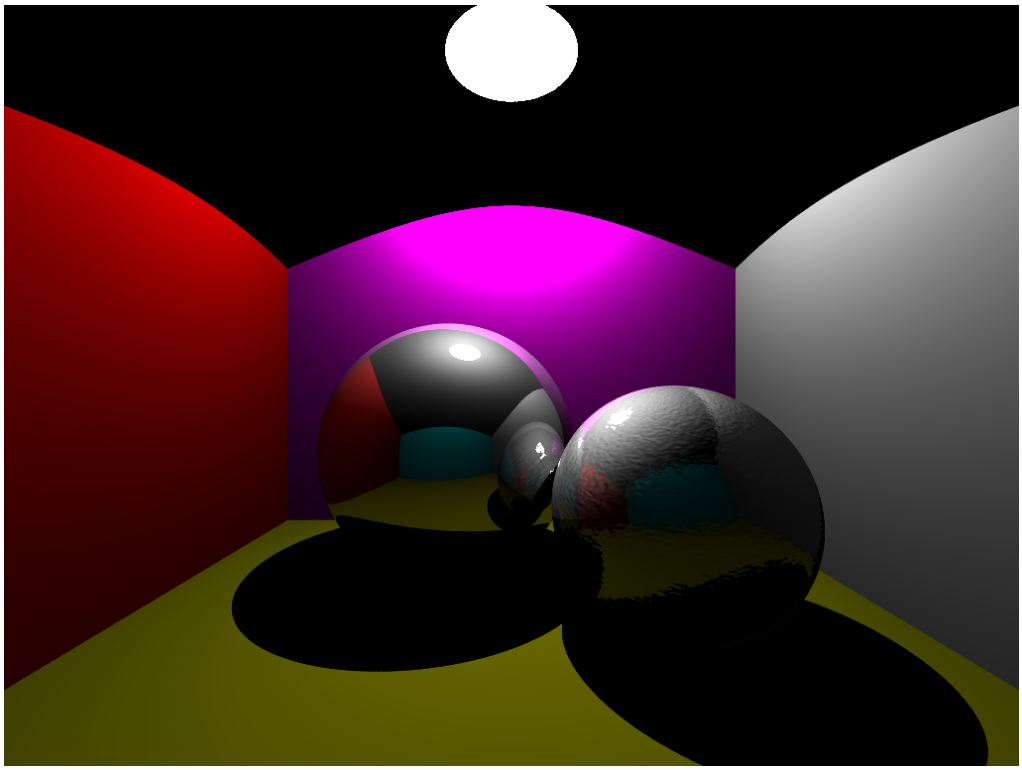
Raytracing

doktor@dyregod.dk — **Ulf Holm Nielsen**
tnjr@ruc.dk — **Thomas Riisbjerg**
mads@danquah.dk — **Mads Danquah**
hartlev@ruc.dk — **Morten Hartlev Poulsen**

Vejleder:

keld@ruc.dk — **Keld Helsgaun**

30. maj 2002



Roskilde Universitetscenter
Naturvidenskabelig Basisuddannelse Hus 14.2
4. semester

Abstract

Projektets formål er at udvikle en avanceret raytracer, der er i stand til at generere billeder med *Monte Carlo* og *Photon Mapping* metoder. Yderligere skal en læser være i stand til at skrive sin egen raytracer efter at have læst rapporten.

Selv om formidlingen af rapporten ikke blev som ventet, er det projektgruppens mening at den præsenterede teori stadig med fordel kan benyttes. Selve raytraceren opnår næsten alle dens mål, bortset fra evnen til at håndtere farvede overflader kombineret med photon mapping, samt muligheden for at følge diffuse stråler i visse tilfælde. Der præsenteres forslag til hvordan den manglende funktionalitet implementeres.

Nøgleord: Raytracing, Photonmapping, Java, Datalogi, strålesporing

The aim of this paper is to develop an advanced raytracer capable of rendering images with the *Monte Carlo* and the *Photon Mapping* methods. Further more, a reader should be able to write his own raytracer upon reading the paper.

While the educational value of the paper did not turn out as expected, it is our belief that the ideas presented in the paper are of some value to the reader. Also, the actual raytracer almost achieves all of its goals, only lacking the ability to handle colored surfaces in conjunction with photon mapping as well as being able to follow diffusely reflected rays under certain circumstances. Suggestions for implementing the missing features are presented.

The raytracer was implemented in Java, the included CD-ROM contains the full source-code as well as compiled binaries.

Keywords: Raytracing, Photon mapping, Java, Computer Science

Indhold

Figurer

Kode-lister

Forord

Denne rapport er udarbejdet som led i 4. semester projekt på den Naturvidenskabelige Basisuddannelse ved Roskilde Universitetscenter. Rapporten er udarbejdet af Ulf Holm Nielsen, Thomas Riisbjerg, Mads Danquah og Morten Poulsen.

Forsidebilledet er en såkaldt Cornell boks, der benyttes til at teste global belysnings algoritmer. Billedet er lavet med raytraceren udviklet i dette projekt.

Der er i projektet lagt vægt på at gøre teorien let tilgængelig, samt anskueliggøre for læseren hvordan den konstruerede raytracer er opbygget. Ideen er at læseren selv skal kunne lave en raytracer efter at have læst rapporten. Dette har dog gjort at vægten ligger mere på teori- og dokumentationsafsnittene end på afprøvning.

Raytracer-koden er dokumenteret på engelsk i kommentarer, men på dansk i rapporten. Det sker ud fra en forestilling om at ikke kun danskere vil interesserede i at læse koden.

Projektet har været lærerigt for alle involverede, da det har været større og mere udfordrende end nogen turde håbe på.

Vi vil her gerne takke Linda Lund Andersen og Kirsten Madsen for korrekturlæsning, foruden hvilken denne rapport nok ikke ville kunne læses som en helhed.

Rapporten er sat i LaTeX, alle figurer er udarbejdet af projektgruppen.

Rapporten og raytracer koden er også tilgængelig i elektronisk form på den medfølgende CD samt på følgende adresse:

http://www.dyregod.dk/raytracer/

Læsevejledning

Målgruppe

Denne rapport er henvendt til folk med interesse for raytracing, og evt folk med interesse i at implementere en sådan.

Der forudsættes matematikkundskaber svarende til A niveau fra gymnasiet, og et kendskab til linær algebra vil være en klar fordel. Der forudsættes datalogiske kundskaber svarende til indledende kurser i programmering på universitetsniveau.

Afsnit

Rapporten er delt op i følgende dele:

Introduktion En overordnet introduktion til hvad raytracing. Introduktionen sluttes af med en kort foranalyse der giver en overordnet kravspecifikation til raytraceren.

Teori Gennemgang af teori der tages i brug under implementeringen af raytraceren. Teorien er delt op i 3 dele

- **Grundlæggende raytracing** Gennemgang af den grundlæggende raytracer teori.
- **Global belysning** Gennemgang af teori til løsning af problemer med indirekte belysning
- **Udvidelser til den grundlæggende raytracer** Teori til elementer den grundlæggende raytracer er udvidet med.

Dokumentation Valg der er gjort i forbindelse med implementeringen af raytraceren raytraceren dokumenteres. En endelig kravspecifikation introduceres, og implementeringen af samt brugen af denne dokumenteres.

Resultater Indledes med et afprøvningsafsnit. Efterfølgende diskuteres implementering og kvalitet af raytraceren. Afsnittet afsluttes med en konklusion.

Med rapporten følger en CD indeholdende forskelligt relevant materiale, samt en færdig version af raytraceren.

Afsnit 1

Indledning

Der er i dag et stigende behov for computergenereret grafik.

Underholdningsbranchen bruger teknikken i film og computerspil. Reklamemageren er storforbruger af copmutergenereret grafik. Arkitekter i stigende grad raytracing til at visualisere hvordan byggerier vil tage sig ud med forskellige former for lys.

Der findes en række muligheder for generering af billeder. To ofte benyttede teknikker er *spanbaseret optegning* og raytracing.

I spanbaseret optegning, tager man udgangspunkt i de objekter, som billedet indeholder og optegner hvert objekt for sig, et af gangen. I raytracing tager man derimod udgangspunkt i lyset og følger det rundt i den opstillede verden, og ud fra denne information optegnes et billede.

Da raytracing bygger på en forsimplet model for lysets udbredelse , giver denne mulighed for de mest realistiske billeder. Spanbaseret optegning bygger på en mindre realistisk model, men er derimod meget hurtigere, da udregningerne bag ikke er så komplekse.

Når man arbejder med computergrafik ønsker man som oftest et så realistisk billede som muligt.I nogle situationer har man brug for at få optegnet billeder så hurtigt, at øjet dem som flydende bevægelser, når de fremvises. Dette er f.eks. et krav i de fleste computerspil, hvorfor de ofte benyttes span-baseret optegning. Hvis man derimod har mulighed for at generere billederne på forhånd, kan man med fordel benytte sig af den mere realistiske raytracing.

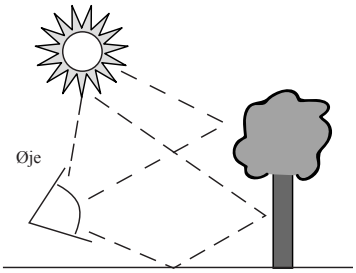
Det interessante ved raytracing er, at man med en perfekt raytracer har mulighed for at få skabt et (set med vores øjne) perfekt billede. Raytracing er af samme grund en grunddisciplin inden for grafisk programmering.

Denne rapport behandler raytracing og de forskellige problemer der opstår i forbindelse med raytracing. Desuden implementeres en raytracer på baggrund af det behandlede stof.

Afsnit 2

Introduktion til Raytracing

Når man betragter den virkelige verden, rammer lyset fra solen forskellige genstande, der reflekterer lyset. Noget af dette lys vil ramme vores øje og danner et billede af verden på vores nethinde , se figur 2.1.



Figur 2.1: Den virkelige verden

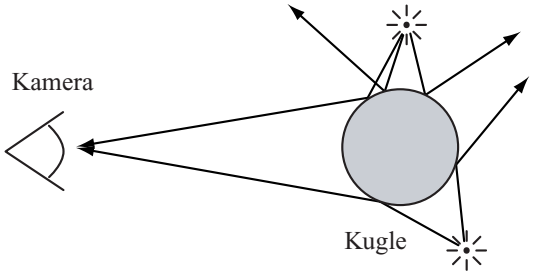
Raytracing er en form for lyssimulering, som tager udgangspunkt i en opstillet verden, kaldet en scene, der indeholder en række figurer, lys og et kamera. Kameraet er vores øje hvormed vi ser ind i den opstillede verdenen.

I raytracing vælges at betragte lysets udbredelse som *partikelbaner*, dvs. opstille dem som linier, frem for bølger.

Ønsker man at simulere hvordan lys udsendes og reflekteres i den virkelige verden, kræver det at man simulerer nærmest uendelig mange stråler.

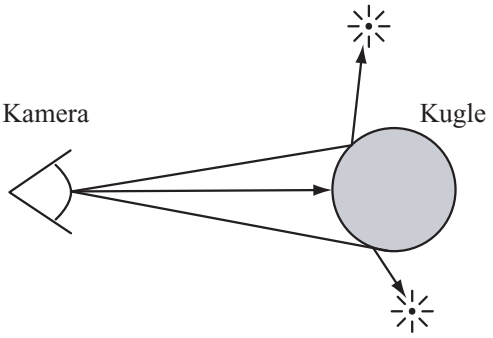
Det skyldes, at det kun er en brøkdel af strålerne fra lyskilderne der, evt via diverse refleksioner, rent faktisk rammer kameraet. Desuden kan man forestille sig at der er områder i scenen som kameraet slet ikke kan se. Man risikerer derfor at følge en masse stråler som slet ikke har betydning for billedet, se figur 2.2.

I stedet for at tage udgangspunkt i lyset og følge strålerne derfra kan man tage



Figur 2.2: Forlæns raytracing vist i 2-D

udgangspunkt i kameraet og følge strålerne baglæns (figur 2.3). På denne måde undgår man at følge stråler, der ikke bidrager til billedet. Men desværre undgår man også de stråler, der *indirekte* bidrager til billedet.



Figur 2.3: Baglæns raytracing vist i 2-D

Følger man strålerne fra lyskilden, kaldes det *forlæns raytracing*, og følger man strålerne fra kameraet, kaldes det *baglæns raytracing*. Baglæns raytracing er hurtigere og simplere end forlæns raytracing, men har det problem, at man ser bort fra en masse stråler. Ved baglæns raytracing ses der kun på direkte lys. Dvs. lyset, der falder på en figur vil kun komme fra lyskilder der lyser direkte på denne, og ikke lys der f.eks. bliver reflekteret fra andre figurer. Det reflekterede lys kaldes *indirekte belysning*.

Indirekte belysning kan simuleres med en række algoritmer, der under et kaldes *global belysning*. De søger alle at simulere alle lysets interaktioner med overflader i scenen. De er oftest en kombination af forlæns og baglæns raytracing, da der er fordele forbundet med begge metoder. Disse komplementerer hinanden godt i mange sammenhænge.

Afsnit 3

Foranalyse

Målet er at implementere en raytracer i Java, der, givet en scene som input, kan generere et billede af denne. Raytraceren skal tage højde for indirekte belysning samt reflektion og transmission.

Raytraceren skal kunne tegne figurerne kugle, plan og cylinder. Desuden skal det være nemt at udvide raytraceren med nye figurer. Endvidere skal det være muligt at kombinere figurer, eksempelvis trække et plan fra en kugle, således at man får en halvkugle.

Raytraceren skal kunne håndtere simple overfladetyper samt teksturering og bump mapping.

Raytraceren skal i alle henseender være nem at udvide med yderligere funktionaliteter, hvilket også er til gavn i udviklingsarbejdet.

Der stilles ikke krav om, at raytraceren skal producere fysisk korrekte billeder.

Del I

Teori

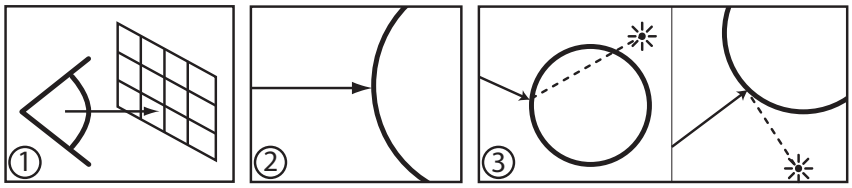
Afsnit 4

Raytracing

De følgende afsnit omhandler alle opbygningen af en baglæns raytracer. Først gennemgås de overordnede dele, hvorefter de enkelte dele vil blive præsenteret mere detaljeret.

4.1 En simpel raytracer

I en scene bestående af et kamera, en kugle og en lyskilde kan en helt simpel raytracer konstrueres ved at betragte kameraet som et plan hvor igennem verden ses. Man kan forestille sig kameraet som en ramme, man kan kigge ud igennem. Det ønskede resultat er et billede på en computer. Billedets pixels lægges ud over planet som et net. Hver maske i nettet svarer derfor til et pixel i det endelige billede.



Figur 4.1: De grundlæggende trin: 1: Udsendelse af stråle, 2: Undersøgelse af skæring, 3: Bestemmelse af belysning v.ha. shadow rays

Derefter "kastes" stråler fra et punkt bag kameraet igennem hvert pixel i planet, se den stiplede linie i figur 4.1, billede 1. Det undersøges, om disse stråler rammer kuglen, billede 2. Hvis strålen rammer kuglen, undersøges om det er muligt at "se" lyskilden fra skæringspunktet. Altså om lyskilden direkte oplyser punktet. I praksis skydes en stråle afsted mod lyskilden, hvorefter det undersøges, om der ligger objekter i vejen. En sådan stråle kaldes en *shadow ray*, se billede 3.

Hvis lyskilden er synlig, undersøges, det hvilken hvad farve kuglen har, ellers sættes farven til sort. Det pixel, som strålen gik igennem på vej ud, bliver tildelt kuglens farve. Hvis strålen slet ikke rammer nogle figurer, tildeles pixelen en baggrundfarve.

Den simple raytracer kan inddeles i følgende dele (jvf. figur 4.1)

1. Udsendelse af stråler
2. Undersøgelse af skæring
3. Bestemelse om lys er synligt
4. Beregning af lys der rammer skæringen

Den simple raytracer kan implementeres i pseudo kode som i listing 5.2.

Listing 4.1: Pseudo kode til simpel raytracer

```
optegn billede
    for hver pixel i billede
        udsend stråle fra kamera gennem pixel
        pixel farve = cast( stråle )

cast( stråle )
    find nærmeste skæring med figur
    hvis lyskilde synlig fra skæring
        farve = figur farve
    returner farve
```

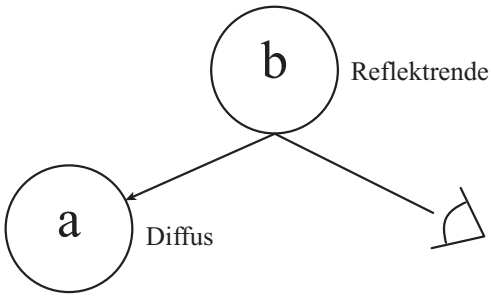
Dette vil resultere i et billede med omridset af en kugle. Det vil ikke være muligt at se detaljer på figurens overflade, ej heller vil der være en fornemmelse af perspektiv i billedet. Denne teknik til opbygning af billede kaldes for raycasting og blev introduceret i 1968 af Appel [FvDFH92].

For at få en fornemmelse af rumlighed, er det derfor nødvendigt at afbillede mere end blot et omrids. Det er nødvendig at kunne se lyset falde på figuren og fordele sig afhængigt af dens overflade. Der tages endnu ikke højde for lys, der reflekteres i flere objekter, før det rammer kameraet, eller for figurer der er gennemsigtige og derfor transmitterer lys.

Det er muligt at simulere lysets indvirken på overflader i en lysmodel. En sådan vil blive præsenteret senere, indtil videre antages det blot at virke uden nærmere forklaring. Se evt. afsnit 5 om belysning, side 26.

Reflektion og transmission kan løses ved at følge stråler videre fra første skæringspunkt. Hvis figuren reflekterer eller transmitterer lys udregnes efter første skæringspunkt en reflektions eller transmissions stråles retning og den nye stråle

følges videre i scenen. For hver stråle der følges videre, adderes farven på den figur, som den nye stråle rammer til den foregående skæring. Derved vil farven fra figurer, der spejles i en figur adderes til denne figurs farve. Dette er illustreret i figur 4.2 hvor en stråle via en reflekterende overflade sendes videre mod en ikke reflekterende.



Figur 4.2: Rekursiv raytracing

Det er nærliggende at implementere en sådan metode rekursivt da der for hvert skæringspunkt med en reflekterende eller transmitterende overflade skal udsendes en stråle hvor præcis de samme beregninger skal udføres for et nyt område i scenen. En rekursiv algoritme kan ses i pseudo kode i listing 4.2.

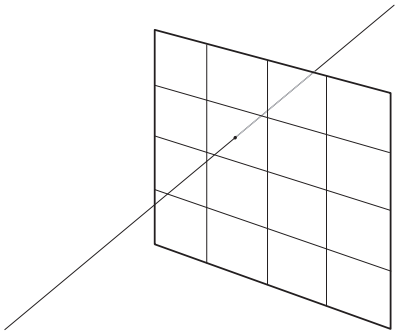
Listing 4.2: Pseudo kode til rekursiv raytracer

```
optegn billede
    for hver pixel i billede
        udsend stråle fra kamera gennem pixel
        pixel farve = trace( stråle )

trace( stråle )
    find nærmeste skæring med figur
    farve = shade( skæring )
    returner farve

shade( skæring )
    beregn reflektionsstråle
    beregn transmissionstråle

    for alle lys i scene
        hvis lys synligt fra skæring
            farve = beregn lys
    farve += trace( reflektionsstråle )
    farve += trace( transmissionstråle )
    returner farve
```

4.2 Komponenter

4.2.1 Opbygning af billedet

Som nævnt i indledningen opbygges outputbilledet ved for hver pixel at følge en stråle fra kameraet ind i scenen og undersøge om den skærer med nogle figurer, og i så fald hvilken farve figuren har i det nærmeste skæringspunkt.



Figur 4.3: Strålerne forlader kameraet og rammer figurer i scenen.

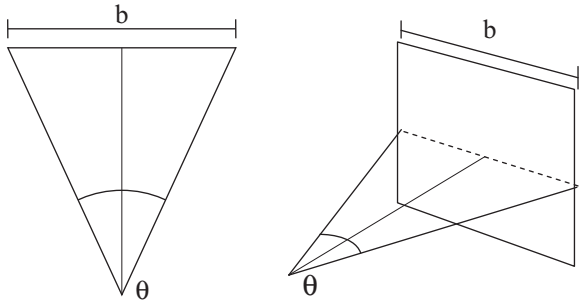
4.2.2 Kamera

Kameraet bruges til at placere og orientere de stråler som sendes ind i billedet.

Fra et abstrakt synspunkt består et kamera i en raytracer af et areal hvorpå scenen projiceres. Et forholdsvis simpelt tilfælde, som vil blive gennemgået her, er et kamera der består af et plan og et udgangspunkt for strålerne, som er placeret én enhed fra planet. Planet kaldes også et *billedplan*. Størrelsen af billedplanet afhænger af forholdet mellem højden og bredden på outputbilledet såvel som bredden på kameraets synsvinkel. Det er vigtigt at bemærke, at selve opløsningen på outputbilledet ikke berører størrelsen af billedplanet, kun tætheden hvormed strålerne sendes ud igennem billedplanet.

