Real-time Ray Tracing
på et SMP Multiprocessor system

Tobias Korning Berthelsen

Kongens Lyngby 2006
IMM-M.Sc.-2006-93

Vejleder: Bernd Dammann og Jackob Andreas Bærentzen

Technical University of Denmark
Informatics and Mathematical Modelling
DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Resumé

Raytracing er en kraftfuld teknik til rendering af billeder med høj realisme, der på mange måder er traditionel (hardware) trekantsrasterisering overlejen. Raytracere har dog den svaghed at de stiller høje krav om rå computerkraft. Denne svaghed har betydet at valget så godt som altid falder på rasterisering, når der skal genereres billeder i real-time. Ved rasterisering må de fleste effekter, som får et billede til at se realistisk ud, designes kunstigt, hvorimod raytraceren, så at sige, tilbyder disse effekter direkte (som en naturlig del af raytraceralgoritmen). Ved brug af en tilstrækkeligt god raytracer vil der således kun være behov for design af selve scenen, der skal renderes, i form af specifikationer for objekter, overflademateriale og lyskilder. Resten (skygger, refleksioner, transmissioner m.m.) vil blive givet som et naturligt resultat af raytracer algoritmen. – Det vil således ikke længere være nødvendigt at bruge mandetimer på kunstigt at designe disse effekter.

Dette projekt handler om hvordan raytracing kan effektiveres, så det bliver muligt at navigere rundt i en 3D scene i real-time. Det essentielle i denne rapport er således de teknikker, der muliggør real-time navigation ved brug parallelisering samt en effektiv datastruktur og gradvist forbedret billedkvalitet (ved rendering af et gradvist forfinet grid og brug af interpolation).

Projektet har resulteret i parallelisering af en forholdsvis simpel raytracer der kan indlæse scener fra *wavefront* filer.
Der er udviklet to løsninger for parallelisering af raytraceren. En løsning der gør brug af OpenMP til at simplificere parallelisering, samt en mere kompleks løsning der benytter POSIX threads. Begge løsninger kan rendere scener bestående af omkring 450.000 trekanter (renderingstiden skalerer logaritmisk i forhold til antallet af trekanter) på 512x512 pixels på ca. 1 sekund. POSIX threads løsningen viser sig dog overlegen, da den løbende leverer outputs i form af interpolerede billeder, der gør det muligt at navigere kameraet rundt i scenen i real-time.

Abstract

Raytracing is a powerful technique for high resolution image rendering. In many ways it is actually superior to traditional (hardware) triangle rasterization. Unfortunately raytracing has one weakness: - It demands a high amount of raw computer power! This almost always results in the choice of rasterization when it comes to real-time image rendering. When rasterization is used most effects which make the image look realistic must be artificially designed. Raytracing on the other hand offers these effects as a natural part of the algorithm. In that way, using a raytracer, you actually only have to design the actual scene including objects, surfaces and light sources. The rest, like shadows, reflections, transmissions etc., will be generated automatically by the raytracer. – Expensive time used to design such effects will no longer be necessary.

This project is about making an efficient raytracer that makes it possible to "walk around" in a 3D scene in real-time. The essential part are those techniques that makes real-time interaction with the scene possible by using parallel executive code, an efficient data structure and by making the image quality improve over several steps till maximum quality is achieved (by rendering a interpolated grid that is refined step by step) .

The outcome of the project is a parallelized relatively simple raytracer that features load of scenes from *wavefront* files.
Two approaches have been used for parallelization. One that uses OpenMP to simplify the process and one, more complex, approach using POSIX threads. Both approaches are capable of rendering 512x512 pixel scenes with about 450,000 triangles in approx. 1 second. The POSIX threads approach shows to be superior because it, in that second, delivers several interpolated outputs that make real-time interaction with the scene possible.

Forord

Dette projekt er udarbejdet i efteråret 2006, som det sidste krav i uddannelsen til Civil Ingeniør på Danmarks Tekniske Universitet (DTU), Kongens Lyngby. Projektet er udarbejdet af Tobias Korning Berthelsen under kyndig vejledning af Bernd Dammann og Jackob Andreas Bærentzen fra Anvendt Matematik, IMM, DTU.