Fra scenen har vi synsvinkelen θ og fra brugeren bredden og højden på outputbilledet. Bredden b på billedplanet findes ud fra den simple trigonometriske betragtning, at billedplanet og udgangspunktet danner en ligesidet trekant.

$$b = 2 \cdot \tan(\frac{\theta}{2})$$



Figur 4.4: Forholdet mellem synsvinkel og billedplanets bredde

Eftersom forholdet mellem outputbilledets bredde og højde skal være ens med billedplanets, findes højden af billedplanet simpelthen ved $h = bredde/højde * b$

Når billedplanets størrelse er kendt, er det muligt at bestemme, hvor på billedplanet hver stråle skal gå igennem, når antallet kendes. Antallet af stråler, der sendes igennem billedplanet bestemmes, af opløsningen på outputbilledet. Billedplanet deles ind i et gitter, med et tern(gitterpixel) hver pixel i outputbilledet. Som udgangspunkt udsendes der en stråle for hver *gitterpixel*, netop for at undersøge hvilken farve den tilsvarende pixel i outputbilledet skal have (se figur 4.2.1).

Listing 4.3: Pseudokode til opbygning af billede

```
optegn billede
    for hver pixel i outputbilledet
        find stråle for (x,y) i billedet
        farve = trace( stråle )
```

Der kan konstrueres andre typer kameraer end det ovenstående. F.eks. kan der et parallelt kamera, hvor alle udsendte stråler ligger parallelt eller et sfærisk kamera hvor stråler udsendes fra et centrum i en kugle frem for igennem et billedplan.

4.2.3 Stråler

Stråler er repræsenteret som *parametriske* linier. De har et udgangspunkt $\vec{w}$ og en *retningsvektor* $\vec{v}$. Parameteren t repræsenterer altså afstanden fra udgangspunktet.

$\vec{x} = \vec{w} + \vec{v}t$

Stråler bruges til at "føle"hvad der ligger i scenen. I tilfældet med ovenstående kamera sættes strålens udgangspunkt $\vec{w}$ til kameraets udgangspunkt, og retningsvektoren $\vec{v}$ til den normaliserede vektor mellem det aktuelle gitterpunkt

på billedplanet og kameraets udgangspunkt. Der dannes altså en stråle for hver pixel i outputbilledet.

4.2.4 Skæring med figurer

For at finde farven i en enkelt pixel undersøges hvad den tilsvarende stråle eventuelt måtte skære. Skæringer mellem givne figurer og stråler findes ved at indsætte *parameterfremstillingen* for linien i parameterfremstillingen for figuren. Afhængig af figuren findes et antal skæringer med strålen. I det følgende er skæring mellem stråle og kugle vist.

Skæring med kugle

For en linie l:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + t \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix}$$

og en kugle defineret ud fra en radius r og et centrum $p = \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix}$ kan en evt. skæring beregnes ved løsning af:

$$Ax^2 + Bx + C = 0, hvor$$

$$\begin{aligned} A &= t^2(\Delta x^2 + \Delta y^2 + \Delta z^2) \\ B &= 2t(\Delta x(x_0 - xc) + \Delta y(y_0 - yc) + \Delta z(z_0 - zc)) \\ C &= (x_0 - xc)^2 + (y_0 - yc)^2 + (z_0 - zc)^2 - r^2 \end{aligned}$$

Bemærk at $A = 1$ eftersom liniens vektor $\vec{v}$ er en enhedsvektor, og derfor har længden 1.

Diskriminanten d er lig

$$d = B^2 - 4C$$

Hvis d er negativ, skærer strålen ikke kuglen. Ellers beregnes indgangs- og udgangs-skæringspunktet:

Løsningen for ligningen

$$t_0 = \frac{-B - \sqrt{(B^2 - 4 * C)}}{2}$$
$$t_1 = \frac{-B + \sqrt{(B^2 - 4 * C)}}{2}$$

t_0 og t_1 er da henholdsvis den første og anden skæring mellem strålen og kuglen.

Listing 4.4: Pseudokode til at finde farven i et givent pixel

```
trace( stråle )
    skæring = find nærmeste skæring i scenen

    hvis skæring
        farve = shade( skæring )

    returner farve
```

Når samtlige skæringer med en given figur og strålen er fundet, vælges den skæring som er tættest på kameraets udgangspunkt.

Afsnit 5

Belysning

Farver i outputbillede er repræsenteret ved *RGB modellen*, hvor tre grundfarver kombineres til alle de farver modellen er i stand til at vise. De tre grundfarver i modellen er *rød*, *grøn* og *blå*, hver med en RGB farve på henholdsvis (1,0,0), (0,1,0) og (0,0,1) [FvDFH92]. Sort er defineret som (0,0,0) og hvid som (1,1,1).

RGB modellen er ikke i stand til at vise alle de farver som det menneskelige øje er i stand til at opfatte [FvDFH92], men er nem at bruge da det er den farvemodel som såvel det underliggende styresystem samt computerhardwaren bruger.

En lyskildes *intensitet* kan da angives som en RGB farve - eksempelvis gul (1,1,0) - samt en overordnet intensitet, eksempelvis 10.000, som ganges på hver komponent i farven.

Farven på et givent punkt på en figurs overflade afhænger af den samlede *lystransport* i punktet, altså hvor meget lys der fra punktet transporteres imod strålen hvormed vi undersøger punktet. Lystransport kan inddeles i fire katagorier: *spekulær reflektion*, *diffus reflektion*, *spekulær transmission* og *diffus transmission*.

Diffus lystransport spreder lys, hvilket resulterer i matte overflader, hvorimod spekulær lystransport blot spejler lys, og derfor resulterer i blanke overflader. En perfekt diffus overflade vil derfor være helt mat, hvorimod en perfekt spekulær overflade vil være spejlblank. Ofte har overflader dog en kombination af både diffus og spekulær belysning.

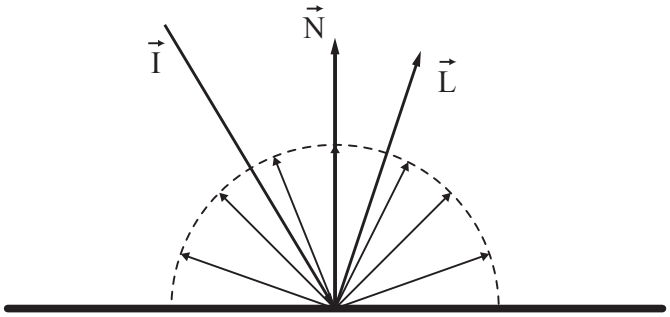
5.1 Diffus reflektion

Diffus reflektion er en grov model for hvad der sker, når et foton rammer et ru materiale og absorberes. Det atom i materialet, som absorberede fotonet, vil -

under de rette betingelser - emittere et nyt foton i en tilfældig retning væk fra overfladen.

En diffus overflade bliver i princippet oplyst fra alle sider. Dog skygger selve figuren for den belysning der måtte komme bagved overfladen. Derfor begrænses det område som overfladen modtager lys fra til den del af scenen som normalvektoren peger imod.

Den mængde af lys, der rammer en overflade og reflekteres diffust, afhænger udelukkende af overfladens orientering i forhold til lyskilden. Hvis overfladen står vinkelret på lyskilden reflekteres den fulde styrke af lyskilden. Efterhånden som vinklen mellem lyskilden og normalvektoren vokser, aftager den mængde lys der reflekteres indtil normalen står vinkelret på lyskilden, eller vender væk fra lyskilden. Da vil lyset ikke bidrage til belysningen af overfladen [Gla89].



Figur 5.1: Diffus reflektion

$\vec{I}$ er indfaldsvektoren

$\vec{L}$ er lysvektoren

$\vec{N}$ er normalvektoren

Retningen på den diffust reflekterede stråle ligger som tidligere nævnt inden for en halvkugle der peger væk fra overfladen. Den nærmere retning kan findes, hvis ϵ_1 og ϵ_2 er to vilkårlige tal inden for 0.0 og 1.0:

$$\vec{v} = (\theta, \phi) = (cos^{-1}(\sqrt{\epsilon_1}), 2\pi\epsilon_2)[\text{Jen01}]$$

Bemærk at ovenstående er udtrykt i polære koordinater, hvor θ angiver *vinkelrotationen* rundt om normalvektoren, og ϕ vinklen mellem normalvektoren og den reflekterede stråle.

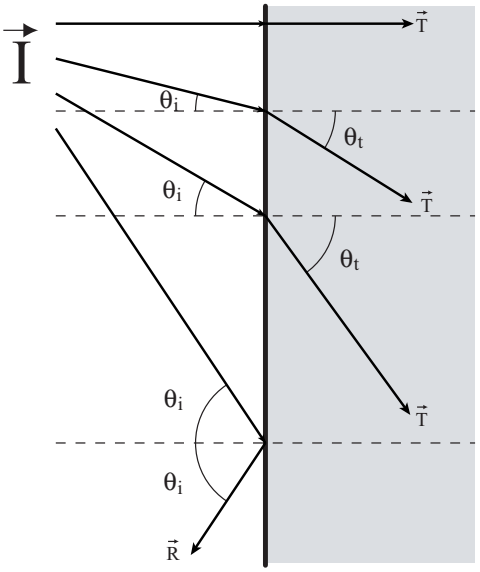
5.2 Spekulær reflektion

Spekulær reflektion er perfekt spejling. Det vil sige, at vinklen mellem lysets *indfaldsvektor* $\vec{I}$ og *normalvektoren* $\vec{N}$ er den samme som vinklen mellem den *reflekterede vinkel* $\vec{R}$ og *normalvektoren* $\vec{N}$. Der henvises til [Gla89] for udledningen af reflektionsvektoren $\vec{R}$.

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

5.3 Spekulær transmission

Spekulær transmission er en model for lystransport igennem en figur. Når lys passerer fra et medie til et andet, eksempelvis luft og glas, kan lysets retning afbøjes. Dette skyldes, at lyset hastighed afhænger af det medie det bevæger sig i. Forholdet mellem lysets hastighed i vacuum kaldes mediets *brydningsindex*. Forholdet mellem to mediers brydningsindex bestemmer hvor meget lysets retning afbøjes ved passagen ind i det nye medie. Dog kan lyset reflekteres fra grænsefladen mellem medierne, hvis vinklen mellem indfaldsvektoren og overfladenormalen er større end *den kritiske vinkel*. Dette betegnes *total intern reflektion*. Da betragtes reflektionen som en spekulær reflektion. Der henvises igen til [Gla89] for udledningen af den transmitterede vektor $\vec{T}$ så vel som den kritiske vinkel.



Figur 5.2: Brydning af lys

$$n_{it} = \frac{\sin(\theta_t)}{\sin(\theta_i)}$$

$$r = 1 + n_{it}^2((\vec{N} \cdot -\vec{I})^2 - 1)$$

$$\vec{T} = n_{it}\vec{I} + (n_{it}(\vec{N} \cdot -\vec{I}) - \sqrt{r})$$

Hvis r er negativ optræder der total indre reflektion frem for transmission.

5.4 Praktisk belysning

I afsnit 5 blev der præsenteret forskellige slags lystransport, der kombineret, resulterer i visuelt flotte billeder. Desværre er der en række ting, der gør, at det ikke er ligetil at implementere en belysningsmodel baseret udelukkende på det førnævnte. Den umiddelbar indlysende måde at beregne belysningen i et skæringspunkt ville være at undersøge lysbidraget fra spekulære og diffuse stråler sendt ud fra punktet. Men eftersom de eneste lyskilder, der er defineret er punktlys, er det ikke muligt at ramme et lys med en stråle, og derfor er det ikke muligt at beregne lysbidraget direkte fra lyskilder.

Et andet stort problem er det generelle lysbidrag fra diffuse reflektioner, altså både fra egentlige lyskilder samt eventuelle reflektioner fra andre figurer. Den samlede diffuse belysning fra en overflade er summen over alt det indkommende lys der reflekteres diffust fra overfladen. En naiv måde at finde den indkommende belysning ville være, at kaste et uendeligt antal stråler fra overfladen ud i scenen for at bestemme hvormeget lys der reflekteres hen imod overfladen fra omgivelserne

I de følgende afsnit præsenteres hurtigere metoder til at beregne de forskellige former for lystransport.

Spekulær reflektion og transmission håndteres som tidligere nævnt ved at skyde en ny stråle i en given retning og undersøge hvad den rammer.

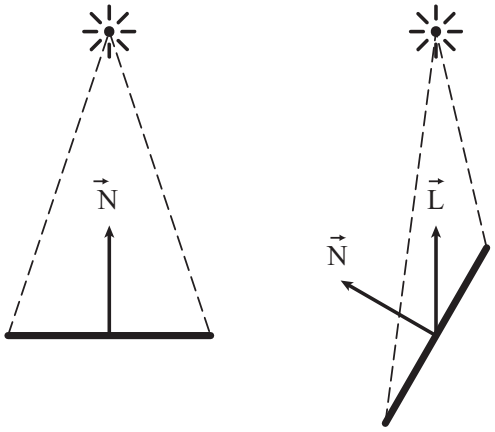
Hver type lystransport har en koefficient tilknyttet. Hvis der ønskes billeder der, opfylder de fysiske love for energibevarelse, bør summen af den diffuse reflektionskoefficient ($k_d r$) og den spekulære reflektionskoefficient (k_{sr}) være ≤ 1.

5.4.1 Diffus belysning

I stedet for at beregne diffus belysning ved at følge egentlige diffust reflekterede stråler, kan effekten delvis efterlignes ved udelukkende at undersøge de lyskilder, som er direkte synlige fra det skæringspunkt, man er interesseret i at belyse. Derved er det muligt at beregne mængden af lys, der kommer direkte fra egentlige lyskilder, som reflekteres diffust fra skæringspunktet. Dette tager kun højde for

direkte diffus belysning fra lyskilder og ignorerer fuldstændig diffus belysning fra andre nærliggende figurer, men giver alligevel et rimelig resultat.

Den mængde af lys som en lyskilde reflekterer i et givet punkt afhænger udelukkende af overfladens orientering i forhold til lyset, da det reflekterede lys distribueres over hele halvkuglen ud fra overfladen. Intensiteten af det lys der reflekteres er givet ved $\cos(\theta)$, hvor θ er vinklen mellem normalvektoren og lysvektoren. Dette kan også skrives som $\vec{N} \cdot \vec{L}$, hvis $\vec{L}$ og $\vec{N}$ er enhedsvektorer. At forholdet afhænger af vinklen, kan ses af figur 5.3, hvor et givent areal på en overflade betragtes. Hvis overfladen vender direkte mod lyset, vil hele intensiteten af lyset reflekteres. Hvis overfladen derimod vender lidt væk fra lyskilden, vil den overflade som lyset rammer være mindre fra lysets synspunkt, hvorfor mindre lys vil blive reflekteret fra overfladen. Det bør bemærkes, at der ikke reflekteres noget lys fra en lyskilde der har en vinkel på 90°eller mere. [Gla89]



Figur 5.3: Diffus reflektion

Lysets farve filtreres med figurens farve. Et hvidt lys (RGB (1,1,1)) reflekteret i en grøn overflade (RGB (0,1,0)) vil resultere i grønt lys. Gult lys (RGB (1,1,0)) reflekteret i en lilla overflade (RGB (0.8,0,1)) vil resultere i rødt lys (RGB (0.8,0,0)).

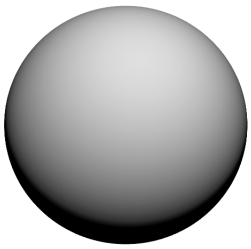
Det diffust reflekterede lys fra en enkelt lyskilde kan skrives som:

$$I_{dr} = k_{dr} * I_i * C * (\vec{N} \cdot \vec{L})$$

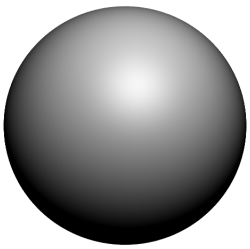
hvor I_{dr} er det reflekterede lys fra den diffuse reflektion, k_{dr} er den diffuse reflektionskoefficient, I_i er incident lyset fra lyskilden, C er objektets farve og $\vec{N} \cdot \vec{L}$ er cosinus til vinklen mellem de to vektorer.

Listing 5.1: Pseudokode til beregning af farven i et skæringspunkt med diffus belysning

```
shade( skæring )
```



Figur 5.4: Diffust belyst kugle



Figur 5.5: Spekulær kugle

```
    normal = find normal i skæring
    for alle lys i scenen
        hvis lyset er synligt fra skæringspunktet
            lysvektor = enhedsvektor der peger fra skæring mod lyskilden
            farve += beregn diffus farve

    returner farve

beregn diffus farve
    farve = kdr * lysintensitet * (normal * lysvektor)
     filtrer farve efter figurens farve
    aftag farve med afstanden fra skæring til lys
    returner farve
```

5.4.2 Spekulær reflektion af figurer

Spekulær reflektion af figurer fungerer som tidligere beskrevet i afsnit 5.2. Reflektionsvektoren $\vec{R}$ findes ud fra, at der skal være samme vinkel mellem den og normalvektoren $\vec{N}$ som der er mellem incidentvektoren $\vec{I}$ og $\vec{N}$. Der dannes en stråle med udgangspunkt i reflektionen og retning $\vec{R}$, og denne stråle spores videre. Den farve, der findes ved at spore denne stråle, adderes til den aktuelle skæring, vægtet med k_{sr}. [Gla89]

Listing 5.2: Pseudo kode til simpel raytracer

```
shade( skæring )
    normal = find normal i skæring
    beregn reflektionsvektor

    for alle lys i scenen
        hvis lyset er synligt fra skæringspunktet
            lysvektor = enhedsvektor der peger fra skæring mod lyskilden
            farve += beregn diffus farve

    farve += ksr * trace( reflektion )

    returner farve
```

5.4.3 Spekulær reflektion af lyskilder

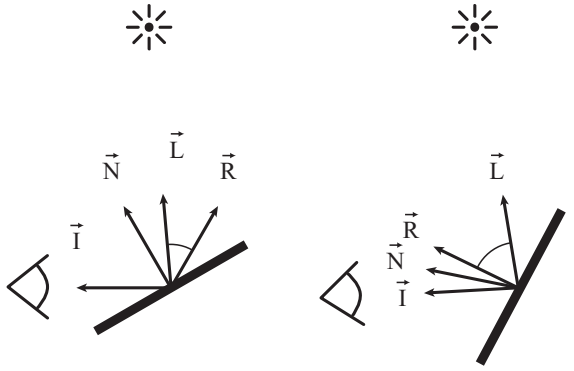
Spekulær belysning er et forsøg på at modellere genskin fra lyskilder. Da det som tidligere beskrevet ikke er muligt at ramme en lyskilde med en stråle og derved tegne selve lyskildens reflektion, må andre metoder tages i brug. I stedet for at forsøge på at ramme lyskilder, betragtes vinklen mellem lysvektoren $\vec{L}$ og reflektionsvektoren $\vec{R}$. Når vinklen imellem de to vektorer er meget lille, må lyset komme fra nogenlunde samme retning som reflektionsvektoren. Lysets intensitet $\vec{I_i}$ adderes til skæringsfarven efter af være blevet vægtet med k_{sr}.

$$I_{sr} = k_{sr} * I_i * (\vec{R} \cdot \vec{L})$$

Dette vil resultere i et meget blødt højlys i retning af lyset. Højlyset kan koncentreres ved at opløfte prikproduktet i en potens, således at den værdi som lysets intensitet I_i skaleres med er væsentlig mindre end 0 undtagen lige omkring reflektionspunktet. Denne model kaldes for *Phong* modellen, eksponenten i ligningen kaldes *phong-eksponenten*.

$$I_{sr} = k_{sr} * I_i * (\vec{R} \cdot \vec{L})^n$$

hvor I_{sr} er det reflekterede lys fra den spekulære reflektion, k_{sr} er den spekulære reflektionskoefficient, I_i er lyskildens intensitet, $\vec{R}$ er reflektionsvektoren, $\vec{L}$ er lysvektoren og n er phong-eksponenten.



Figur 5.6: Spekulær figur

Listing 5.3: Pseudokode til beregning af spekulær belysning

```
shade( skæring )
    normal = find normal i skæring
    reflektion = beregn reflektionsvektor

    for alle lys i scenen
        hvis lyset er synligt
```

```
            lysvektor = enhedsvektor mod lyskilden
            farve += beregn diffus farve
            farve += beregn spekulær reflektion højlys

    returner farve

beregn spekulær reflektion højlys
    farve = ksr * lysintensitet * ( reflektionsvektor * lysvektor)^n
    aftag farve med afstanden fra skæring til lys
    returner farve
```

5.4.4 Transmission

Beregning af transmission minder om reflektion. Den eneste forskel er at transmissionsvektoren $\vec{T}$ bruges i stedet for reflektionsvektoren $\vec{R}$. Beregningen af $\vec{T}$ er vist foroven

```
shade( skæring )
    normal = find normal i skæring
    reflektion = beregn reflektionsvektor
    transmission = beregn transmissionsvektor

    for alle lys i scenen
        hvis lyset er synligt fra skæringspunktet
            lysvektor = enhedsvektor der peger fra skæring mod lyskilden
            farve += beregn diffus farve
            farve += beregn spekulær reflektion højlys
            farve += beregn spekulær transmission højlys

    farve += ksr * trace( reflektion )
    farve += kst * trace( transmission )

    returner farve

beregn transmissionsvektor
    hvis intern reflektion
        returner reflektion
    ellers
        returner transmission

beregn spekulær transmission højlys
    farve = kst * lysintensitet * (transmissionvektor * lysvektor)^n
    aftag farve med afstanden fra skæring til lys
    returner farve
```

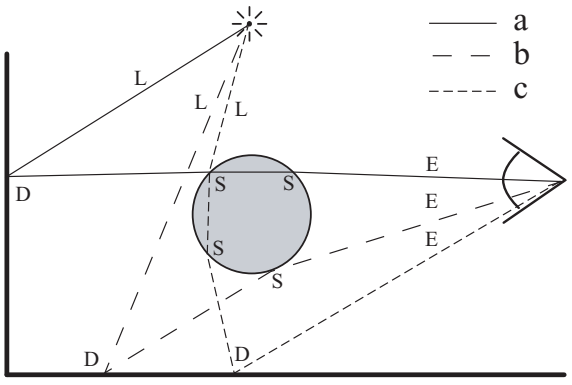
Afsnit 6

Global belysning

Almindelig rekursiv baglæns raytracing er ikke i stand til at simulere alle former for belysning, f.eks. kan den ikke håndtere indirekte belysning og lyskoncentration. Der findes flere metoder til at simulere denne slags belysning, de vil kort blive gennemgået her.

6.1 Notation

Heckbert introducerede[Hec90] en notation for hvilke "stier" en stråle kan tage i en scene.



Figur 6.1: Lys stier i en scene

De forskellige typer af stråler er:

L en stråle fra en lyskilde

E en stråle til kameraet

S en stråle der reflekteres eller transmitteres perfekt

D en stråle der reflekteres eller transmitteres diffust

De kan alle kombineres som regulære udtryk, vha operatorerne `+`, `*`, `?` og `|` således:

`k+` en eller flere k

`k*` ingen eller flere k

`k?` ingen eller en k

`k|v` enten k eller v

I bilag C på side 122 kan desuden findes en syntaxbeskrivelse for regulære udtryk på Backus-Naur form.

På figur 6.1, er kuglen spekulær og væggen og gulv er diffus. De optegnede lysstier a,b,c kan ud fra Heckbert noteres:

a `LDSSE`

b `LDSE`

b `LSSDE`

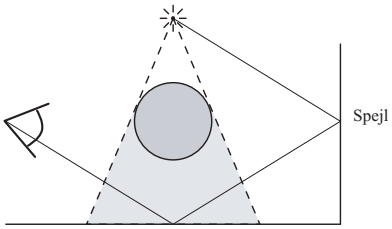
Almindelig raytracing simulerer `LD?S*E`, hvilket betyder at den kan følge stråler baglæns fra kameraet(`E`) til lyskilden(`L`), via ingen eller flere perfekt reflekterende eller transmitterende overflader(`S*`) samt en eller ingen diffus overflade(`D`). `D?` er her den direkte belysning på en diffus overflade.

6.2 Problemer i raytracing

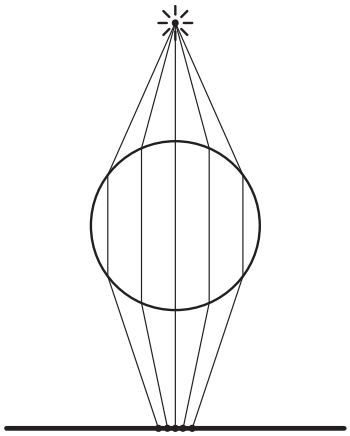
Det er med almindelig raytracing ikke muligt at gengive indirekte belysning fra diffuse overflader da det vil kræve at der blev fulgt stråler videre baglæns fra diffuse overflader. Eller der blev fulgt stråler fra lyskilden via diffuse samt perfekte reflektioner/transmissioner. Problemet opstår fordi man ved almindelig raytracing, ikke kan beregne det lysbidrag der måtte være fra en lyskilde reflekteret i en anden overflade. Dette er illustreret på figur 6.2.

Det er ej heller muligt at simulere fokusering af lys i baglæns raytracing bla. p.ga. brugen af punktlys. Skulle lyset under en kugle fokuseres som vist på fig.

6.3, ville det kræve at strålen blev fulgt tilbage igennem kuglen, og ramte et uendeligt lille punkt (lyset) efter at være blevet afbøjet to gange. Dette vil i praksis ske så få gange at det ikke vil kunne ses.



Figur 6.2: Indirekte belysning



Figur 6.3: Fokusering af lys

Det vil være nærliggende at foreslå forlæns raytracing som en løsning på problemet omkring indirekte belysning, det vil da være muligt at få beregnet alle lysbidrag i scenen. Men det vil kræve mange stråler fra lyskilden før bare en af dem rammer kameraet, dette vil medføre at der beregnes en masse stråler som slet ikke giver et bidrag til det færdige billede. En tilfredsstillende løsning være en hvor der til dels løser de to ovenstående problemer og samtidig ikke laver unødigt mange beregninger.

6.3 Rendering equation

Rendering equation er den matematiske basis for alle global belysnings algoritmer. Den opstiller en model for lysets interaktion med overflader i en scene. For en udledning af ligningen se evt [Lás00].

Hvis en scene betragtes som et lukket system, hvor alle lystransporter skal være i ligevægt, vil den samlede udgående radiance L_o fra punktet x være:

$$L_o(x, \vec{w}) = L_e(x, \vec{w}) + L_r(x, \vec{w}) \qquad \text{[Jen01]} \tag{6.1}$$

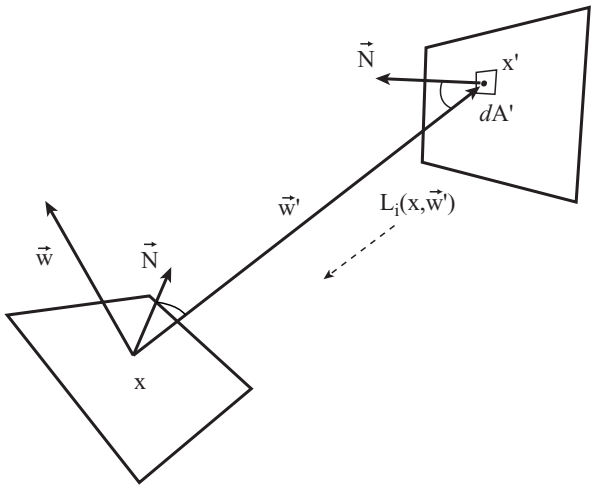
L_e er den udsendte radiance fra overfladen i x

L_r er den reflekterede radiance i x

$\vec{w}$ er en vektor til kameraet

x er skæringpunktet med objektet

(Se figur 6.4).Det betyder kort sagt, at summen af det lys som udsendes fra x plus det der modtages via reflektioner i x er lig det totale lys fra x. Den reflekte-



Figur 6.4: Render Equation

rede del kan skrives som integralet over alle reflektioner i punktet x i retningen $\vec{w}'$. $f_r(x, \vec{w}', \vec{w})$ er *BRDF funktionen*. BRDF står for *Bidirectional Reflectance Distribution Function* og den beskriver forholdet mellem indkomne lys og det udgående lys for en given overflade.

$$L_r = \int_S f_r(x, \vec{w}', \vec{w}) L_i(x, \vec{w}')(\vec{w}' \cdot \vec{n}) d\vec{w}' \qquad \text{[Jen01]} \tag{6.2}$$

L_i er radiance fra x' mod x hvor x er det oprindelige skæringspunkt og x' er et nyt skæringpunkt der reflekteres i x.

Ligningen kan omskrives således at ledet $V(x, x')$ angiver en "synligheds" funktion der, hvis x kan se x' er lig 1 og hvis de ikke kan se hinanden er 0. Det er derfor denne del der afgør om punktet modtager direkte belysning.

$$L_r = \int_S f_r(x, \vec{w}', \vec{w}) L_i(x, \vec{w}') V(x, x') G(x, x') dA \qquad \text{[Jen01]} \tag{6.3}$$

$L_i(x, \vec{(w)})$ er, jvf figur 6.4, lig med den udgående radiance fra punktet x' altså $L_o(x', \vec{(w)}')$. $L_i(x', \vec{(w)}')$ er så igen lig med den udgående radiance i et andet punkt x'' altså $L_o(x'', \vec{(w)}'')$ og så fremdeles.

Der ønskes en metode til at evaluere netop denne "sti" gennem scenen.

6.4 Path Tracing

Path tracing er en udvidelse af almindelig raytracing, hvor samtlige lysstier i en scene følges. Dette gøres ved at følge *alle* mulige reflektioner og transmissioner, også diffuse. Dvs der følges stråler på formen `L(S|D)*E`. Men da en diffus overflade reflekterer i alle retninger væk fra overfladen, vil der være et uendeligt antal stråler der skal følges i scenen.

En måde at løse dette på, er at approximere den ved såkaldte *Monte Carlo integration*. Bilag B indeholder en kort beskrivelse af Monte Carlo integration. Navnet *Path Tracing* kommer af, at metoden danner en sti gennem scenen.

Der følges kun en enkelt stråle videre fra hver reflektion uanset materialet. Der benyttes en sandsynligheds funktion for hvilken type af reflektion der skal følges. Denne funktion afhænger af overfladens materiale. F.eks. vil en 80% diffus overflade have 80% sandsynlighed for at følge en diffus stråle videre. Det vil samtidig medføre netop at den overflade vil have en overvægt at diffust reflekterede stråler når den rammes af mange stråler.

For at få et så præcist billede som muligt skal der sendes mange stråler gennem hver punkt i kameraet. Det skyldes at Monte Carlo integration har en standard afvigelse på $\frac{1}{\sqrt{N}}$ hvilket vil komme til udtryk som støj på billedet, hvis ikke antallet af pixels sendt ind i scenen er stort nok. Ofte skal der mellem 1000 og 10000 stråler til for at få et "støjfrit" billede.

6.5 Radiosity

For at løse problemet omkring indirekte belysning er det nærliggende at udføre forlæns raytracing. Det er mere eller mindre hvad radiosity gør.

Ved radiosity udsendes stråler fra lyskilden og intensiteten af hver stråle afsættes i et net udspændt over alle figurers overflade. Derefter kan lysintensiteten til et givent punkt bestemmes umiddelbart.

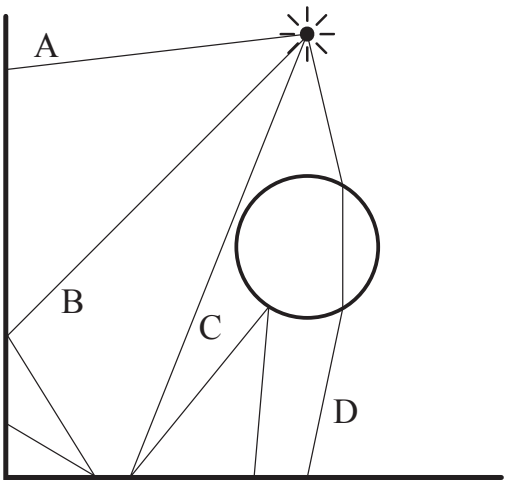
Metoden er god til scener der kun indeholder diffuse overflader da de varierer langsomt. Men jo mere kompleks og jo flere ikke diffuse overflader jo dårligere bliver radiosity, da nettet for reflekterende overflader skal være langt mindre masket end ved diffuse da genskin og højlys varierer meget over små afstande. Da nettet afhænger af geometrien vil nettet bliver finere og finere jo mere detaljerede objekterne bliver. Dette bevirker samtidig af hukommelseskravet stiger i takt med scenernes kompleksitet. Radiosity giver ingen løsning på problemet omkring fokusering af lys.

6.6 Photon Mapping

Photon Mapping er basalt set en forlæns raytracer. Den "skyder"fotoner afsted fra lyskilderne i scenen, og der hvor de rammer et objekt i scenen gemmes information om fotonens styrke, retning og skæringspunkt med objektet. I modsætning radiosity gemmer Photon Mapping denne information i en særskilt datastruktur, uafhængig af geometrien.

Derved kan lyset for et vilkårligt punkt i scenen estimeres ved at tage gennemsnittet af de nærmeste fotoners intensitet. På samme måde som ved radiosity det at der sendes mange fotoner afsted fra hver lyskilde for at kunne gengive perfekte reflektioner og transmissioner. Derfor benyttes baglæns raytracing til netop det den metode er god til: reflektioner og transmission. PhotonMappet benyttes herefter til de diffuse reflektioner og transmissioner.

Dette gøres ved at der i Photon Mappet kun gemmes fotoner, der rammer en diffus overflade. Ved perfekt reflektion eller transmission sendes fotonen blot videre. Ved en diffus overflade benyttes en sandsynlighedsfunktion baseret på en vilkårlig BRDF til at afgøre om fotonen skal absorberes eller reflekteres diffust. Ved diffus reflektion sendes fotonen videre i en tilfældig retning.



Figur 6.5: Photon paths fulgt af Photon Mapping

PhotonMap metoden følger eksempelvis stråler på følgende former, jævnfør figur 6.5

A `LD`

B `LDDD`

C `LDSD`

D `LSSD`

Da både indirekte belysning og lysfokusering bliver omfyldt er Photon Mapping en god løsning på problemerne.

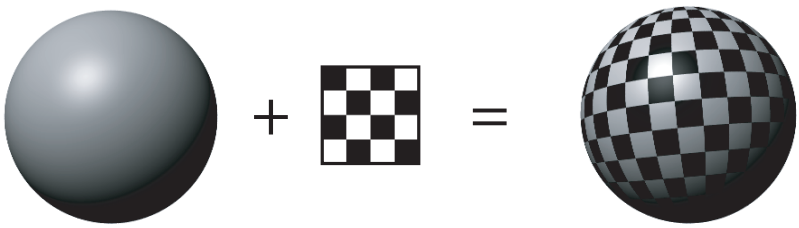
Når PhotonMappet er opbygget udføres en Monte Carlo eller almindelig baglæns raytracing hvor L_e delen af ligning 6.1 beregnes ved opslag i PhotonMappet. Resten beregnes vha almindelig raytracing. Det betyder at photon mappet benyttes til at beregne stråler på formen `L(S|D)*D`, hvorefter en almindelig raytracer benyttes til at beregne stråler på formen `(LS*E)|(DS*E)`.

Afsnit 7

Overflade egenskaber for figurer

Overfladen på en figur definerer hvorledes den fremtræder på billedet. Lysintensiteten i hvert skæringspunkt er summen af op til (i vores tilfælde) fire forskellige former for lystransport: Diffus belysning, genskin, spejling og transmission. Yderligere er det muligt enten at definere en konstant farve for hele objektet, eller at strække et billede (en såkaldt *tekstur*) ud over objektets overflade.

7.1 Teksturer



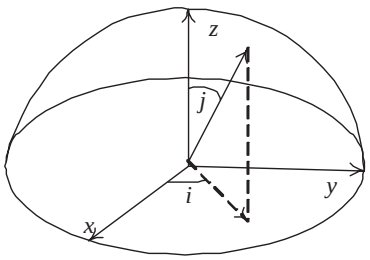
Figur 7.1: Teksturering af kugle med skaktern

Teksturer kan opfattes som et "tapet"man dækker sine objekters overflade med. Det er en let og hurtig måde at tilføje detaljer til et objekts overflade. se på figur 7.1.

Da overfladen for en figur er 2-dimensionel og et billede også er 2-dimensionelt, kræver teksturering ingen transformation i dimensioner. Dog skal man finde en måde at beskrive et punkt på figurens overflade, således at man kan finde det tilsvarende punkt på figurens tekstur.

Derfor man to koordinatsystemer, et UV koordinatsystem for figuren, hvor koordinaterne ligger inden for (0,0) og (1,1) og et ST koordinatsystem hvor koordinaterne ligger inden for teksturens dimensioner i pixels, eksempelvis (0,0) og (255,255). Figurens farve findes ved at beregne UV koordinatet i skæringspunktet, skalere disse efter teksturens bredde og højde, og til sidst slå farven for det pågældende punkt op i teksturen. Algoritmen til at beregne UV-koordinater afhænger af figur-typen.

Som eksempel kan man betragte en kugle. For at lave UV koordinatsystemet, må man have tre på hinanden retvinklede vektorer. Disse vektorer definerer et internt 3-dimensionalt koordinatsystem i kuglen, dvs. en x, en y og en z akse. Hvert punkt på kuglen kan så beskrives ved to vinkler; i mellem x og y akserne, og j mellem z og normalvektoren til punktet (vektoren defineret ved skæringspunkt fratrukket centrum af kuglen). Disse vinkler svarer til højde og breddegrader på en globus. Hvis vinklerne er i radian dividerer man i med 2π og j med π, således at begge er mellem 0 og 1.



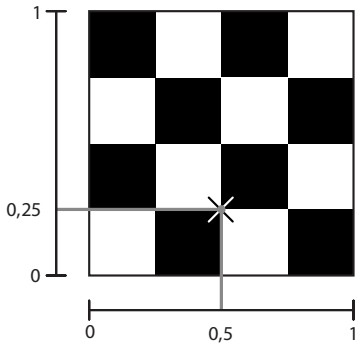
Figur 7.2: Vinklerne i og j

Hvis man vil lægge sin tekstur flere gange, som i figur 7.1, skal man lade i og j gå over flere perioder. Med kvadratiske billeder, som man ikke ønsker strakt, kan man lade i gå over dobbelt så mange perioder som j, da i beskriver kuglen hele vejen rundt, mens j kun beskriver den fra top til bund.

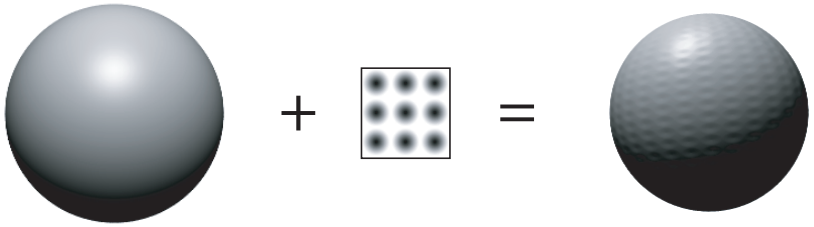
ST koordinaterne afhænger af billedet, som er valgt til tekstur. Hvis billedet er 600×480 pixels stort, og UV koordinaterne er henholdsvis $U = 0.5$ og $V = 0.25$, vil ST koordinaterne være $S = 600 \cdot 0.5 = 300$ og $T = 480 \cdot 0.25 = 120$. Dette aflæses på tekstur billedet, som den pixel, der er 300 pixels inde i bredden og 120 i højden, set fra nederste venstre hjørne.

7.2 Bumpmapping

Ud over normal teksturering, sker det at man ønsker at lave ujævne overflader på figurer, f.eks. en golfkugle (se fig 7.4). Som det er tilfældet med normale teksturer, vil det også her blive meget besværligt hvis man skulle modellere hver enkelt ujævnhed på selve objektet. I stedet kan man lægge et såkaldt *bumpmap* på figu-



Figur 7.3: Punktet, som ST koordinaterne svarer til



Figur 7.4: Fra glat kugle til golfkugle v.ha. bumpmap

ren. Bumpmappet fungerer som en slags højdekurve, der er lagt uden på figuren. Et bumpmap vil normalt være et gråtone-billede, hvor hvid repræsenterer høj og sort er lav. I praksis sker dette ved at der ved skæring med en figur, findes den tilsvarende skæringen i bumpmappet. Intensiteten for skæringspunktet, og punktet til højre for (S_h) og punktet neden for (S_n) slås op. Herefter dannes der to vektorer med koordinaterne

$$\begin{pmatrix} 0 \\ 1 \\ s \end{pmatrix} og \begin{pmatrix} 1 \\ 0 \\ t \end{pmatrix}$$

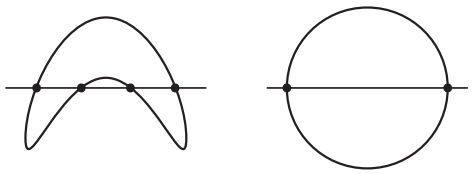
Hvor (s, t) ligger imellem $1, -1$ og repræsenterer højden i henholdsvis S_h og S_n i forhold til skæringen. Herefter udregnes krydsproduktet for de to vektorer, hvilket giver normalvektoren til punktet. Figurens normalvektor transformeres nu ind i et nyt system med de tre vektorer som base. Da normalvektoren bestemmer hvorledes lyset falder over et objekt, vil det resultere i den ønskede effekt af en ujævn figur, selvom der som sådan ikke er tale om at figuren ændres i form. Således vil et kugle stadig have et kuglerundt omrids.

Afsnit 8

Constructive Solid Geometry

De figurer, man kan konstruere i raytracing, er altid begrænsede af, at det skal være muligt at beregne skæring med deres overflade og en stråle. Dette kan gøre det svært og tidskrævende at lave mere komplekse figurer. Derfor indfører man Constructive Solid Geometry (CSG), som er en fleksibel måde at lave figurer, som består af kombinationer af flere figurere. Disse kaldes underfigurer. Der findes tre CSG-operationer: *forening*, *fællesmængde* og *differens*.

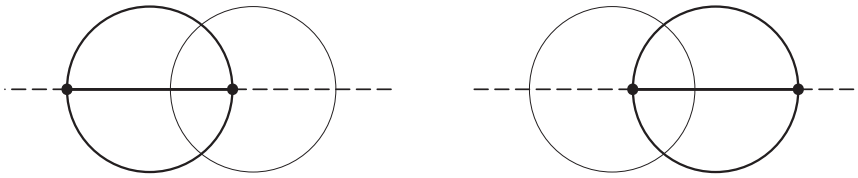
Alle figurer i raytraceren er lukkede figurer. Det betyder, at når en stråle skærer figuren vil der altid være et lige antal skæringer, da der altid vil være en udgang for hver indgang i figuren. Hvis figuren er konkav kan strålen skære ind i figuren flere steder, således at der optræder flere sæt af indgangs- og udgangsskæringer. Da kugler er konvekse, vil hver kugle kun have ét skæringssæt.