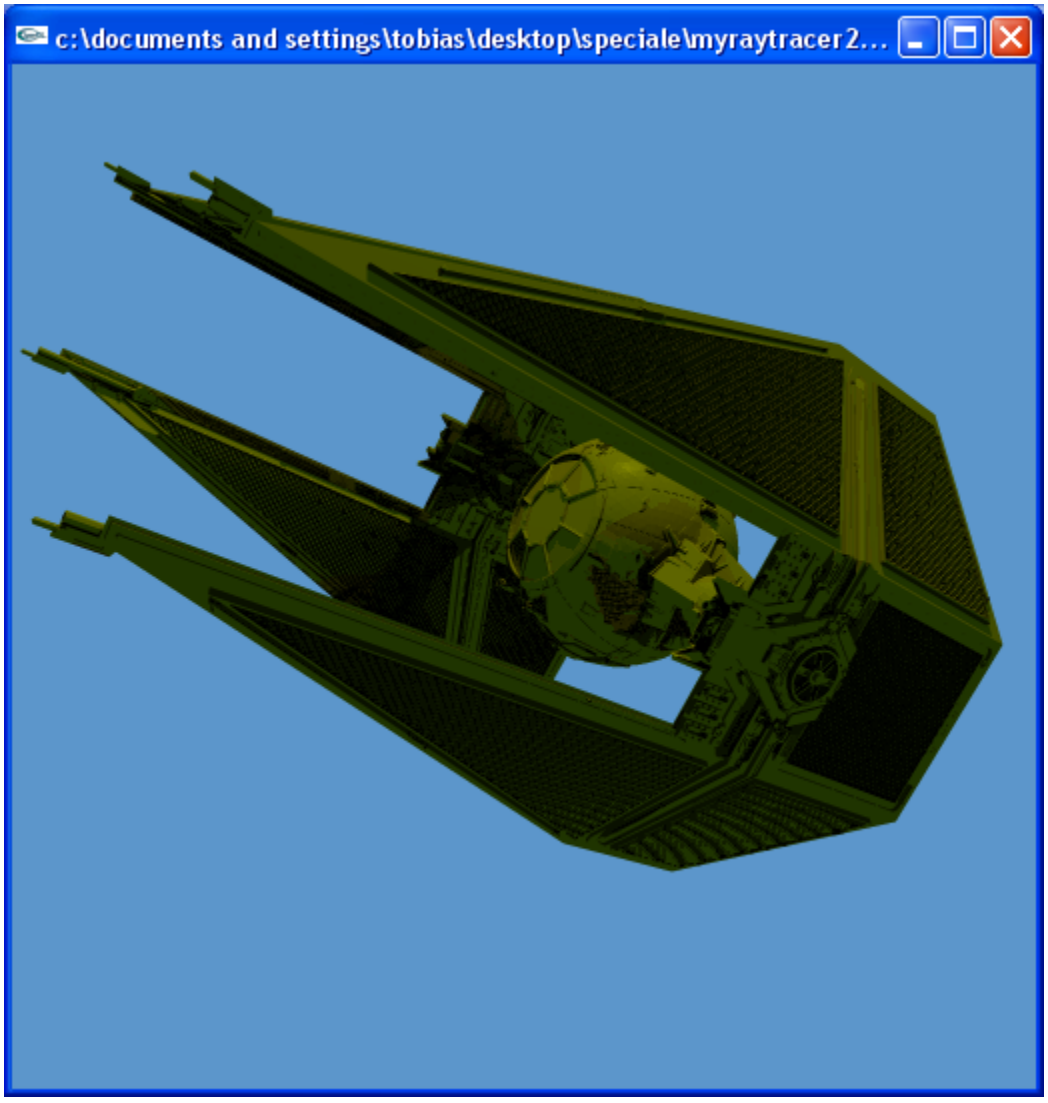
Kongens Lyngby, 31. oktober – 2006

Tobias Korning Berthelsen

Indholdsfortegnelse

Indledning

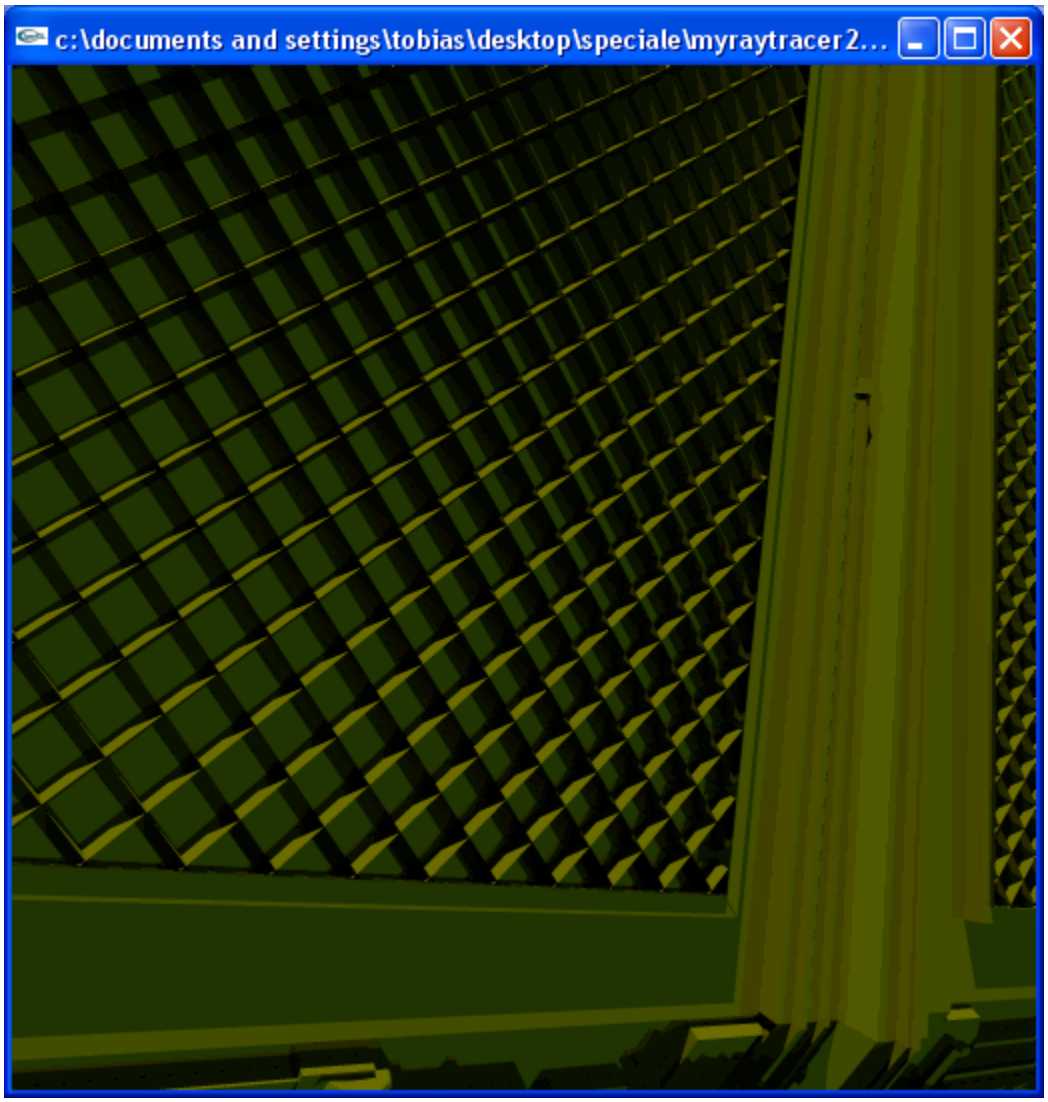
Dette Master projekt omhandler, hvordan parallelisering på et SMP Multiprocessor System kan anvendes til forbedret performance ved raytracing.