Figur 8.1: Henholdsvis konkav og konveks figur

8.1 Fællesmængde

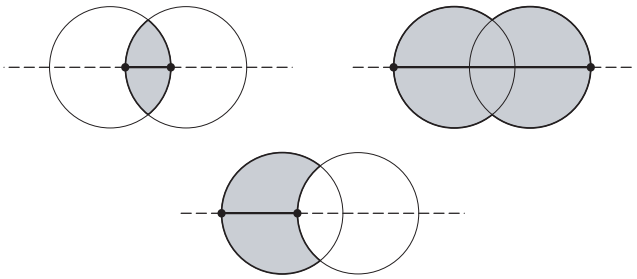
For at finde *fællesmængden* mellem to kugler sammenlignes hver kugles skæringssæt. Hvis de to kugler overlapper, fjernes de skæringer som ikke ligger inden for begge kugler (den første indgang og den sidste udgang). Hvis kuglerne ikke overlapper, er der ingen fællesmængde mellem de to figurer.



Figur 8.2: Skæring med CSG-figur. Først skæres den ene underfigur, derefter den anden.

8.2 Forening

To kuglers *forening* findes ved at fjerne eventuelle skæringer, som ligger midt inde i kuglerne - kun de yderste skæringer beholdes. Hvis kuglerne ikke skærer hinanden beholdes alle skæringerne.



Figur 8.3: De forskellige CSG-typer: fællesmængde, foreningsmængde, differens

Fælles for fællesmængde- og foreningoperationerne er, at rækkefølgen er ligegyldig. Fællesmængden mellem kugle A og B er den samme som fællesmængden mellem B og A.

8.3 Differens

Anderledes forholder det sig for *differens* operationen, eftersom man trækker én figurs volumen fra en anden. Differensen mellem to kugler, altså voluminet fra en kugle trukket fra en anden, findes ved at se bort fra skæringer inden i den kugle, som bliver trukket fra.

Det er interessant at bemærke, at en CSG-figur kan optræde som underfigur i en anden CSG-figur. Det er for eksempel muligt at trække en linse (fællesmængden mellem to kugler) fra en anden kugle, hvilket vil resultere i et linse-formet hak ind i den sidste kugle. Ved at kombinere flere CSG-figurer er det muligt at opbygge komplekse figurer.

Afsnit 9

Antialiasing

Figur 9.1: Et udsnit af en cirkelbue
med og uden anti-aliasing

Et gennemgående problem ved raytracing er, at man i raytracing forsøger at gengive en kontinuert verden v.ha. en diskret algoritme. Således vil en cirkel komme til at fremstå som en række punkter fremfor en jævn linje. Som et resultat af dette, vil man se meget bratte overgange i billedet (da man enten vil ramme en figur eller ikke), og der vil for alle linjer, der ikke er parallelle med skærmens x eller y akse, opstå "trappetrin". For at kompensere for dette kan man indføre *antialiasing* (fig:9.1).

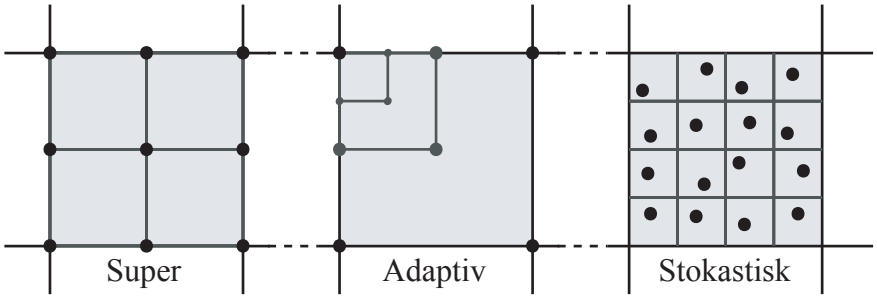
Antialiasing fungerer grundlæggende ved at lave stikprøver (samples) omkring hver enkelt stråle, der kastes fra kameraet. Gennemsnitsværdien af disse ekstra stråler afgører den enkelte pixels farve. På denne måde kompenseres der for den manglende håndtering af snitning af figure, og trappetrins-effekten udjævnes.

Der findes forskellige metoder til at undgå aliasing, der alle har det til fælles at der kastes flere stråler pr. pixel i billedplanet.

9.1 Super Sampling

Der lægges et gitter ud over den enkelte pixel. Ud igennem hvert knudepunkt i gitteret kastes en stråle. Det betyder i praksis, at man forøger opløsningen

for hver pixel med en konstant. Fordelen ved antialiasing vha. supersampling er at det er meget simpel at implementere. Problemet er at man kan risikere at ramme ved siden af linjer med et tilpas uheldig placering. Derved kan der opstå regelmæssige fejl igennem billedet.



Figur 9.2: Super Sampling, Adaptiv Sampling, Sokastisk Sampling

9.2 Adaptiv Sampling

Som før nævnt er det især i overgange aliasing bliver synligt. Det kan derfor være fordelagtigt at have ekstra høj opløsning i netop disse områder frem for andre. I adaptiv sampling kastes der en stråle igennem hvert hjørne af den ønskede pixel. Hvis resultatet af en af disse stråler varierer markant opdeles pixlen i 4 kvadranter. Processen gentages nu for kvadranten der har den markante stråle som hjørne. Fordelen er at man opnår den ønskede forhøjede opløsning i bratte overgange, og derved koncentrerer sig om kritiske områder. Problemet er dog det samme som ved super sampling, nemlig at der stadigvæk ikke tages højde for regelmæssige fejl.

9.3 Stokastisk Sampling

Stokastisk sampling er et forsøg på at tage højde for regelmæssige fejl i billedet. Pixlen inddeles som i Super Sampling i et gitter. For hver gitter-pixel kastes der en stråle, som kastes tilfældigt inden for den pågældende gitter-pixel. Fordelen ved dette er, at man ikke undersøger scenen med regelsmæssige mellemrum, og derved kan undgå uhensigtsmæssigheder [FvDFH92].

Del II

Dokumentation

Afsnit 10

Kravspecifikation

På baggrund af den gennemgåede teori udarbejdes følgende kravspecifikation:

Der skal implementeres en raytracer i Java. Programmet skal indeholde følgende:

Primitive figurer Følgende matematiske figurer implementeres:

- Kugle
- Plan
- Cylinder

Lysegenskaber for figur

- Spejling
- Transmission
- Diffusion

Tekstur Mulighed for teksturer. Teksturen indlæses fra en ekstern grafik-fil.

Bumpmap Mulighed for bumpmapping. Bumpmappet indlæses fra en ekstern grafik-fil.

CSG To figurer skal vha. CSG kunne sammensættes til én. Følgende CSG-metoder skal implementeres:

- Fællesmængde (Intersection)
- Foreningsmængde (Union)
- Differens (Difference)

Lys Lyskilder implementeres som punktlys

Antialiasing Det endelige billede skal kunne antialiases med stokastisk sampling

Global Illumination

- Photon mapping
- Monte Carlo path-tracing

Input Scenen defineres i en ekstern XML-fil der skal kunne indlæses af raytraceren. Desuden skal formatet være nemt at udvide i forbindelse med fremtidige udvidelser af raytraceren.

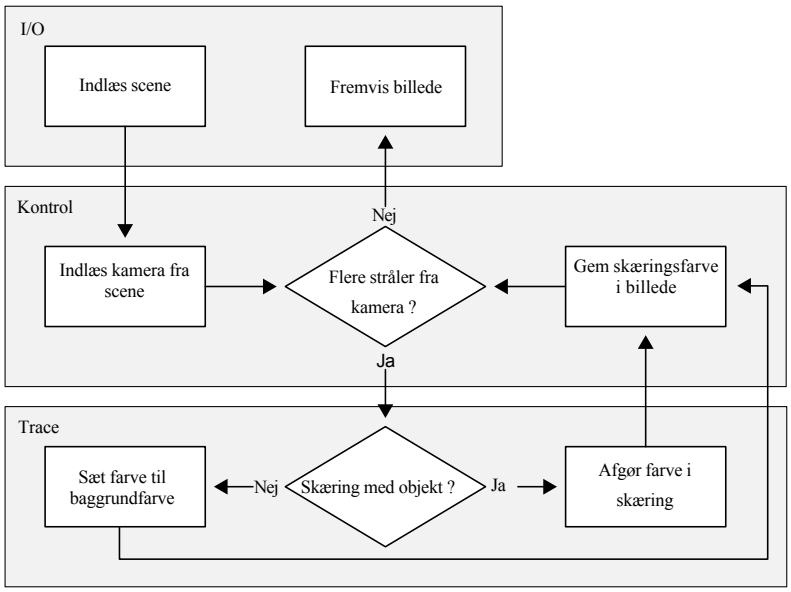
Output Det færdige billede skal dels kunne fremvises på skærmen, og dels kunne gemmes i en grafik-fil.

Udvidelser Programmet skal så vidt muligt være opbygget således at dele kan udskiftes og selve raytraceren udvides.

Brugergrænseflade Fra grænsefladen skal brugeren kunne vælge fra hvilken fil scenen skal indlæses, samt vælge hvorvidt billedet skal gemmes i en fil eller vises på skærmen.

Afsnit 11

Program Beskrivelse



Figur 11.1: Overordnet flowchart for raytraceren

Programmet er overordnet inddelt i tre dele: I/O(input/output), kontrol og tracer.

Programmet begynder med en grafisk brugerflade, hvor man kan vælge hvilken scene fil man vil indlæse, hvorvidt man vil bruge antialiasing, photon map og indstillingerne for dem. Endvidere skal man specificere hvilken opløsning det færdige billede ønskes i. Herefter kan man igangsætte selve raytracingen. I/O delen indlæser en scene, som indeholder figurer, lyskilder og et kamera.

Dernæst kører kontrol delen, som sørger for at indlæse den første stråle, der skal følges fra kameraet og at give den videre til trace delen. Tracedelen undersøger én stråle. Den undersøger hvorvidt denne stråle rammer nogle figurer. Hvis strålen

rammer en eller flere figurer, beregnes farven i skæringspunktet nærmest kameraet. Hvis strålen ikke rammer nogen figurer sættes farven til baggrundsfarven. Denne farve returneres til kontroldelen, som giver farven videre til I/O delen. Herefter indlæser den den næste stråle, der skal følges (såfremt der er flere) og giver den videre til trace delen.

I/O delen omdanner løbende de farver, den får, til et billede, som fremvises på skærmen. I/O delen giver endvidere mulighed for at gemme billedet, når det er færdigt, i en fil i filsystemet.

Ovenstående er vist i et flow-chart i figur 11.1.

Ud over de beskrevne dele, er der også en foton udsprednings del, som desværre ikke virkede da rapporten blev trykt. Denne del skulle sørge for at opbygge et Photon Map som

11.1 I/O

I/O delen styrer input og output. Elementerne i I/O delen består af følgende klasser:

I/O delen består af følgende klasser:

- Sceneloader
- DefaultOutput
- RaytracerDisplay

11.1.1 Sceneloader

For at programmet skal kunne forstå indholdet af en scenefil er der brug for en klasse, der kan oversætte indholdet til programmets interne scene repræsentation. SceneLoader klassen gør netop dette.

SceneLoader klassen bruger ElectricXML[Ele02] pakken, der er en lille og hurtig XML parser.

Klassen opretter et sceneobjekt indeholdende de i scenefilen definerede objekter, f.ek.s kamera, lys og objekter. Sceneobjektet opbygges fra buildScene metoden.

11.1.2 DefaultOutput

Den vigtigste del i DefaultOutput klassen er billedet, som er et array af farver. Dette array bliver initialiseret til at alle farverne er sorte, således at man altid kan hente billedet fra DefaultOutput, selv mens billedet genereres. DefaultOutput kan desuden gemme billedet i en fil.

11.1.3 RaytracerDisplay

Selvom denne klasse er del af I/O delen, sætter den alligevel et af kontrol delens klasser i gang. Dette er Raytracer klassen, som bliver behandlet under kontrol afsnittet.

RaytracerDisplay fremviser billedet til brugeren efterhånden som det bliver genereret. Dette gøres v.ha. raytracer-objektet RaytracerDisplay har modtaget ved instansiering. Vinduet er udstyret med en progressbar der viser status for optegningen. Når billedet er opbygget, kan RaytracerDisplay lagre billedet i filsystemet.

11.2 Kontrol

Kontrol-elementerne i raytraceren binder de resterende elementer sammen.

Klasserne i dette kontrol delen er:

- RaytracerGUI
- Raytracer

11.2.1 RaytracerGUI

Denne klasse fremviser brugergrænsefladen. Igennem dette kan brugeren vælge scenefiler og justere på de forskellige opsætninger for raytraceren. Når brugeren godkender disse opsætninger (ved tryk på render-knappen) startes selve raytraceren. Den valgte scenefil indlæses vha. et sceneloaderobjekt, et optionsobjekt instansieres og opsættes med de valgte opsætninger, og et output objekt instansieres. Disse objekter bruges endeligt til at instansiere et raytracerobjekt. Det klargjorte raytracerobjekt gives som argument til et displayobjekt, der startes.

11.2.2 RaytracerDisplay

RaytracerDisplay fremviser billedet til brugeren efterhånden som det bliver genereret. Dette gøres v.ha. raytracer-objektet RaytracerDisplay har modtaget ved instansiering. Vinduet er udstyret med en progressbar der viser status for optegningen. Når billedet er opbygget, kan RaytracerDisplay lagre billedet i filsystemet.

11.2.3 Raytracer

Raytraceren modtager et indlæst sceneobjekt, et outputobjekt samt et options objekt fra GUI'et.

Kameraet forespørges i sceneobjektet for et strålearray. Hver stråle i dette array traces og en farve for det pågældende array beregnes og gemmes. Kameraet forespørges igen, indtil det returnerer null. Herefter returneres de beregnende farver.

Listing 11.1: Pseudokode der viser Raytracer klassens render metode

```
render()
  gennemløb alle pixels
    bestem stråler for aktuelle pixel (via kameraet)

    gennemløb alle stråler for pixel
      bestem farven for den aktuelle stråle (via Render klassen)

    tag gennemsnittet af alle strålernes farve, og gem som aktuel pixel
```

11.3 Tracing

Trace delen sørger for at følge en stråle og afgøre hvilken farve den skal returnere.

Elementerne i I/O delen består af følgende klasser:

- Render
- Camera
- Object (figurer)
- Light
- Shader

Object er en fællesmængde for alle de klasser, som beskriver figurer. Fælles for dem alle er, at de arver fra AbstractObject. Shader klassen beskriver overfladeegenskaber for en figur. Til alle Object objekter, er der tilknyttet et Shader objekt.

Camera dækker over to klasser, DefaultCamera og RandomAntiAliasing. DefaultCamera er et standard kamera, der ikke tager højde for aliasing, og derfor kun genererer en stråle pr. pixel i outputbilledet. RandomAntiAliasing laver antialiasing via stokastisk sampling.

11.3.1 Render

Denne klasse tager imod en stråle fra raytracer klassen, der er et element fra kontroldelen. Denne stråle gives videre til alle figurerne, som returnerer de skæringspunkter de evt. måtte have. Renderklassen finder det skæringspunkt, som er nærmest kameraet, og giver det videre til den skårne figurs shader.

11.3.2 Camera

Dette er, som før nævnt klasserne DefaultCamera og RandomAntiAliasingCamera. Begge bruger som input en matrix, som beskriver placering og orientering, og en opløsning. For RandomAntiAliasingCamera kan man endvidere angive kvadratroden, af hvor mange gitterpunkter man ønsker hver pixel opdelt i, og hvor mange stråler man ønsker at sende ud for hver af disse gitterpunkter.

Begge klasser kan ud fra nummeret på en pixel i outputbilledet returnere den tilsvarende stråle.

11.3.3 Object

Disse klasser repræsenterer figurer. De skal ud fra en stråle kunne afgøre deres evt. nærmeste skæring. Endvidere skal de kunne afgøre normalvektoren og UV koordinatet til et skæringspunkt.

I object er endvidere nogle CGS figurer. En CSG figur kombinerer to figurer, og virker udadtil, som en normal figur.

11.3.4 Light

Beskriver lys. Består af en position og en farve.

11.3.5 Shader

Shaderens ansvar er at holde styr på alle de overfladeegenskaber en figur bliver tildelt. Endvidere skal den ud fra et skæringspunkt mellem en stråle og den figur den er tildelt, kunne beregne, hvilken farve denne figur har i dette skæringspunkt.

Til at beregne dette skæringspunkts farve bruger Shaderen andre klassers funktioner. Disse klasser er: Light, Object og Render. Light bruges til at finde ud af hvor lyskilderne er, og hvilken farve lys de udstråler. Dernæst opstilles en linie op til hver af lyskilderne og det undersøges via Object klasserne, hvorvidt der ligger figurer imellem skæringspunkt og lyskilden. Til alle former for overflader skal figurens normalvektor findes, hvilket Object bruges til. De overflader, der har tekstur og/eller bumpmap findes endvidere UV koordinaterne fra figuren, dvs. Object klasserne.

Såfremt figuren spejler udsendes en ny stråle i spejlingsretningen ved hjælp af klassen Render.

11.4 Værktøjer i programmet

Der er indført nogle ekstra typer, som er praktiske til raytracing. Disse er Vector, Matrix, Color, Ray og Intersection, som hver har deres klasse.

11.4.1 Vector klassen

Denne klasse bruges til at repræsentere vektorer, og metoderne er til at lave almindelige matematiske operationer på vektorer.

11.4.2 Matrix klassen

Dette er en matrice, som indeholder tre vektorer, forward, right og up, hvilket er matricens interne koordinatsystem, og svarer til koordinatsystemets x, y og z akse. Derudover indeholder det en stedvektor givet ved variablerne x, y og z.

Udover at indeholde mange informationer, er matricen også praktisk i forbindelse med rotation og flytning af figurer. Dette gøres ved at gange en vektor $\vec{v}$ eller en matrice N med en matrice M. Vektoren $\vec{v}$ eller matricen N bliver så roteret med forskellen mellem det globale koordinatsystem og det lokale koordinatsystem i M, hvorefter den bliver forskudt med stedvektoren i M.

Standard matricen har samme koordinatsystem som det globale og stedvektoren er $\vec{0}$.

En nærmere beskrivelse af matematikken bag rotation og forskydning med matricer findes i appendiks.

11.4.3 Color klassen

Klassen bruges til farver, og er opbygget af tre tal imellem 0 og 1, som repræsenterer rød, grøn og blå. Desuden indeholder den nogle metoder til operationer på farver.

11.4.4 Ray klassen

En Ray er en stråle. Den indeholder en stedvektor og en retningsvektor, som repræsenterer en parameterfremstilling for en linie. Derudover indeholder den to talværdier, som er brydningsindexet for det stof, den befinder sig i, og vigtigheden af strålen.

11.4.5 Intersection klassen

En Intersection indeholder en reference til en figur samt en afstand.

11.5 Beskrivelse af metoder for de vigtigste klasser

11.5.1 SceneLoader klassen

Syntaxen for scene formatet i en scene fil, er beskrevet i brugervejledningen. Klassen SceneLoader, sørger for at omdanne en scene-fil i XML format til objekter i programmet.

I klassen er en del metoder, som hver bruges til at oprette scenens indhold (f.eks. et Plane objekt), ud fra hvad der står i scene-filen. Da disse minder en del om hinanden er kun et af dem beskrevet.

Beskrivelse af metoder:

addPlane Denne metode tilføjer et Plane objekt og et tilhørende Shader til scenen ud fra oplysninger i scene filen.

Argumenter Element

Returnerer Void

buildScene Denne metode bruges til at søge igennem filen og finde de scene-elementer (kamera, lyskilder og figurer) i scenen-filen, som skal tilføjes scenen. Hver gang et scene-element findes, undersøges hvilken type det er, og derpå køres den metode i programmet, som opretter et objekt af den klasse, der svarer til scene-elementet.

Argumenter Ingen

Returnerer Scene

11.5.2 defaultOutput

Sørger for at indeholde billedet. Kan skrive billedet til en grafik fil i PNG format.

De vigtigste metoder i defaultOutput:

Konstructor Denne metode laver et billede, på den ønskede opløsning, og farver alle pixels i billedet sort, således at man kan vise billedet under optegnelsen.

Argumenter Options

write Kan skrive billedet til en grafik fil i PNG format.

Argumenter File

Returnerer void

getImage Returnerer billedet.

Argumenter Image

Returnerer Image

11.5.3 RaytracerDisplay

Denne klasse sørger for at starte Raytracer klassen og vise billedet efterhånden, som det optegnes.

Følgende metoder er centrale i klassen:

RaytracerDisplay Konstruktør, modtager et raytracerobjekt og lagre det. Fremviser grænsefladen.

Argumenter Et Raytracerobjekt

run Starter raytracerobjektet ved at kalde render() i klassen Raytracer. Hvert sekund kaldes getIndex i renderobjektet, og den returnerede værdi bruges til at opdatere progressbaren.

Argumenter Ingen

Returnerer Void

save Fremviser en filbrowser der giver brugeren mulighed for angive hvor han/hun ønsker at gemme outputbilledet. Billedet gemmes i formatet PNG.

Argumenter Ingen

Returnerer Void

11.6 RaytracerGUI

Laver den grafiske brugerflade.

Følgende metoder er centrale i klassen.

renderAction Aktiveres ved tryk på "render"-knappen i GUI'et. Undersøger om brugeren har valgt en scenefil, og om den i så fald eksistere. Hvis det er tilfældet kaldes render.

Argumenter Ingen

Returnerer Void

render Opretter et options-objekt med opsætninger valgt i GUI'et, opretter et sceneobjekt med den valgte scenefil, opretter et output objekt, og indlæser til sidst et Raytracerobjekt med disse objekter. Et RaytracerDisplayobjekt instansieres med Raytracerobjetet, og startes ved et kald til display.start.

Argumenter Ingen

Returnerer Void

11.6.1 Raytracer klassen

Raytracer klassen er den klasse, som er ansvarlig for at styre flowet i raytraceren.

Følgende metoder er centrale i klassen

Raytracer Konstruktør. Lagre sceneobjektet lokalt som scene. Kalder getBitmap () i outputobjektet, og gemmer resultatet lokalt som bitmap. Gemmer optionsobjektet lokalt som options.

Argumenter

- Et sceneobjekt
- Et outputobjekt
- Et optionsobjekt

render Kalder først PhotonTracer.emitPhotons fra PhotonTrace klassen der opbygger photonmappet. Herefter indhentes et array af rays ved at kalde scene.getCamera().getRaysForPixel(i). For hver stråle i arrayet kaldes Render .trace(scene, stråle) og resultatet gemmes i bitmap fra outputobjektet.

Argumenter Ingen

Returnerer Void

getIndex Returnere indexet på en pixel i outputbilledet der i kaldsøjeblikket bliver behandlet.

Argumenter Ingen

Returnerer Integer

11.6.2 Render

Sørger for at følge strålerne. Består reelt kun af en metode.

Trace Denne metode er statisk.

Ved hjælp af en scene og en stråle finder denne det nærmeste skæringspunkt med strålen.

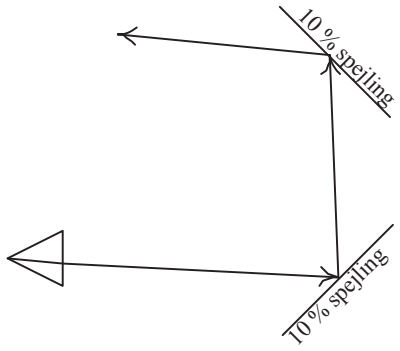
En stråle har et tal for hvor vigtig den er for den pixel, som undersøges. Dette tal er givet ved variablen importance. Eksempelvis vil en stråle, der er resultatet af to spejlinger på 10%, som vist på figur 11.2, udgøre 1% af den endelige pixel.

Såfremt en stråles importance er under $0,001$ vil Trace metoden returnere baggrundsfarven, som er givet ved scene.ambientColor.

Ved at kalde scene.getObjects() får den alle Object objekter (alle figurer), som den ligger i et array kaldet obj.

Dernæst følger en løkke:

```
for (int i = 0; i < obj.length; i++)
        {
            temp = obj[i].getNearestIntersection( ray );
```



Figur 11.2: Den øverste stråle vil blive undersøgt efter to spejlinger. Den har en vigtighedsfaktor på 1%.

```
        if ( temp.t < t )
        {
            nearestObj = temp.object;
            t = temp.t;
        }
    }
```

Metoden obj[i]. getNearestIntersection (ray) finder Object objekt nummer i i arrayet af Object objekter, og beregner denne figurs skæring med strålen Ray. Metoden returnerer det Object objekt, som der er skæring med og et tal, som er længden fra strålens udgangspunkt til skæringspunkt. Dette tal, temp.t sammenlignes med det tal t, som indtil videre er tættest på. Såfremt tallet temp.t er mindre end t, bliver t sat til temp.t.

Farven på det Object objekt, som er tættest på findes ved linien:

```
color = nearestObj.getShader().getIntersectionColor(ray, t);
```

Hvor det Shader objekt, der tilhører Object objektet, bestemmer farven ved metoden getIntersectionColor(ray, t).

Denne farve returneres. Hvis ingen Object objekteter skærer med linien, returneres baggrundsfarven.

Argumenter Scene, Ray

Returnerer Color

11.6.3 Camera

Der bliver brugt to forskellige kameraer, enten defaultCamera eller RandomAntiAliasingCamera , som begge arver fra AbstractCamera. Kameraet der bliver brugt skal lave strålerne,

som raytraceren skal følge. Det skal generere parameterfremstillinger, som er en stedvektor og en retningsvektor, for strålernes linier.

Da alle linier har samme udgangspunkt, er deres stedvektorer lig hinanden, hvilket også er stedvektoren i matricen, hvorfor det reelt kun er retningsvektorerne for linierne, der skal findes.

Da vektorer ikke er stedbestemt, dannes retningsvektorerne, som var kameraet placeret i origo.

Følgende metoder er fælles for dem begge:

Konstruktør Definerer øverste venstre hjørne i billed planet, og nogle vektorer, der beskriver, hvor meget man skal gå til henholdsvis højre og ned, for hver stråle, man vil sende af sted.

Argumenter float, Options

getRaysForPixel Beregner hvilke stråler der skal udsendes, for et pixelnummer.

Argumenter int

Returnerer Ray array

Disse metoder ligner for de to forskellige kameraer hinanden en del, dog er RandomAntiAliasingCamera klassens metoder mere kompliserede end defaultCamera , hvorfor dens metoder er beskrevet nærmere.

11.6.4 RandomAntiAliasingCamera

Dette kamera sørger for at lave stråler og for stokastisk antialiasing.

Beskrivelse af vigtigste klasser:

konstruktør Det spændende ved konstruktøren er vektoren frustumTopLeft, som er stedvektoren for billedplanets øverste venstre hjørne i forhold til kameraets udgangspunkt. Implementeringen, hvor den får en egentlig værdi, ser således ud:

```
frustumTopLeft = matrix.transform( new Vector( 1.0f, halfWidth, halfHeight )).
    sub(frustumOrigin);
```

Vektoren, som initialiseres med new Vector(1.0f, halfWidth, halfHeight), svarer til den stedvektor, som øverste venstre hjørne af billedplanet vil have såfremt kameraets matrice ikke er roteret (svarende til at matricens basis er det samme som koordinatsystemets).

Denne vektor bliver så ganget med kameraets matrice, og fratrukket med matricens stedvektor, således at vektoren får samme orientation, som matricen, dog uden at blive flyttet.

Vektorerne frustumDown og frustumRight, er retningsvektorer for henholdsvis højden og bredden af billedplaner.

getRaysForPixel Denne metode bestemmer de stråler, der skal bruges for en pixel.

Det vigtige er bestemmelse af retningsvektorerne for strålerne, hvilket foregår i følgende løkke, som giver samtlige stråler for én pixel:

```
for (int i = 0; i < plotPrPixel; i++) {
    int gridX = i % plotPrPixelSquared;
    int gridY = i / plotPrPixelSquared;

    for (int j = 0; j < raysPrPlot; j++)
    {
        randomY = (float)Math.random();
        randomX = (float)Math.random();
        float rightscale = ((float)x+(float)gridX+randomX)*rigthPrPixel;
        float downscale = ((float)y+(float)gridY+randomY)*downPrPixel;

        rayDir = rayDir.add( frustumTopLeft );
        rayDir = rayDir.add( frustumDown.scale( downscale ));
        rayDir = rayDir.add( frustumRight.scale( rightscale ));
        rayDir = rayDir.normalized();

        arrRays[i*raysPrPlot+j] = new Ray( frustumOrigin, rayDir, 1.0f, 1.0f);
    }
}
```

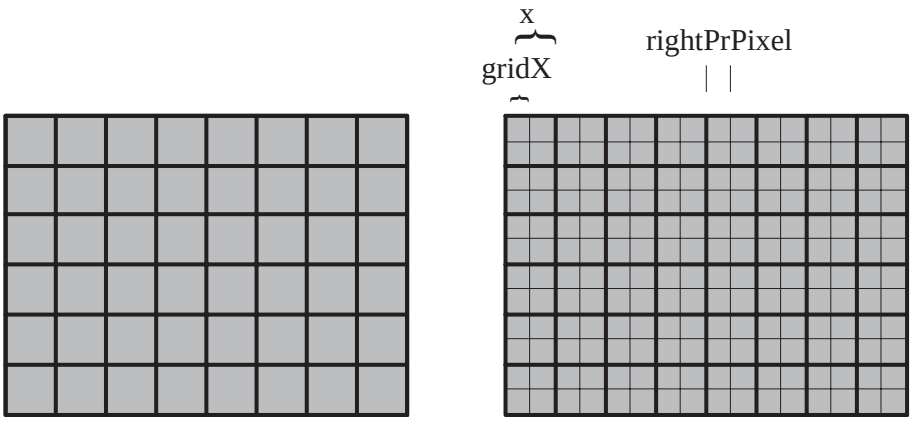
Den første løkke gennemgår alle de gitter-punkter, som en pixel er opdelt i (stokastisk antialiasing se figur 9.2 side 47). Den finder koordinatsættet for det aktuelle gitter-punkt indenfor den nuværende pixel, hvilket er variablerne gridX og gridY.

Den næste løkke gennemgår alle de tilfældige stråler indenfor et gitterpunkt. Variablerne rightPrPixel og downPrPixel er den længde, som et gitterpunkt i en pixel har i henholdsvis bredden og højden.

Variablerne x og y er koordinatsættet for pixelen i billedplanet, ganget med antallet af gitterpunkter i bredden af en pixel(=antal gitterpunkter i højden). Dvs. at værdien (float)x+(float)gridX er bredde-koordinatet på det aktuelle gitterpunkt, hvor man medregner alle gitterpunkter i alle pixels. Bemærk at definitionsmængden på gridX går fra 0 op til x's minimum, da dette er en mindre end

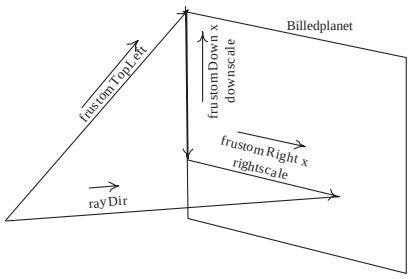
At tilføje variablen randomX betyder, at man får et tal mellem breddekoordinatet på det aktuelle gitterpunkt og det næste gitterpunkt i bredden.

Tallet ((float)x+(float)gridX+randomX) ganget med rightPrPixel, giver længden fra venstre side af billedplanet hen til værdien på førnævnte breddekoordinat.



Figur 11.3: Billedplan med og uden pixel-gitter. Hver pixel er her opdelt i et 2×2 gitter, hvorfor x altid vil være $2n$, hvor $n \in \mathbb{N}_0$. rightPrPixel er en længde

Vektoren rayDir sættes først til stedvektoren for billedplanets øverste venstre hjørne. Derefter skaleres enheds- og retningsvektoren for højden af billedplanet, frustomDown med længden fra toppen af billedplanet til højdekoordinatet, og lægges til rayDir. På tilsvarende måde som frustomDown skaleres, skaleres frustomRight og lægges til rayDir.



Figur 11.4: Grafisk illustration af beregningen af rayDir

Til sidst genereres strålen, og lægges i et array.

Når dette array er færdigt returneres det.

11.6.5 Object

Alle object klasser arver fra AbstractObject klassen, hvorfor denne først behandles. CSG figurer behandles i et senere afsnit

11.6.6 Om AbstractObject klassen

De vigtiste metoder i AbstractObject er:

Konstuktøren Konstruktøren i AbstractObject initialiserer en standard matrix og en standard shader.

Argumenter Scene

getNearestIntersection Finder det nærmeste skæringspunkt med en stråle. Dette gøres ved at bruge metoden getIntersections i den figur, som arver fra den og får de skæringerne der måtte være, hvorefter den finder den, som er tættest på strålens udgangspunkt.

Argumenter Ray

Returnerer Intersection

11.6.7 Metoder i object

Alle figurer, fungerer udefra set ens, dette gælder også CSG figurer.

Alle figurer har tre metoder:

getIntersections Finder to skæringspunkter med figuren, såfremt figuren er 2-dimensionel, som f.eks. et plan, vil den ene værdi være uendelig, hvis der ikke er nogen skæringspunkter returneres null

Argumenter Ray

Returnerer Intersection array

getSurfaceNormal Returnerer normalvektoren for et skæringspunkt.

Argumenter Vector

Returnerer Vector

getUV Returnerer UV koordinat for skæringspunkt.

Argumenter Vector

Returnerer float array

Derudover har hvert figur en konstruktør, som tager forskellige inputs alt efter hvilken type figur.

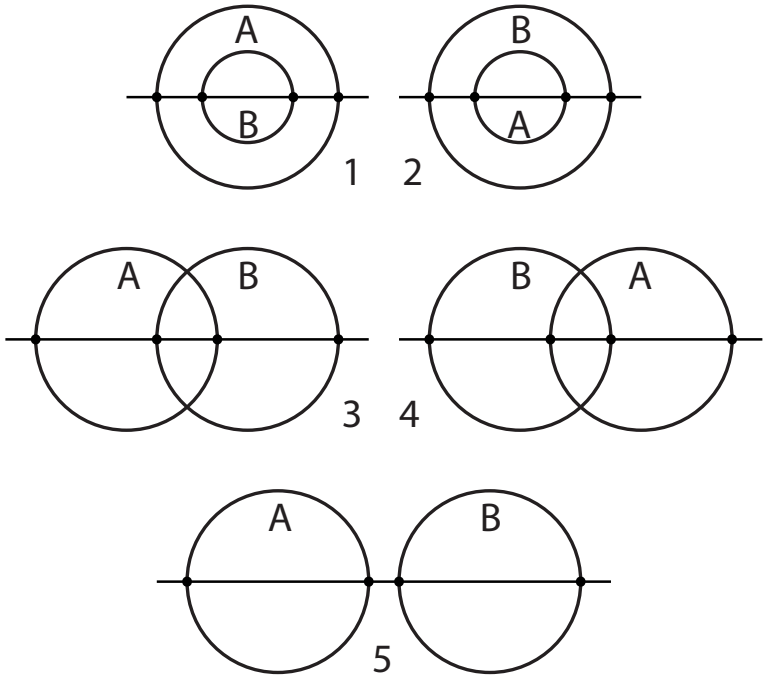
Som eksempel tager konstruktøren i en kugle, som input en radius og en scene. Scenen bliver brugt til at køre overklassens konstruktør med.

11.6.8 CSG

CSG figurer dækker over tre klasser:

- CSGUnion
- CSGIntersection
- CSGDifference

Disse tre klasser arver alle fra AbstractCSGObject, som igen arver fra AbstractObject.



Figur 11.5: De 5 forskellige kombinationer af CSG-figurers skæringer

CSG-figuren operere med to underfigurer, A og B. Disse figurers fællesmængde, foreningsmængde eller differens, bestemmer hvor CSG-figurens overflade er defineret. I praksis opnås dette ved at operere på to arrays indeholdende ind og udgangs-skæringer med underfigurerne. Under gennemløb af disse arrays, opbygges et nyt array med skæringssæt (et skæringssæt er én ind- og én udgangsskæring)repræsentere den nye figur. Da der altid kikkes på et skæringspar, dvs et skæringssæt fra A og et fra B, er der et endeligt antal af kombinationer (figur 11.5). Som en hjælp til at holde styr på disse indføres der i klassen to konstanter ENTRY EXIT der henholdsvis har værdien 0 og 1. For skæringsarrayet right [] vil ind og udgangs par nr i kunne findes ved

```
Intersection ind = right[i + ENTRY];
Intersection ud  = right[i + EXIT];
```

Det skal bemærkes at A og B internt i raytraceren hedder left og right. Når der tales om "nærmeste"og "fjerneste"skæring, skal det ses i forhold til origo for linjen, der skærer figuren.

11.6.9 Union

Ved union er CSG-figuren defineret i A, og B's foreningsmængde. Vi er derfor interesseret i at lade figuren med første indgangs-skæring, og figuren med sidste udgangs-skæring repræsentere skæring med CSG-figurens ind- og udgangsskæring.

De fem kombinationer håndteres på følgende måde

1-4 Skæringsparret med den nærmeste indgang føjes til listen som indgang, skæringsparret med det fjerneste udgang føjes til listen som udgangsskæring.

5 Skæringsparret indeholdende den nærmeste indgang føjes til skæringslisten, efterfulgt af det andet par.

Algoritmen til dette er implementeret på følgende måde i raytraceren

```
while ( leftIndex < left.length && rightIndex < right.length ) {
    final boolean leftEntersFirst = (left[leftIndex + ENTRY].t < right[rightIndex
        + ENTRY].t);
    final boolean leftExitsFirst = (left[leftIndex + EXIT].t < right[rightIndex +
        EXIT].t);

    // Do the intersection pairs overlap?
    if ((left[leftIndex + ENTRY].t < right[rightIndex + EXIT].t) && (right[
        rightIndex + ENTRY].t < left[leftIndex + EXIT].t))
    {

        if ( leftEntersFirst )
        {
            list.add( left[leftIndex + ENTRY] );
        }
        else // right enters first
        {
            list.add( right[rightIndex + ENTRY] );
        }

        if ( leftExitsFirst )
        {
            list.add( right[rightIndex + EXIT] );
        }
        else
        {
            list.add( left[leftIndex + EXIT] );
        }

        leftIndex += 2;
        rightIndex += 2;
    }
    else // the intersections don't overlap
```

```
        {
            if ( leftEntersFirst )
33          {
                list.add( left[leftIndex + ENTRY] );
                list.add( left[leftIndex + EXIT] );
36
                leftIndex += 2;
            }
39          else
            {
                list.add( right[rightIndex + ENTRY] );
42              list.add( right[rightIndex + EXIT] );

                rightIndex += 2;
45          }
        }
    }
48
    while ( leftIndex < left.length )
    {
51      list.add( left[leftIndex + ENTRY] );
        list.add( left[leftIndex + EXIT] );

54      leftIndex += 2;
    }

57  while ( rightIndex < right.length )
    {
        list.add( right[rightIndex + ENTRY] );
60      list.add( right[rightIndex + EXIT] );

        rightIndex += 2;
63  }
```

Da der kikkes på foreningsmængden imellem to figurer er rækkefølgen ligegyldig. Således vil obj1 og obj2 grafisk beskrive den samme CSG-figur i det følgende:

```
AbstractObject obj1 = new CSGUnion( right, left, matrix, scene );
AbstractObject obj2 = new CSGUnion( left, right, matrix, scene );
```

Intersection

Ved intersection er CSG-figuren defineret for fællesmængde af A og B. Her er man interesseret i den fjerneste indgang samt første udgang. Dette gøres ved at gennemløbe skærings-parrene for hver underfigur parallelt, og undersøge om parrene overlapper hinanden. Ved en overlapning tilføjes sidste indgang og første udgang til CSG-figurens skærings-liste. Hvis to skæringspar ikke overlapper hinanden, inkrementeres arrayet indeholdende den nærmeste indgang.

De 5 kombinationer håndteres på følgende måde

1-4 Den fjerneste indgang samt nærmeste udgang føjes til skæringslisten.

5 Da de to figurer ikke har nogen fællesmængde, vil den samlede CSG-figur ikke være defineret.

Algoritmen til dette er implementeret på følgende måde i raytraceren :

```
while ( leftIndex < left.length && rightIndex < right.length )
{
    final boolean leftEntersFirst = ( left [ leftIndex + ENTRY].t < right[rightIndex
        + ENTRY].t);
    final boolean leftExitsFirst = ( left [ leftIndex + EXIT].t < right[rightIndex +
        EXIT].t);

    // Do the intersection pairs overlap?
    if (( left [ leftIndex + ENTRY].t < right[rightIndex + EXIT].t) && (right[
        rightIndex + ENTRY].t < left[leftIndex + EXIT].t))
    {
        // Does the ray enter the right object after it enters the left ? (get the
            last entry)
        if ( leftEntersFirst )
        {
            list.add( right [rightIndex + ENTRY] );
        }
        else
        {
            list.add( left [ leftIndex + ENTRY] );
        }

        // Does the ray exit the right object before it exits the left ? (get the
            first exit)
        if ( leftExitsFirst )
        {
            list.add( left [ leftIndex + EXIT] );
            leftIndex += 2;
        }
        else
        {
            list.add( right [rightIndex + EXIT] );
            rightIndex += 2;
        }
    }
    else // the intersections don't overlap
    {
        // Advance the object with the closest entry point
        if ( leftEntersFirst )
        {
            leftIndex += 2;
        }
        else
```

```
39          {
                rightIndex += 2;
            }
42      }
    }
```

Da der kikkes på fællesmængden imellem to figurer er rækkefølgen ligegyldig. Således vil obj1 og obj2 grafisk beskrive den samme CSG-figur i det følgende:

```
AbstractObject obj1 = new CSGIntersection( right, left, matrix, scene );
AbstractObject obj2 = new CSGIntersection( left, right, matrix, scene );
```

Difference

Ved difference er CSG-figuren defineret for alle punkter der findes i left, men ikke i right. Right beskærer left.

Man er her interesseret i at lade alle indgangs-skæringer med right optræde som udgangs-skæringer med left. Dette er en smule kompliceret og differens er da også den metode der kræver flest overvejelser i forbindelse med de 5 kombinationer. De håndteres på følgende måder:

1 Indgangen for A føjes til listen som CSG-figurens indgang, og indgangen for B føjes til listen som CSG-figurens udgang. Derefter føjes udgangen for B til som CSG-figurens indgang, og udgangen for A som csg-figurens udgang.