Figur 1: TieIntercepter set fra siden, renderet med raytraceren udviklet i dette projekt.

Raytracing er en teknik, som er bredt anerkendt og anvendt når der skal laves computergrafik med høj realisme. En god raytracer kan levere

imponerende billeder med høj detaljegrad (se eksempelvis Figur 2, hvor der er zoomet ind på vingen af TieIntercepteren som ses på Figur 1) og flotte effekter, så som refleksioner, transmissioner, skygger m.m.



Figur 2: Samme TieIntercepter som i Figur 1, hvor der nu er zoomet helt ind på den ene vinge.

Så snart der skal laves computergrafik i real-time bliver der desværre benyttet andre teknikker, selvom disse ikke har det samme potentiale som raytracing. Dette sker fordi raytracing er en meget beregningstung algoritme, der har svært ved at levere billeder i real-time. I stedet benyttes typisk rasterisering, som nutidens grafikkort behandler yderst effektivt. Ved rasterisering er det nødvendigt manuelt at designe mange af de effekter som leveres som en naturlig del af raytraceralgoritmen. Desuden betyder den

stigende kompleksitet i nutidens grafikkort at programmering af grafik til disse ligeledes stiger i kompleksitet.
Hvis det kan lade sig gøre at udvikle en raytracer, der kan rendere billeder i real-time, vil raytraceren i mange nye brancher være et (bedre) alternativ til rasterisering. Bl.a. er spilbranchen et enormt marked, hvor det ville være spændende, hvis raytraceren en dag kunne hjælpe med til, at der opnås endnu bedre realisme, end vi ser det i dag. Derfor giver det god mening at forsøge at optimere raytracing, så der kan leveres billeder i real-time. Optimering kan gribes an fra flere vinkler. Der kan kigges på, om selve implementeringen kan effektiveres ved *old school* hardware optimering, der kan benyttes effektive datastrukturer og der kan kigges på parallelisering. Sidstnævnte har stigende aktualitet i denne tid, hvor de store chip-producenter har stor fokus på at udvikle flerkærnede CPU'er. Der vil ikke gå mange år før størstedelen af hjemme PC markedet har flerkærnede CPU'er. Hvis det således kan lade sig gøre at opnå real-time rendering ved parallelisering, vil raytracing være et seriøst alternativ til rasterisering og vil f.eks. kunne benyttes i spilbranchen.
Jeg vil i dette projekt udvikle en simpel raytracer, som jeg dernæst vil arbejde på at få til at kunne rendere billeder i real-time. Jeg vil først lave en kompakt parallelisering ved brug af OpenMP. Da jeg mistænker det anvendte SMP system (består af 48 1200MHz UltraSPARC IV CPU'er) for ikke at have tilstrækkeligt kraftige processorer til, at real-time rendering vil kunne opnås, vil jeg desuden lave en mere avanceret løsning, der skal muliggøre real-time navigation i scenen. Dette skal ske ved at opbygge billedet over flere steps, hvor der i første omgang hurtigt leveres et billede i lav kvalitet, men som gradvist forbedres indtil den maksimale kvalitet opnås.

Koden er udviklet til at kunne køre på Microsoft Windows såvel som på Unix systemer. Den primære løsning afvikles dog kun på Unix.

Rapporten er delt op i 5 kapitler:

Kapitel 1 omhandler selve raytraceren. Først bliver begrebet *raytracing* overordnet beskrevet hvorefter de væsentligste aspekter beskrives i detaljer. Kapitlet afrundes med en opsummering der beskriver de beslutninger der er taget i kapitlets underafsnit.

Kapitel 2 omhandler parallelisering af raytraceren beskrevet i kapitel 1. Indledende beskrives hvad parallelisering går ud på, dernæst hvordan parallelisering udnyttes i raytracersammenhæng og hvilke strategier der er benyttet i dette projekt.

Kapitel 3 beskriver kort hvilken form for interaktivet der understøttes.