2 Da B dækker hele A springes skæringsparrene over og intet indsættes i skæringslisten.

3 Indgangen for A føjes til listen som CGS-figurens indgang, og indgangen for B føjes til som CSG-figurens udgang.

4 Udgangen for B føjes til listen som CSG-figurens indgang, og udgangen for A føjes til som CSG-figurens udgang.

5 Skæringsparret for A føjes til listen som CSG-figurens skæringspar.

Algoritmen til dette er implementeret på følgende måde i raytraceren :

```
while ( leftIndex < left.length && rightIndex < right.length ) {
    final boolean leftEntersFirst = (left[leftIndex + ENTRY].t < right[rightIndex
        + ENTRY].t);
3   final boolean leftExitsFirst = (left[leftIndex + EXIT].t < right[rightIndex +
        EXIT].t);

    // Do the intersection pairs overlap?
6   if ((left[leftIndex + ENTRY].t < right[rightIndex + EXIT].t) && (right[
        rightIndex + ENTRY].t < left[leftIndex + EXIT].t))
```

```
    {
        if ( leftEntersFirst )
 9      {
            list.add( left[leftIndex + ENTRY] );
            list.add( right[rightIndex + ENTRY] );
12
            if ( leftExitsFirst )
            {
15              leftIndex += 2;
            }
            else
18          {
                left[leftIndex + ENTRY] = right[rightIndex + EXIT];

21              rightIndex += 2;
            }
        }
24      else // right enters first
        {
            if ( leftExitsFirst )
27          {
                // The right span covers the entire left span, don't add anything,
                // Cut off the part of the right span up until the left span's exit
30              // and advance to the next left span
                right[rightIndex + ENTRY] = left[leftIndex + EXIT];

33              leftIndex += 2;
            }
            else
36          {
                // Delete the first part of the left span, which is covered by the
                    right span
                // In other words, push the left entry forward to right's exit.
39              left[leftIndex + ENTRY] = right[rightIndex + EXIT];

                // We've passed the right span, get the next one
42              rightIndex += 2;
            }
        }
45  }
    else // the intersections don't overlap
    {
48      if ( leftEntersFirst )
        {
            list.add( left[leftIndex + ENTRY] );
51          list.add( left[leftIndex + EXIT] );

            leftIndex += 2;
54      }
        else
        {
57          rightIndex += 2;
        }
```

```
        }
60  }

    while ( leftIndex < left.length ) {
63      list.add( left[leftIndex + ENTRY] );
        list.add( left[leftIndex + EXIT] );

66      leftIndex += 2;
    }
```

Da der kikkes på en differens imellem to figurer er rækkefølgen afgørende. Således vil obj1 og obj2 grafisk beskrive to forskellige CSG-figurer:

```
AbstractObject obj1 = new CSGDifference( right, left, matrix, scene );
AbstractObject obj2 = new CSGDifference( left, right, matrix, scene );
```

11.6.10 Shader

Sørger for at returnere en farve til et skæringspunkt.

En del metoder i denne klasse bruges til at sætte forskellige egenskaber. Disse er:

setColor Sætter farven på figuren.

Argumenter Color, float

Returnerer void

setTexture Sætter en tekstur på en figur.

Argumenter Image

Returnerer void

setBumpmap Sætter bumpmap på figuren.

Argumenter Image, float

Returnerer void

setSpecular Angiver hvor meget spekulær reflektion af lyskilder, figuren har.

Argumenter float, float

Returnerer void

setTranslucency Sætter hvor gennemsigtig en figur er, og dens brydningsindex.

Argumenter float, float

Returnerer void

setRelectivity Sætter hvor gennemsigtig figuren er.

Argumenter float

Returnerer void

setTextureScale Angiver hvor stor teksturen skal være på figuren, og derved også hvor mange gange.

Argumenter float, float

Returnerer void

setTextureOffset Angiver hvor på figuren en tekstur skal starte.

Argumenter float, float

Returnerer void

setBumpmapScale Samme som setTextureScale, bare for bumpmap.

setBumpmapOffset Samme som setTextureOffset, bare for bumpmap.

Beskrivelse af andre vigtige metoder:

getLighting Denne metode bestemmer bidraget fra diffus belysning og spekulær reflektion af lyskilder.

Først får den alle lyskilder ved metoden scene.getLights(). Disse lyskilder løber den via en løkke igennem.

For hver løkke undersøges hvorvidt nogen andre figurer, ligger mellem skæringspunktet og lyset. Dette sker nogenlunde, som i trace metoden i Render klassen.

Hvis ingen figurer ligger mellem lyskilden og skæringspunktet, beregnes bidraget fra diffus belysning og spekulær reflektion af lyskilden.

Argumenter Vector, Vector, Vector, Color

Returnerer Color

getSurfaceColor Denne metoder returnerer overflade farven i et skæringspunkt. Såfremt Shader objektet ikke har nogen tekstur, er figuren ensfarvet, og denne farve returneres.

Figuren spørges om dets UV koordinat i skæringspunktet ved hjælp af metoden getUV(intersection) i figuren. Ud fra denne beregnes ST koordinatet, og dernæst nummeret på den pixel i teksturen, som ST koordinatet svarer til. Farven på denne pixel returneres.

Argumenter Vector

Returnerer Color

getSurfaceNormal Denne metode returnerer normalvektoren for figuren i skæringspunktet. Såfremt Shader objektet ikke har noget bumpmap, vil metoden returnere figurens normalvektor, beregnet ved metoden getSurfaceNormal (intersection) i figuren.

Såfremt figuren har et bumpmap findes figurens UV koordinat ved metoden getUV(intersection), som findes i figuren. Dernæst beregnes den rotation normalvektoren for figuren uden bumpmap skal have. Dette sættes i en matrice, som normalvektoren fra figuren ganges med. Dette sker i linien:

```
Vector normal = matrix.transform( obj.getSurfaceNormal(
intersection ) );
```

Denne normalvektor returneres

Argumenter Vector

Returnerer Vector

getReflection Denne metode laver en ny stråle ud fra en reflektions vektor og skæringspunktet. Reflektions vektoren er retningen for den linie, som reflekterer ud fra figuren. Denne stråle følges ved hjælp af trace metoden i Render klassen, som returnerer en farve. Denne farve returneres.

Argumenter Vector, Vector, float

Returnerer Color

getTransmission Denne metode beregner retningsvektoren for den stråle, som går i gennem den. Dette gøres ud fra brydningsindexset. Ud fra denne retningsvektor og skæringspunktet laves en stråle, som følges via trace metoden i Render klassen. Farven, som trace metoden returnerer, returneres.

Argumenter Vector, Vector, Vector, float

Returnerer Color

getIntersectionColor Denne metode returnerer farven for et skæringspunkt, ved hjælp af de andre metoder i klassen.

Først findes skæringspunktet ved metoden getVector(t) i Ray klassen. Dernæst beregnes normalvektoren i skæringspunktet ved klassens egen metode getSurfaceNormal (intersection), som tager skæringspunktet som input.

Såfremt Shader objektet har nogen form for reflektion, beregnes reflektionsvektoren.

Dernæst beregnes de forskellige lysbidrag. Først figurens farve i skæringspunktet ved klassens egen metode, getSurfaceColor(intersection). Denne farve benyttes til at beregne bidraget fra diffus belysning og spekulær reflektion af lyskilder, hvilket gøres med klassens egen metode, getLightning(intersection , reflection , normal, surfaceColor), som tager følgende input: Skæringspunkt, den beregnede reflektions vektor, den beregnede normal vektor og figurens farve i skæringspunktet.

Dernæst beregnes bidrag fra spekulær reflektion af figurer med metoden getReflektion(intersection , reflection , ray. refraction), som findes i klassen. Input er: skæringspunkt, reflektions vektoren og brydningsindex for strålen.

Til sidst beregnes bidrag fra transmission med metoden getTransmission (intersection , ray. direction , normal, ray. refraction). Denne metode findes i klassen og tager følgende input: Skæringspunkt, retningsvektoren for den indkomne stråle, normalvektoren, brydningsindexen for strålen og figurens farve i skæringspunktet.

De forskellige lys bidrag skaleres ned i forhold til det, de er defineret (hvis der er 20% spejling vil dette bidrag blive skaleret med 0.2). Derefter lægges bidragene sammen, og summen returneres.

Argumenter Ray, float

Returnerer Color

11.6.11 PhotonMap klassen

PhotonMap klassen repræsenterer samtlige fotoner i en given scene, der har ramt en diffus overflade. Da der ofte kan være over 100000 fotoner i en scene er det vigtigt at repræsentation fylder så lidt som muligt. Desuden skal der foretages søgninger for at finde n mindste fotoner for hver skæring mellem en figur og en ray. Derfor skal søgning i datastrukturen være hurtig.

Derfor er det valgt at implementere Photon Map som et kd-træ [Jen01]. Plads kravet for et kd-træ er $O(n)$ hvor n er antallet af fotoner og tiden for at finde k nærmeste fotoner til et givet punkt er $O(k+\log(n))$ i gennemsnit for et balanceret træ. I værste fald er den $O(n)$, hvis træet ikke er balanceret.

PhotonMap klassen implementerer et 3 dimensionalt kd-træ hvori alle fotoner, der interagerer med en diffus overflade lagres. Et kd-træ kan også have andre dimensioner end 3, fx er et 1 dimensionalt kd-træ det samme som et binært søgetræ.

11.6.12 PhotonTracer klassen

PhotonTracer klassen minder tildels om Render klassen, der er groft sagt tale om en forlæns raytracing. Klassen indeholder metoden trace der følger fotoner rundt i scenen. Overflade interaktioner beregnes i henhold til en vilkårlig BRDF.

PhotonTracer klassen modtager antallet af fotoner, der skal skydes ud fra alle lyskilder tilsammen. Antallet af photoner i Photon Mappet kan altså meget vel blive højere end det tal, der gives til metoden. Fotoner skaleres til slut med antallet af fotoner for hver lyskilde.

11.6.13 Data repræsentation

Photoner indeholder information om lyskildens intensitet, position for skæring med figur, retning for skæring med figur samt i hvilket plan kd-træet er splittet.

11.6.14 Balancering

Da der under indsætning af fotoner i træet ikke er behov for at finde noget i træet er det valgt at balanceringen foregår efter alle fotoner er behandlet.

Balancering af en kd-træ gøres ved at der for hver knude i træet vælges et plan om hvilket der skal splittes, dvs alle fotoner i venstre undertræ ligger under planets median og alle fotoner i højre undertræ ligger over planets median. Planet om hvor der skal deles vælges udfra i hvilken retning rummet har størst udbredelse.

Den rekursive balancerings kode ser i pseudo kode således ud:

Listing 11.2: Balancering af KD-Træ

```
knude balance(kd−træ)
    find bounding box
    find dimension hvor afstand er størst
    find medianen i dimension
    a = punkter under medianen
    b = punkter over medianen
    venstre knude = balance(a)
    højre knude = balance(b)

    returner knude;
```

Se evt http://www.rolemaker.dk/nonRoleMaker/uni/algogem/kdtree.htm for en grafisk præsentation af balancering af en 2 dimensionalt kd-træ.

11.6.15 Radiance estimatet

Da raytracing delen benytter sig af PhotonMappet til at beregne den udgående radiance er det også nødvendigt med en metode til at finde k nærmeste photoner.

Algoritmen til at finde k nærmeste fotoner er i pseudokode:

Listing 11.3: Radiance estimate beregning i KD-Træ

```
find_K_nærmestePhotoner(knude)
    hvis afstand til splitplane for knude < 0
        find_K_nærmestePhotoner(venstre undertræ)
        hvis afstand til splitplane for knude < max_afstand
            find_K_nærmestePhotoner(højre undertræ)
```

```
    ellers
        find_K_nærmestePhotoner(højre undertræ)
        hvis afstand til splitplane for knude < max_afstand
            find_K_nærmestePhotoner(venstre undertræ)

    hvis afstand til splitplane for knude < max_afstand
        indsæt photon i resultatsæt array
        juster max_afstand fra knude til roden i resultatsæt array
```

Summen af de k nærmeste skaleres derefter med tætheden af fotoner i området.

11.7 Gennemgang af kørsel

Her følger en kort gennemgang af de vigtigste metodekald ved programudførsel. Navnet i overskrifterne er navne på klasser.

11.7.1 RaytracerGUI

Her startes den grafiske brugerflade, som er selve starten af programmet.

11.7.2 Raytracer

I denne klasse startes selve raytracingen.

Når et raytracer objekt initialiseres skal den bl.a. have en scene, som input.

Metoden render kører en løkke for hver pixel i det færdige billede. I hver løkke kalder den metoden getRaysForPixel med nummeret for den aktuelle pixel i klassen, som tilhører kameraet. Metoden, som forefindes i kameraet, returnerer de stråler (stedvektor og retningsvektor for en linie) der er relevante for den aktuelle pixel. Inde i løkken er der en ny løkke, hvor hver stråle gives som input til trace metoden, som findes i Render klassen. Denne returnerer farven som strålen resulterer i.

11.7.3 Render

I Render klassen køres den statiske metode trace med en stråle og scenen, som input. Den gennemløber alle figurer i scenen, og finder den figur hvis skæringspunkt ligger tættest på kameraet eller ingen, hvis ingen figurer ligger på strålen. Skæringspunkt findes ved metoden getNearestIntersection, som forefindes i alle figurer.

Hvis der er skæring med figur(er), går trace metoden ind til figurens tilknyttede shader objekt og spørger denne hvilken farve der er i skæringspunktet, hvilket gøres i shader'ens getIntersectionColor metode.

Denne farve returneres.

11.7.4 Shader

Metoden getIntersectionColor finder normalvektoren i skæringspunktet ved metoden getSurfaceNormal, som findes i alle object klasser. Derefter beregnes hvilket lys der sendes mod kameraet i skæringspunktet. Denne farve returneres.

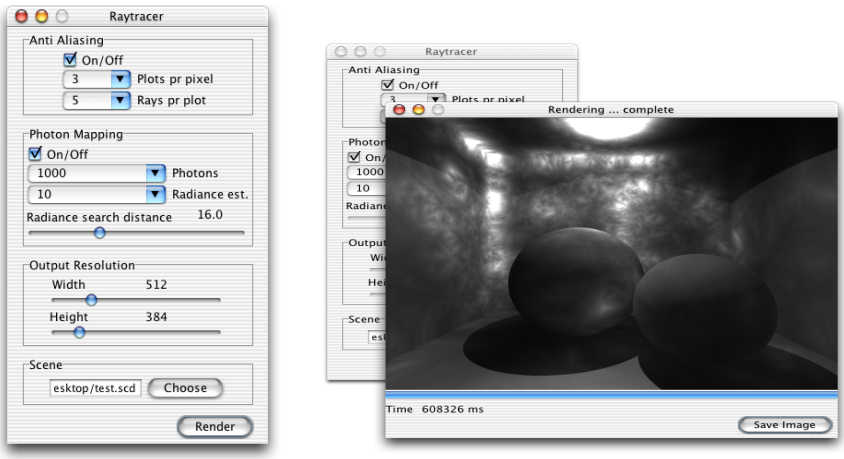
Afsnit 12

Brugervejledning

12.1 Grafiske brugerflade

For at køre programmet skal man have en scenefil. Opbygningen af en sådan er beskrevet i kapitel 12.3

Denne brugerflade er vist i figur 12.1.



Figur 12.1: Brugerfladen som programmet starter med

For at vælge scene kan man enten vælge at indtaste destinationen for en scene fil i det hvide felt, eller at søge manuelt efter den ved at trykke på knappen "Choose".

Herefter vil billedet blive optegnet. Når billedet er færdigt, kan man gemme billedet som en "png" fil, ved at trykke på knappen "Save Image".

12.1.1 Valgmuligheder

- Anti Aliasing

  On/Off Vælger / fravælger antialiasing

  Plots pr pixel Bredde af gitter

  Rays pr plot Antal stråler per gitterpunkt

- Photon Mapping

  On/Off Vælger / fravælger photonmapping

  Photons Antallet af fotoner der sendes

  Radiance est. Angiver hvor mange fotoner der indgår i estimated

  Radiance search distance Maximal afstand der søges efter fotoner i

- Output Resoulution

  Width Bredden af outputbilledet

  Height Højden af output billedet

- Scene Her vælges en scenefil

12.2 Konstruktion af scenefil

For at indlæse en scene i raytraceren skal der opbygges en scenefil. Valg af filformat er som før nævnt faldet på er XML. XML er opbygget som et hierarki, bestående af en række elementer med 0 eller flere atributter. Hvis et element indeholder andre elementer kaldes det "*forælder*" til underelementerne der kaldes "*børn*'. Et element og dets argumenter indkapsles i "<" og ">". Hvis elementet indeholder andre elementer startes elementet med <*elementnavn attributter*> og afsluttes med </*elementnavn*>. Hvis elementet ikke indeholder andre elementer noteres det <*elementnavn attributter* />. Et eksempel på et element kunne være

Listing 12.1: To XML elementer

```
<sphere radius="2">
    <rotation x="0.5" y="0.3" z="0.1" />
</sphere>
```

Hvor sphere er forælder til sit barn rotation. Sphere har desuden attributten radius med værdien 2. Det er tilladt at have et vilkårligt antal hvidtegn (tabulator, mellemrum, linjeskift) imellem element-indkapslingen. Dette kan med fordel bruges til at gavne overskuligheden af et element.

Listing 12.2: Brugen af hvidtegn gavner overskuelighed

```
<surface     specularamount="0.7"
             specularexp="2"
              refraction ="0.2"
             refractionamount="0"
              reflection ="0">
    <color r="0" g="1" b="1" />
</surface>
```

En XML-fil startes med et element der angiver hvilken version af XML der bruges i filen. Normalt vil dette element bliver efterfulgt af et såkaldt DOCTYPE element. I den nuværende version benyttes dette dog ikke i sceneformatet.

Listing 12.3: Det specielle XML-element

```
<?xml version="1.0"?>
```

12.3 Sceneformatet

Syntaxen for sceneformatet er som følger.

12.3.1 Syntax for sceneformatet

```
<?xml version="1.0"?>
<scene>
    <plane|sphere|intersection | difference |union attributter>
        <surface
            specularamount="genskin"
            specularexp="genskin-skarphed"
            refractionamount="brydning"
             refraction ="brydningsindex"
                                                    reflection ="spejling" >

            <color r="rød" g="grøn" b="blå" />
            <texture      file ="filnavn"
                         scalex="x-skalering"
                         scaley="y-skalering"
                          offsetx ="x-forskydning"
                          offsety ="y-forskydning" />
            <bumpmap file="filnavn" height="height" />
        </surface>
        <rotate x="x-rotation" y="y-rotation" z="z-rotation" />
        <position x="x-koordinat" y="y-koordinat" z="z-koordinat" />
    </figur>
    .
    . (0 eller flere figurere)
    .
```

```
<light int="Styrke">
    <color r="rød" g="grøn" b="blå" />
    <position x="x-koordinat" y="y-koordinat" z="z-koordinat" />
</light>
.
. (1 eller flere lys)
.
<background refraction="globalt-brydningsindex">
    <color r="rød" g="grøn" b="blå" />
</background>
.
. (0 eller 1 baggrundsfarve)
.
<camera fov="synsvinkel">
    <rotate x="x-rotation" y="y-rotation" z="z-rotation" />
    <position x="x-koordinat" y="y-koordinat" z="z-koordinat" />
</camera>
</scene>
```

12.3.2 scene

Hovedelementet der indeholder hele scenen.

Påkrævende børn

- præcis ét camera

Eksempel

```
<scene>
    <camera fov="45">
        <position x="0" y="0" z="0.0"/>
    </camera>
</scene>
```

12.3.3 light

Indsætter et punktlys i scenen

Attributter

int Intensiteten af lyset

Barn af

- Scene

Mulige børn

- position
- color

Eksempel

```
<light int="1337">
    <position x="0" y="-1" z="0" />
</light>
```

12.3.4 sphere

Indsætter en kugle i scenen.

Attributter

radius Radius af kuglen

name navn, bruges i forbindelse med CSG

Barn af

- Scene

Mulige børn

- rotate
- surface

Eksempel

```
<sphere radius="2" name="Left">
    <position x="0" y="-1" z="0" />
</sphere>
```

12.3.5 plane

Indsætter et uendeligt plan i scenen.

Attributter ingen

Barn af

- scene

Mulige børn

- rotate
- surface

name navn, bruges i forbindelse med CSG

Eksempel

```
<plane name="floor">
    <position x="0" y="0" z="-1" />
</plane>
```

12.3.6 difference

Beskriver en figur der er differensen mellem to andre figurer. Højre underfigur bliver trukket fra venstre underfigur. For at kombinere to figurer kræves at de er navngivet med *name* attributten.

Attributter

left Venstre underfigur

right Højre underfigur

name navn, bruges i forbindelse med CSG

Barn af

- scene

Mulige børn

- rotate
- surface

Eksempel

```
<sphere radius="1" name="A">
    <position x="0" y="0.25" z="0" />
</sphere>

<sphere radius="1" name="B">
    <position x="0" y="-0.25" z="0" />
</sphere>

<difference left="A" right="B">
    <position x="3" y="-0.5" z="0" />
    <rotate x="0.2" y="0" z="0.5" />
</difference>
```

12.3.7 intersection

Beskriver en figur der er fællesmængden mellem to andre figurer. For at kombinere to figurer kræves at de er navngivet med *name* attributten.

Attributter

left Venstre underfigur

right Højre underfigur

name navn, bruges i forbindelse med CSG

Barn af

- Scene

Mulige børn

- rotate
- surface

Eksempel

```
<sphere radius="1" name="A">
    <position x="0" y="-0.25" z="0" />
    <rotate x="0" y="0" z="0" />
</sphere>

<sphere radius="1" name="B">
    <position x="0" y="0.25" z="0" />
    <rotate x="0" y="0" z="0" />
</sphere>

<intersection  left ="A" right="B">
    <position x="3" y="0" z="0" />
    <rotate x="0" y="0" z="0" />
</intersection>
```

12.3.8 union

Beskriver en figur der er foreningsmængden mellem to andre figurer. For at kombinere to figurer kræves at de er navngivet med *name* attributten.

Attributter

left Venstre underfigur

right Højre underfigur

name navn, bruges i forbindelse med CSG

Barn af

- scene

Mulige børn

- rotate
- surface

Eksempel

```
<sphere radius="1" name="A">
    <position x="0" y="-0.25" z="0" />
    </sphere>

<sphere radius="1" name="B">
    <position x="0" y="0.25" z="0" />
</sphere>

<union left="A" right="B">
    <position x="3.5" y="0" z="0" />
</union>
```

12.3.9 position

Beskriver et elements placering i scenen.

Attributter

x x-koordinatet for positionen

y y-koordinatet for positionen

z z-koordinatet for positionen

Barn af

- sphere
- plane
- intersection
- difference
- union
- camera

Eksempel

```
<sphere radius="1">
    <position x="0" y="-0.25" z="0" />
</sphere>
```

12.3.10 surface

Beskriver et elements overflade

Attributter

specularamount Spkeularitet(genskin) af en overflade

specularexp Tja

refraction Tjo

refractionamount så deeet

reflection Graden af spejling

Barn af

- rotate
- surface

Mulige børn

- color

Eksempel

```
<sphere radius="2">
    <surface    specularamount="0.5"
                specularexp="10"
                refraction ="1.2"
                refractionamount="0"
                reflection ="0">
        <color r="1" g="0.5" b="0" />
    </surface>
</sphere>
```

12.3.11 color

Angiver et elements farve

Attributter

r Rød

g Grøn

b Blå

Barn af

- surface

Eksempel

```
<sphere radius="2">
    <surface ...>
        <color r="1" g="0.5" b="0" />
    </surface>
</sphere>
```

12.3.12 camera

Attributter

fov Værdi imellem 1-179, angiver Field Of Vision (synsvinkel)

Barn af

- scene

Mulige børn

- rotate
- surface

Eksempel

```
<sphere radius="2">
    <surface ...>
        <color r="1" g="0.5" b="0" />
    </surface>
</sphere>
```

12.3.13 rotation

Beskriver et elements rotation i forhold til scenen.

Attributter

x Værdi imellem 0-1, angiver rotation i x-aksen

y Værdi imellem 0-1, angiver rotation i y-aksen

z Værdi imellem 0-1, angiver rotation i z-aksen

Barn af

- sphere
- plane
- intersection

- difference
- union
- camera

Eksempel

```
<camera fov="45">
    <rotate x="3" y="0" z="0" />
    <position x="-3" y="0" z="0.0"/>
</camera>
```

12.3.14 position

Beskriver et elements placering i scenen.

Attributter

x X-koordinat

y Y-koordinat

z Z-koordinat

Barn af

- sphere
- plane
- intersection
- difference
- union
- camera

Eksempel

```
<sphere radius="1">
    <position x="-3" y="0" z="0.0"/>
</sphere>
```

12.3.15 bumpmap

Angiver bumpmap til en overflade.

Attributter

file Filen indeholdende bumpmappet. Formatet af filen er frit så længe det er undestøttet af Java

height Justere højden på bumpmappet. Angiver højden af hvid.

scalex Skalere bumpmappet langs x-aksen.

scaley Skalere bumpmappet langs y-aksen.

offsetx Forskyder bumpmappet langs x-aksen.

offsety Forskyder bumpmappet langs y-aksen.

Barn af

- surface

Eksempel

```
<bumpmap file="bump.gif" height="0.5" />
```

12.3.16 texture

Angiver teksturen til en overflade.

Attributter

file Filen indeholdende teksturen. Formatet af filen er frit så længe det er undestøttet af Java

scalex Skalere teksturen langs x-aksen.

scaley Skalere teksturen langs y-aksen.

offsetx Forskyder teksturen langs x-aksen.

offsety Forskyder teksturen langs y-aksen.

Barn af

- surface

Eksempel

```
<texture    file ="pattern.gif"
            height="0.5"
            scalex="0.25"
            scaley="0.5"
            offsetx ="5"
            offsety ="5"
/>
```

12.3.17 background

Beskriver den globale baggrund i scenen.

Attributter

refraction Angiver det globale brydningsindex

Barn af

- scene

Mulige børn

- color

Eksempel

```
<background refraction="1">
    <color r="0.5" g="0.5" b="0" />
</background>
```

Del III

Resultater

Afsnit 13

Afprøvning

Her beskrives og afprøves test scener. Der designes ækvivalensklasser ud fra kravspecifikationen, således at det kan vises om de i kravspecifikationen stillede krav er opfyldt.

13.1 Afprøvnings teori

Afprøvningen deles op i to dele: en intern og en ekstern.

Det skal selvfølgelig bemærkes at tests kun kan afsløre fejl, aldrig udelukke dem.

13.1.1 Ekstern test

I den eksterne test tester man programmet uden at benytte sig af kendskab til programmets interne opbygning. Alle tests foretages ud fra programmets kravspecifikation eller lignende. I en ekstern test prøver man at finde manglende eller fejlfyldte funktioner i programmet.

I eksterne test benytter man sig af ækvivalensklasser. En ækvivalensklasse er et scenario, der er repræsentativt for flere andre. I raytracerens tilfælde vil der blive designet en række testscener, der har til formål at teste den op imod kravspecifikationen. F.eks. kunne en testscene indeholde de 3 forskellige figurer påført en tekstur samt en kugle påført et bumpmap. Da selve opslaget i bumpmappet benytter sig af samme kald som opslaget i teksturen, er det ikke nødvendigt at teste for bumpmap på mere en een figur, så længe teksturer fungerer på de resterende.

Testscenerne vil blive opbygget således, at der så vidt muligt bliver testet så mange funktioner som muligt på een gang. Dette må dog ikke gå ud over kvali-

teten af de enkelte tests.

13.1.2 Intern test

Interne tests er tests med kendskab til programmets interne opbygning. Det er de enkelte dele af koden i selve programmet der testes, og ikke programmet, som helhed. I interne tests vil man prøve at tvinge programmet ud i grænsetilfælde, sørge for at alle linjer i koden bliver udført, alle logiske afgørelse bliver testet for både true og false, og alle løkker bliver gennemløbet maksimalt, samt minimalt antal gange.

Efter en succesfuld intern test skal programmet have gennemgået alle mulige tilstande.

Umiddelbart skulle man forvente, at et program der består ekstern test opfylder sin kravspecifikation, og derved er fuldendt. Man skal dog altid gå ud fra, at der altid vil kunne findes fejl i programmet. Et problem ved ekstern test er at de som regel er farvede af testdesigneren. Der gås målrettet efter at undersøge en række bestemte grænsetilfælde. Men fejl af mere tilfældig karakter, så som slåfejl eller misforståelse af algoritmer, vil ikke nødvendigvis blive fundet.

13.2 Valg af afprøvninger

Undervejs i udviklingen af programmet har vi lavet interne afprøvninger, men pga. tidsnød vælger vi, at vi ikke dokumenterer disse eller laver nye interne test.

Her følger en liste over de egenskaberne vi vil afprøve, samt hvilke spørgsmål afprøvningen skal besvare.

Transmission Kan man se figurer bag en gennemsigtig figur, og vil lyset blive afbøjet ?

Spekulær reflektion af figur Vil man kunne se en anden figur i en spejlende figur ?

Spekulær reflektion af lyskilde Vil man kunne se et lysende område på figurer, der hvor figuren spejler lyskilden ?

Diffus reflektion af lyskilde Vil man se en mat overflade ?

CSG Fællesmængde Vil man se fællesmængden af figurerne ?

CSG Foreningsmængde Vil man se en foreningsmængde af figurerne ?

CSG Differens Vil den ene figur være fratrukket den anden ?

Tekstur Kan man sætte et billede på en figur ?

Bumpmap Kan man lave overflade konturer på en figur ?

Photon Map Vil et spejl kunne reflektere lys fra lyskilden ned på en anden figur ?

Udover at afprøve disse egenskaber afprøver vi programmet med en scene, der minder om Cornellboks[Jen01], da denne giver en god afprøvning af global belysning. Med Cornellboks'en laver vi to afprøvninger, en normal, og en hvor lyset ikke aftages. Dette skyldes, at vi i forvejen ved, at lysaftagelsen ikke er korrekt ved spekulære reflektioner. Derved vil vi prøve hvorvidt der eksisterer andre problemer ved spekulære reflektioner.

Der er under udviklingen af programmet blevet foretaget eksterne tests for at sikre at de enkelte komponenter fungerede som ønske. I det følgende er de mest repræsentative af disse.

13.3 Test af raytraceren

Dette afsnit beskriver de afprøvningsscener, som er valgt til at afprøve forskellige dele af programmet. I alle scener hvor intet andet er nævnt, er kameraet placeret i origo, og har retning ud af den positive x akse.

Scenefilerne, som er brugt til afprøvning, ligger på den vedlagte CD.

13.3.1 Transmission

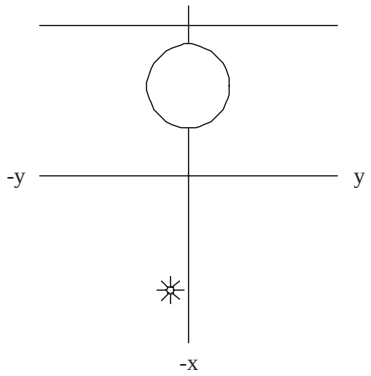
En delvis gennemsigtig kugle placeres foran et plan med en lyskilde bag ved kameraet. Scenen er illustreret ved figur 13.1.

Formålet med scenen er, at teste for transmission. Såfremt transmissionen er perfekt vil man kunne se planet gennem kuglen. Endvidere vil strålerne blive afbøjet, således at skyggen bag kuglen, vil virke lavere set igennem kuglen.

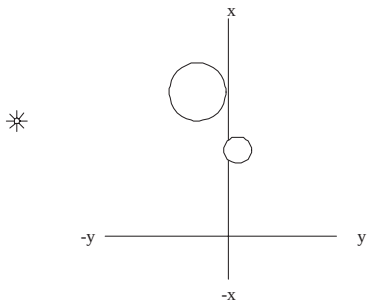
Ved test viser det sig at raytraceren ikke kan håndtere transmission, men i stedet går ned.

13.3.2 Spekulær reflektion af figur

Her benyttes to kugler, hvoraf den ene er reflekterende. Foran den reflekterende kugle ligger der lidt forskudt en kugle med diffus overflade. Situationen er vist i figur 13.2.



Figur 13.1: Forholdet mellem figurerne og origo er længere end figuren viser. Lyskilden er desuden placeret højere end kuglen således, at den lyser noget af planet bag kuglen op.



Figur 13.2: Lyskilden er i virkeligheden placeret længere ude af den negative y akse.

Lyset vil lyse ind på den lille kugle, som så igen vil lyse ind på den store reflekterende kugle. Stedet hvor man vil kunne se den lille kugle i den store ligger således, at det ikke vil modtage direkte lys.

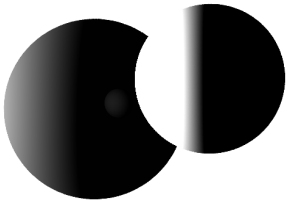
Såfremt reflektionen virker, vil man kunne se den lille kugle afbilledet i den store.

Resultatet ses i figur 13.3 Selvom der er meget lys på billedet, kan man ikke se den lille kugle blive spejlet i den store, som den burde.

13.3.3 Spekulær reflektion af lyskilde

Denne scene er kun opbygget af en kugle og en lyskilde. Lyskilden er placeret højere og lidt til venstre for kuglen, set fra kameraet. Kuglen reflekterer lyset.

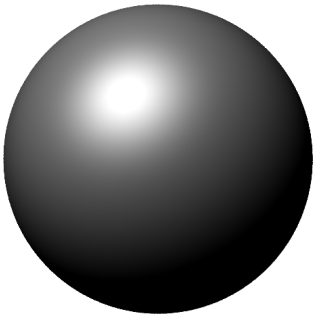
Scenen skal vise et meget lyst område omkring det sted hvor lyskilden vil blive spejlet i figuren. Det er lettest at vurdere scenen i forhold til scenen med en diffus



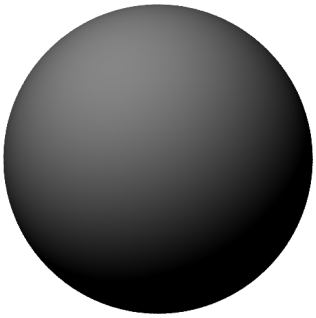
Figur 13.3: Resultat af test for spekulær reflektion af en figur

overflade, da denne har samme konfiguration bortset fra kuglens overflade.

Resultatet kan ses i figur 13.4.



Figur 13.4: Resultat af test for spekulær reflektion af lyskilde



Figur 13.5: Resultatet for diffus overflade

Der er en meget lys plet på kuglens overflade, som også svarer til lyskildens placering. Specielt i forhold til figur 13.5, som har en diffus overflade, kan man se at effekten er reel.

13.3.4 Diffus overflade

Scenen er opbygget som scenen med spekulær reflektion af lyskilde. Eneste forskel er, at overfladen på kuglen i denne scene er diffus. Scenen består af en kugle med diffus overflade og en lyskilde, som set fra kameraets synspunkt, er placeret højere og lidt til højre for kuglen.

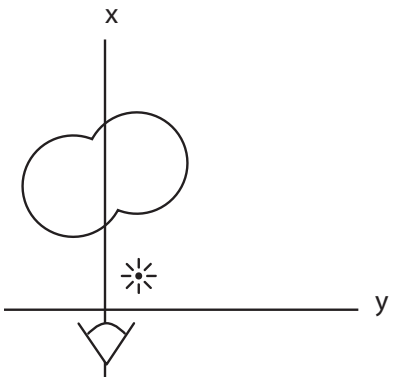
Scenen skal vise en mat overflade på kuglen. Overfladen skal være lysere jo mere

normalvektoren i skæringspunktet nærmer sig lyskildens indfaldsvinkel.

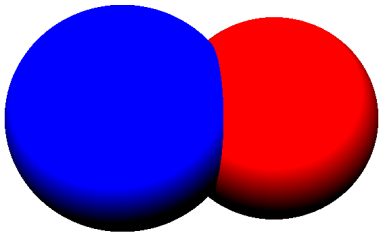
Resultatet kan ses i figur 13.5. Resultatet er som forventet.

13.3.5 CSG Foreningsmængde

To kugler, henholdsvis blå og rød, er opstillet således, at de skærer hinanden. Derefter er de kombineret i et CSG foreningsobjekt, som er roteret en anelse omkring z aksen. Set fra kameraet er den blå kugle længst til venstre.



Figur 13.6: Skitse af scene til test af CSG-union



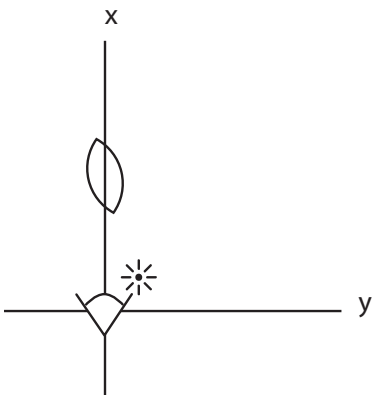
Figur 13.7: Resultatet for CSG foreningsmængde

Denne afprøvning skal vise begge kugler, som skærer hinanden. Resultatet kan ses i figur 13.7, Resultatet er som forventet.

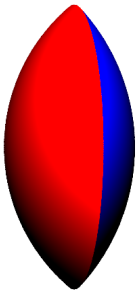
13.3.6 CSG Fællesmængde

Denne scene er bygget af to kugler, som skærer hinanden. Disse er kombineret i et CSG fællesmængde objekt, der er roteret en anelse omkring z aksen. Kuglerne er henholdsvis rød og blå, med den blå længst til venstre set fra kameraet.

Afprøvningen skal vise en figur, som består af det volumen, som de to kugler deler, se figur 13.9. Denne figur skal på venstre side i forhold til kameraet være rød, og på højre side være blå. Resultatet kan ses i figur 13.9. Denne figur fungerer fint. Bemærk sidernes forskellige farver.



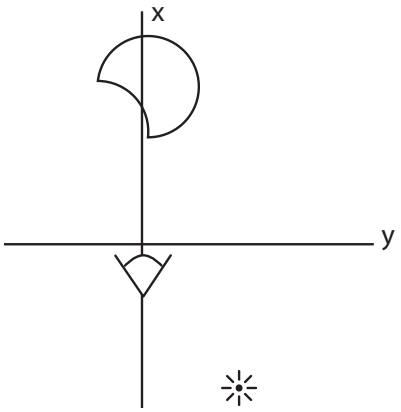
Figur 13.8: Skitse af scene til test af CSG-intersection



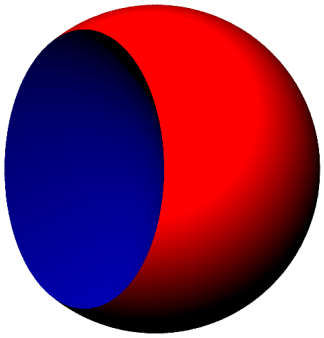
Figur 13.9: Resultat for CSG fællesmængde

13.3.7 CSG Differens

Som de andre CSG tests, er også denne opbygget af to kugler, der skærer hinanden. Kuglen til venstre for kameraet blå, den anden rød. På disse er udført CSG differens således, at den blå skærer den røde. CSG objektet er roteret omkring z aksen, således, at "hullet" i den røde kugle bliver mere synligt. Kameraet er flyttet lidt i y aksens retning, således at den røde kugle bliver placeret i centrum af billedet se figur 13.8.



Figur 13.10: Skitse af scene til test af CSG-difference



Figur 13.11: Resultat for CSG differens

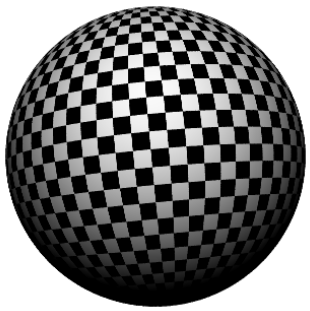
Hvis scenen vises korrekt, vil den røde kugle være beskåret med en kugle. "Hullet" i den røde kugle vil være blåt, da kuglen som beskærer er blå.Resultatet kan ses i figur 13.11 Resultatet er som forventet.

13.3.8 Tekstur

Scenen indeholder en kugle og en lyskilde. På kuglen er der lagt en tekstur med skiftevis sorte og hvide firkanter, som på et skakbræt.

Såfremt teksturen virker vil man kunne se den på kuglen.

Resultatet kan ses i figur 13.12. Resultatet er som forventet.



Figur 13.12: Resultat for tekstur

13.3.9 Bumpmap

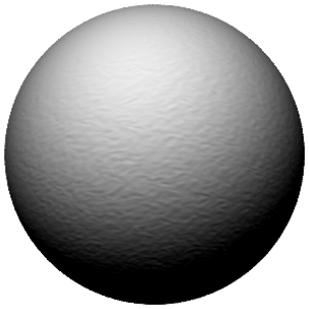
Her er opstillet en kugle, hvorpå der er lagt et bumpmap med vilkårlige højdekurver. Desuden er der i scenen en lyskilde.

Hvis bumpmappet virker vil det virke, som om kuglen er ujævn.

Resultatet kan ses i figur 13.13. Resultatet er som forventet.

13.3.10 Photon map

I denne scene er opstillet en lyskilde, en kugle og to planer. Kuglen er opstillet med lyskilden direkte over sig, og et plan nedenunder. Det sidste plan er, set fra



Figur 13.13: Resultat for bumpmap

kameraet, placeret lidt til højre. Dette plan er desuden fuldkommen spejlende. De to planer er vinkelret på hinanden. Opstillingen er vist på figur 13.14.

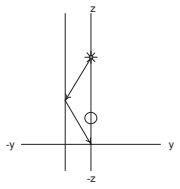
Såfremt photonmappet virker, vil skyggen under kuglen ikke være fuldstændig sort, men oplyst af spejlingen fra det spejlende plan. Resultatet kan ses i figur 13.15. Photon mappet virker. Skyggen under kuglen er ikke fuldkommen sort, hvilket er helt i overensstemmelse med hvad der burde ske.

13.3.11 Cornellboks

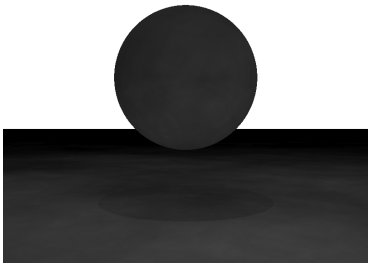
Denne scene minder om den såkaldte Cornellboks.

Scenen består af indersiden af en kube, og to kugler. Begge kugler er helt spejlblanke, således at alt, lys der rammer dem, reflekteres. Den ene kugle har desuden fået lagt et *bumpmap* på sin overflade, som resulterer i ujævn spejling. Hver væg har en unik solid farve.

Scenens hovedformål er at teste indirekte belysning. En lyskilde er placeret i en udhulning i loftet således at kun de nederste dele af væggene og en del af gulvet oplyses direkte fra lyskilden. Så længe der kun bruges direkte belysning vil loftet derfor være helt sort. Med en velfungerende implementering af global belysning, vil noget af det lys, der rammer gulvet, væggene og kuglerne blive reflekteret tilbage op i loftet, således at loftet oplyses og bliver synligt. En anden konsekvens af indirekte belysning er, at farver fra nærliggende figurer "bløder"over på



Figur 13.14: Test af photon map. Pilene illustrerer hvordan lyskilden lyser ind på det spejlende plan, og ned på det andet plan

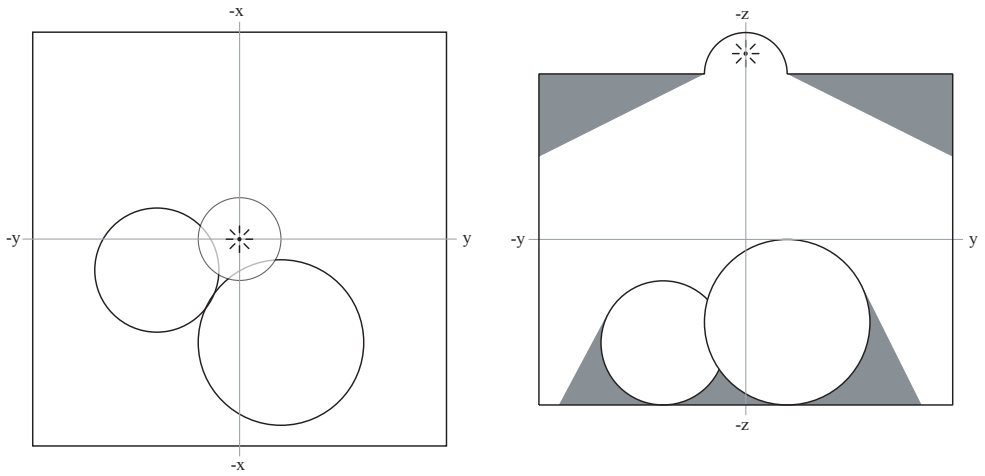


Figur 13.15: Restultat af test for photonmapping

hinanden. Det vil sige at de forskellige markante farver, som hver væg har, vil "smitte af"på nærliggende overflader.

Cornellboks med aftagelse af lys

Resultatet kan ses i figur 13.17. Dette billede er ikke som forventet. Alle skyggerne og loftet er fuldkomne sorte, hvilket betyder, at den globale belysning ikke virker. Desuden kan man ikke se, at kuglerne spejler, da lyset aftager for kraftigt. I kuglen til venstre er desuden nogle fejl, i form af små hvide prikker i et ellers sort område.



Figur 13.16: Conellboks

Figur 13.17: Resultat for cornellboks med aftagelse af lys

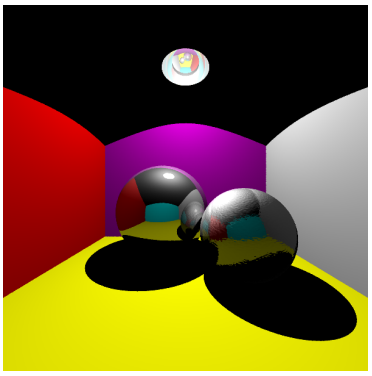
Cornellboks uden aftagelse af lys

Resultatet kan ses i figur 13.18. Her virker den globale belysning ikke, hvorimod spejlingen i de to kugler virker efter hensigten.

13.4 Vurdering af resultater

En generel fejl er at måden lyset aftager på, ikke er korrekt. Fejlen i programmet er, at spekulær reflektion af figurer bliver beregnet på samme måde som diffus reflektion. Dette bevirker, at figurers spejling bliver stærkt nedsat.

Hvorfor transmission udløser et programstop er uvist. For at rette fejlen bliver man nødt til at se på de fejlmeddelelser, der kommer, når programmet stop-



Figur 13.18: Resultat for cornellboks uden aftagelse af lys

per, og se hvor i implementeringen fejlen ligger. Nogle interne tests vil sandsynligvis afsløre problemets placering. Det samme gælder de hvide prikker under afprøvning med cornellboks med lysaftagelse.

Global belysning/photon map virker ikke, da implementeringen af denne ikke er gennemført.

Teksturering, bumpmap, spekulær reflektion af lyskilde, diffuse overflader og alle CSG operationer fungerer upåklageligt.

Afsnit 14

Diskussion

Dette afsnit vil behandle følgende aspekter af de designovervejelser der ligger til grund for udviklingen af raytracerprogrammet, nemlig ønsket om et særdeles abstrakt og modulært design såvel som ønsket om letlæselig kode der lægger vægt på læsbarheden frem for hastigheden. Yderliger påpeges hvorfor det ikke umiddelbart er til at sammenligne billeder genereret med Shader og SchlickShader samt hvad der mangler for at implementere Monte Carlo path tracing og Photon Mapping fuldt ud.

14.1 Letlæselig kode vs. hastighed

Et af kravene til koden i raytraceren var, at den skulle være nem at læse således at andre kunne lære af at læse den. I klasserne Vector og Color har dette betydet, at mange af metoderne er implementeret således, at de returnerer nye instanser af klasserne fremfor at modificere felterne i de objekter, metoderne kaldes på. Et eksempel på dette vises i listing 14.1:

Listing 14.1: Eksempel på summen af to farver

```
Color red = new Color(1,0,0);
Color green = new Color(0,1,0);

Color result = red.add( green );
```

Her er det vigtigt at bemærke, at det objekt, som referencen red peger på ikke ændres. add() metoden lægger de to farver sammen og returnerer et nyt Color objekt med farven gul (1,1,0).

Denne fremgangsmåde betyder, at det er nemmere at opbygge komplekse udsagn. farve1 + farve2 + farve3 kan altså skrives som:

```
Color resultat = farve1.add( farve2 ).add( farve3 );
```

Der er dog en meget vigtig konsekvens af dette. Metodekaldet farve1.add(farve2) laver et nyt objekt. Dette objekt adderes til farve3 og det resulterende objekt får resultat som reference. Der laves altså to midlertidige objekter som ikke bruges til andet end mellemregninger og ellers kan vente på at garbage collectoren kommer forbi. Grunden til at dette pointeres er, at der bliver lagt mange farver sammen i shaderen hver gang den skal finde en enkelt skæringsfarve. Selv et lille billede på 320 × 240 pixels uden antialiasing resulterer i mindst 76800 kald til getIntersectionColor(), som laver mange midlertidige Color objekter. Resultatet af dette er, at antallet af instantierede Color objekter nemt kommer op i millionerne. Foruden et unødigt højt hukommelsesforbrug betyder det også at Javas garbage collector bliver nødt til at aktivere og rydde op i de gamle midlertidige objekter adskillige gange mens et billede genereres. Det ville være ønskværdigt at undgå at aktivere garbage collectoren, da den sløver optegningen af billedet mærkbart.

Det samme gælder for Vector, Intersection og Ray klasserne. Vector returnerer nye instanser af klassen ved mange metodekald analogt til Color. Ray og Intersection klasserne gør ikke noget særligt, men eftersom der laves adskillige instanser af klasserne hver gang en skæringsfarve skal findes, bliver der også instantieret rigtig mange af disse under en almindelig kørsel. En mulig løsning på problemet er at implementere såkaldte object-pools, hvor en samling af objekter genbruges i stedet for at instantiere og bortkaste objekter løbende. Overfladiske forsøg viste at der er i størrelsesorden 50.000 Vector objekter instansieret på én gang. Det er stadig mange flere end der reelt er brug for, men det er også en helt anden størrelsesorden end millioner. Formålet med at implementere et object-pool for de tre udvalgte klasser ville være at spare på mængden af hukommelse der er allokeret på én gang, og derved undgå at aktivere garbage collectoren. Desværre er det ikke ligetil at implementere object-pools, da man manuelt skal håndtere "frigivelse"af objekterne. Det er kompliceret, når der bruges så mange midlertidige værdier i udregninger, uden at gøre koden sværere at gennemskue. Intersection og Ray klasserne er nok de eneste klasser, hvor en object-pool ville være ligetil at implementere.

14.2 Abstraction

Raytraceren er så vidt muligt designet således, at man ofte blot skal tilføje en ny klasse, der nedarver fra en given abstrakt baseklasse for at implementere ny funktionalitet.

Som eksempel nævnes *path tracing*, som blev implementeret i raytraceren udelukkende ved at udskifte shaderen til én, der var i stand til at vælge lystransport ud fra en sandsynlighedsfordeling. Ligeledes blev *antialiasing* implementeret ved at lave et nyt kamera, der simpelthen skyder flere stråler afsted pr. pixel. Både antialiasing og path tracing benytter sig af at kunne undersøge flere stråler pr. pixel.

Figurer og kameratyper er i sig selv helt selvindholdte, forstået på den måde, at så længe de implementerer de respektive abstrakte klasser, så er der ikke nogle steder i raytraceren, hvor der skal ændres for at understøtte en ny figur eller et nyt kamera. Dog er det p.t. nødvendigt at ændre i SceneLoader klassen for hver ny figur/kamera/shader der tilføjes hvis man vil kunne tilføje det ved hjælp af en scene-fil.

Ved implementering af nye figurer har man valgfriheden til at vælge at transformere selve figuren eller transformere strålerne, når der undersøges skæring. De tre figurer, der er implementeret p.t. transformerer selve figuren, men man kunne forestille sig en figur, hvor det var hurtigere at transformere strålen ind i figurens lokalkoordinatsystem fremfor at transformere figuren. En sådan figur kunne være en figur, der indeholder et antal trekanter der tilsammen danner en større figur. Hvis en sådan figur indeholder flere tusind trekanter er det unægtelig hurtigere at transformere strålen ind i figurens lokalkoordinatsystem og finde de trekanter, som den transformerede stråle måtte skære med.

14.2.1 Monte Carlo

To shadere er møntet på at understøtte Monte Carlo integration: SchlickShader og DistributedShader. Sidstnævnte er en udvidelse af den almindelige Shader der beregner en tilfældig lystransport efter en stokastisk fordeling.

Hverken SchlickShader eller DistributedShader er tilstrækkelig implementeret til at lave ordenlig Monte Carlo path tracing. SchlickShaderen understøtter ikke farvede overflader, og ingen af shaderne formår at følge diffuse reflekterede stråler, hvilket er et af hovedpointerne i Monte Carlo path tracing.

Hvis en shader blev udvidet til at følge diffuse stråler ved direkte belysning, ville raytraceren være i stand til at lave Monte Carlo path tracing.

14.2.2 Photon map

Vores photon map knytter sig til SchlickShaderen, som kun er delvist implementeret. En af manglerne ved SchlickShaderen er farvehåndtering. Det blev først klart for projektgruppen hvorledes en Schlick BRDF skal behandle farve lige op til afleveringen.

På nuværende tidspunkt tager SchlickShaderen slet ikke højde for farvede figurer - alle overflader antages at være hvide. I virkeligheden afhænger en figurs reflekterede farve af tre ting: figurens farve, lyskildens farve samt indfaldsvinklen. Indfaldsvinkelen er en af de faktorer, der bestemmer fresnel-faktoren for overfladen i det undersøgte punkt. Når fresnel-faktoren er ved sit minimum, altså ved perfekt diffus reflektion, filtreres lyskildens belysning med figurens overfladefarve.

Når fresnel-faktoren er ved sit maksimum, altså ved perfekt spekulær reflektion, filtreres lyskilden slet ikke af figurens overfladefarve. Graden af filtrering interpoleres imellem de to grænseværdier for fresnel-faktoren[Gla89], [Sch94a].

Det betyder selvfølgelig, at billeder genereret med SchlickShaderen har hvide figurer. Yderligere betyder det også, at lyset i photonmappet ikke påvirkes når det reflekteres rundt i scenen, da SchlickShaderen og photonmappet er tæt knyttet til hinanden. Det er dog vigtigt at bemærke, at selve photonmappet samt de tilhørende metoder fungerer efter hensigten. Det er udelukkende SchlickShaderen der ikke formår at påvirke lyset når det reflekteres fra forskellige figurer.

Bortset fra mangel på farvehåndtering mangler SchlickShaderen også mange af de overfladeegenskaber som den almindelige Shader implementerer, bl. a. teksturering og bumpmapping.

14.2.3 Fysisk korrekthed

Det har aldrig været et mål for for raytraceren at opfylde fysikkens love hvad angår lys og energibevarelse, altså at der ikke kan udsendes mere lys fra et punkt end der modtages fra omgivelserne.

Vores almindelige Shader (og DiffuseShader, der blot nedarver fra Shader) forsøger slet ikke at håndtere energibevarelse. Det gør SchlickShaderen derimod [Sch94a], sammen med photonmappet.

Det betyder, at det ikke er til at sammenligne billeder lavet med Shaderen med billeder lavet med SchlickShaderen og photon map. Billeder lavet med Shaderen vil være markant lysere end det lavet med SchlickShaderen.

Afsnit 15

Konklusion

Afsnit 16

Perspektivering

Da photon mapping efter endt projektforløb ikke virker tilfredsstillende er det naturligt nok første skridt i eventuelle udvidelser af raytraceren at færdiggøre denne del. I forbindelse hermed kan man evt. tage kontakt til opfinderen af photon mapping, Wann Jensen.

Andre udvidelser kunne være understøttelse af figurer bestående af vilkårligt mange trekanter. Da dette er en hyppig anvendt teknik i eksisterende 3d-programmer, vil det blive lettere at importere scener fra disse. Der er i den nuværende implementering ikke lagt synderlig vægt på hastighed. Optimering af raytraceren vil derfor også være oplagt. F.eks. kan søgningen efter skæringer lettes betydelig ved brug af bounding volumes. Photonmappet kunne forbedres vha. bedre sampling. For en øget fysisk realisme kunne der implementeres en fysisk korrekt BRDF. Endelig kunne man forestille sig værktøjer til animation og modellering af scener som en del af raytraceren.

Raytraceren vil i sin nuværende form kunne finde anvendelse for studerende og andre interesserede der ønsker mere information om hvordan en raytracer kan implementeres. Med velvalgte udvidelser kunne en raytracer med global belysning i java f.eks. finde anvendelse hos virksomheder der sælge udstyr til boligindretning over internettet. Man kunne hurtigt rendere billeder af en rimelig kvalitet af f.eks. et køkken på baggrund af kundens ønsker om indretning.

Afsnit 17

Referencer

[DIK01] DIKU. *Strålesporing i farver.* Opgaveformulering på DAT0 kursus på DIKU, 2001.
http://www.diku.dk/teaching/2001f/dat0/filer200001/k2.pdf

[Ele02] The Mind Electric. *Electric XML 4.1 User Guide.* 2002.
http://www.themindelectric.com/docs/xml/guide/index.html

[FFKN01] Anders Fleron, Morten Frank, Michael Kjems, and Ulf Holm Nielsen. *Java 3D Engine, En Java programpakke til tredimensionel visualisering.* RUC, 2001.
http://www.dyregod.dk/j3d/j3dprojekt.pdf

[FvDFH92] Foley, van Dam, Feiner, and Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, second edition, 1992.

[Gla89] Andrew S. Glassner. *An Introduction to Ray Tracing.* Academic Press, 1989.

[Hec90] Paul S. Heckbert. *Adaptive radiosity textures for bidirectional ray tracing.* Computer Graphics, SIGGRAPH '90, 1990.

[Jen01] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* A K Peters LTD., 2001.

[Lás00] Szirmay-Kalos László. *Monte Carlo Methods in Global Illumination.* Institute of Computer Graphics, Vienna University of Technology, 2000.

[Sch94a] Christophe Schlick. *An Inexpensive BDRF Model for Physically based Rendering.* Computer Graphics Forum, 1994.
http://dept-info.labri.u-bordeaux.fr/~schlick/publi.html

Afsnit 18

Supplerende Litteratur

[Dra02] Chris Dragan. *Raytracing shapes*. 2002.
http://www.cfxweb.net/article.php?sid=997

[Dut01] Philip Dutré. *Global Illumination Compendium*. Program of Computer Graphics, Cornell University, 2001.
http://www.cs.kuleuven.ac.be/~phil/GI/

[HB97] Donald Hearn and M. Pauline Baker. *Computer Graphics, C version*. Prentice-Hall Inc., second edition, 1997.

[Hut93] David J Hutchinson. *Monte Carlo Path Tracing for Photorealism*. Department of Computer Science, University of Manchester, 1993.

[Jan99] Torpum Jannak. Java 2 graphics rendering, an architecture for extreme animation. *Dr. Dobb's Journal*, September, 1999.

[MT97] Tomas Möller and Ben Trumbore. *Fast, Minimum Storage Ray-Triangle Intersection*. Journal of graphics tools, 2 edition, 1997.
http://www.acm.org/jgt/papers/MollerTrumbore97/

[Sch94b] Christophe Schlick. *A Survey of Shading and Reflectance Models for Computer Graphics*. Computer Graphics Forum, 1994.
http://dept-info.labri.u-bordeaux.fr/~schlick/publi.html

[Vea97] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Department of Computer Science, Stanford University, 1997.

[Wil94] Nicholas Wilt. *Object Oriented Ray Tracing in C++*. Wiley, 1994.

Indeks

Del IV

Appendiks

Appendiks A

Supplerende matematik

A.1 Skæring mellem en linie og et plan

[DIK01]

En linie har parameterfremstillingen:

$$l = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + t \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \end{pmatrix} = p + t\vec{v}$$

For et plan defineret ud fra et punkt i planet $p = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$ og en normalvektor $\vec{n} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ kan ligningen for et plan opstilles:

$$a(x - x_1) + b(y - y_1) + c(z - z_1) = 0 \Leftrightarrow ax + by + cz + d = 0$$

Ud fra et plan og en linie kan man så udregne skæring (t værdien i linien) :

$$t = -\frac{ax_0 + bx_0 + cz_0 + d}{a\Delta x + b\Delta y + c\Delta z}$$

Såfremt nævneren er 0 er linien parallel med planet.

A.2 Matricer

[FFKN01]

Dette afsnit beskæftiger sig med, hvordan matricer bliver brugt i programmet. Med matrix menes den form for matrix, der bliver brugt i programmet. Dette betyder at denne matrix på nogle punkter kan være matematisk ukorrekt.

Der benyttes 4×4 matricer i programmet. En matrix indeholder et intern koordinatsystem også kaldet matricens *basis*. Denne basis består af tre vektorer. Derudover indeholder matricen også en stedvektor. De tre vektorer er altid ortogonale, dvs. de står vinkelret på hinanden.

Matricen er således opbygget:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & x \\ m_{10} & m_{11} & m_{12} & y \\ m_{20} & m_{21} & m_{22} & z \\ m_{30} & m_{31} & m_{32} & m_{33} \end{pmatrix}$$

De tre basis vektorer er givet ved de tre første tal i de tre første kolonner, f.eks. $\begin{pmatrix} m_{00} \\ m_{10} \\ m_{20} \end{pmatrix}$. Stedvektoren er $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$.

Ud over at matricen indeholder mange informationer, er den også praktisk med hensyn til rotation og flytning af vektorer og andre matricer, hvilket gøres ved hjælp af matrix transformation, dvs. multiplikation med en tranformationsmatrice.

Hvis man ønsker at rotere en matrix eller en vektor med en vinkel θ omkring én af de tre akser, skal man multiplicere matricen eller vektoren med én af de følgende tre matricer, alt efter hvilken, akse der skal roteres om:

$$M_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Såfremt man vil forlænge en vektor eller en matrice med stedvektoren $\vec{v} = \begin{pmatrix} s \\ t \\ r \end{pmatrix}$

skal man multiplicere med matricen:

$$M_{\vec{v}}(s,t,r) = \begin{pmatrix} 1 & 0 & 0 & s \\ 0 & 1 & 0 & t \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Der er valgt 4×4 matricer for at man kan multiplicere matricerne, egentlig er det kun de tre øverste søjler, der er relevante. Det der ønskes, når man multiplerer matricen med noget, er at værdierne s, t og r bliver lagt til (multipliseret med 1) , hvilket i den ovenstående matrice netop er tilfældet. De 3-dimensionelle vektorer bliver udvidet med et 1 tal således, at den er 4-dimensionel: $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$.

Hvis man har en vektor $\vec{u}$, som man gerne vil rotere omkring x-aksen med 45° og derefter forlænge med vektoren $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$, får man:

$$\vec{u'} = M_x(45°) \cdot M_{vecv}(x,y,z) \cdot \vec{u} = M' \cdot \vec{u}$$

Hvor $\vec{u'}$ er den endelige matrice og $M' = M_x(45°) \cdot M_{vecv}(x,y,z)$.

Appendiks B

Monte Carlo integration

Hvis man skal løse flerdimensionale integraler som i f.eks. global belysning, numerisk er dette ofte meget dyrt tidsmæssigt. Derfor vil det være ønskeligt, hvis man kan bruge en approksimation. En sådan er Monte Carlo metoden.

Monte Carlo integration benytter sig af tilfældigt valgte tal inden for funktionen $f(x)$ grænser, a og b, til at give et estimat for integralet.

$$F = \int_a^b f(x)dx$$

Dette integrale kan approksimeres ved at beregne middelværdien for $f(x)$ i intervallet a til b og gange med længden af intervallet. Dvs at integralet bliver:

$$F = (a - b)f(x)_{mid}$$

Da $f(x)$ middelværdi ikke er kendt, og ej heller nem at beregne kan den approximeres ved at tage gennemsnittet af N gange $f(\xi)$ hvor ξ er et tilfældigt tal i intervallet fra a til b. Integralet kan da skrives som:

$$F_{mc} = \frac{1}{N}(a - b)\sum_{i=1}^{N} f(\xi_i)$$

Jo større N jo bedre approximerer F_{mc} F. For at beskrive hvor hurtigt F_{mc} går imod F kan vi udregne standard afvigelsen σ for F_{mc}. Den fås til:

$$\sigma \propto \frac{1}{\sqrt{N}}$$

Hvilket betyder at vi for at halvere fejlen, dvs F_{mc} er nærmere på F, skal fordoble N.

Appendiks C

Regulære udtryk syntax

```
        <re> ::= <expression> { <expression> }
               | <re> '|' <re>

<expression> ::= <term>
               | <term> '?'
               | <term> '+'
               | <term> '*'

      <term> ::= <label>
               | '(' <re> ')'

     <label> ::= <symbol>
               | '[' <range> { <range> } ']'
               | '[' ']' { <range> } ']'
               | '[' '^' <range> { <range> } ']'
               | '[' '^' ']' { <range> } ']'

     <range> ::= <symbol>
               | <symbol> '-' <symbol>

    <symbol> ::= '.'
               | 0 .. n (any element of alphabet)
               | '\' <symbol>
```

Appendiks D

CD

Indhold

- Raytraceren iVamp til både Mac OS X, Windows og Unix
- Testscener
- Færdige billeder af testscener
- Kildekoden til iVamp
- Javadoc til iVamp
- Java Runtime Environment 1.4