Kapitel 4 omhandler resultater og tests med fokus på paralleliseringen.

Kapitel 5 afrunder rapporten med en konklusion.

Sidst i rapporten vil der desuden kunne findes en alfabetisk sorteret ordforklaring samt referenceliste. I ordforklaringen vil fagudtryk, forkortelser m.fl. blive beskrevet.

1 Raytracing

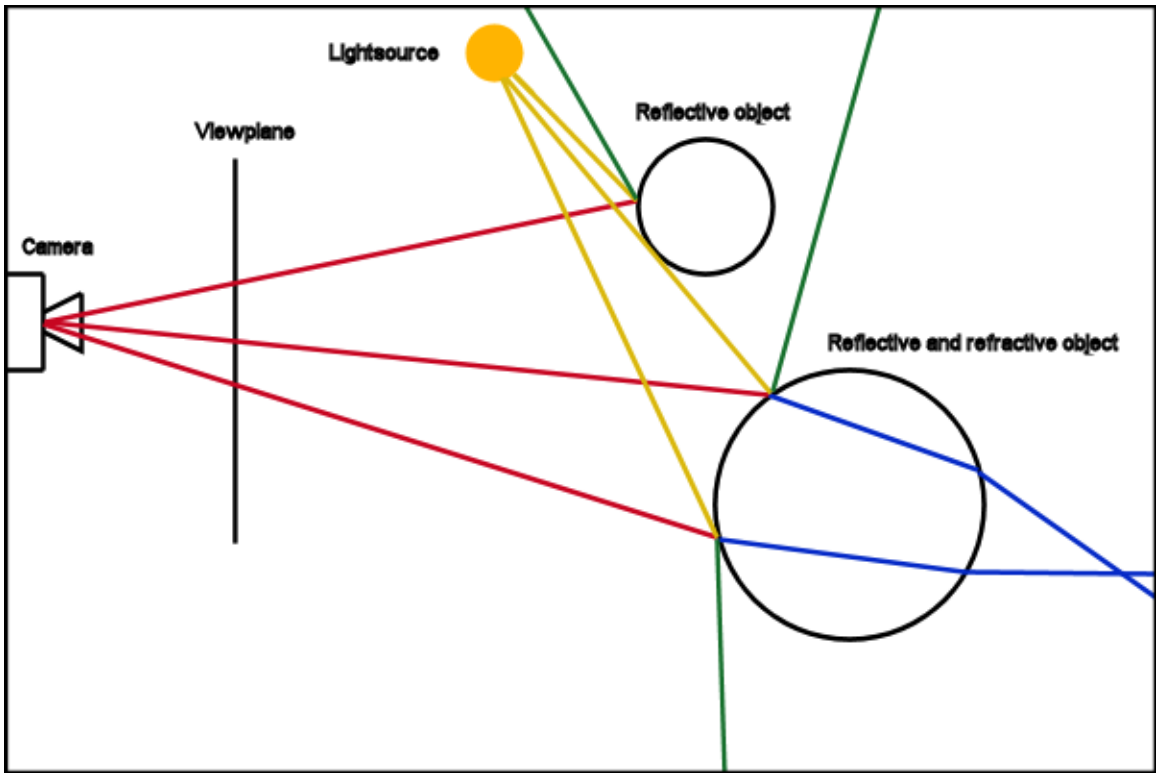
1.1 Introduktion

Raytracing er en intuitiv måde, hvorpå der kan dannes billeder af høj kvalitet med stor realisme. Det intuitive ligger i den måde hvorpå lysstråler følges fra lyskilder, via. eventuelle refleksioner på og brydninger af materialer på objekter indtil de rammer kameraet (eller øjet). Realismen kommer sig af, at der ved beregning på disse stråler og deres interaktion i rummet benyttes fysikkens love. Dette gør sig bl.a. gældende ved beregninger på de nævnte refleksioner og brydninger - her kan der f.eks. tages højde for materialetyper (lys brydes eksempelvis forskelligt i vand og luft). Der kan desuden beregnes realistiske bløde skygger ved at regne på hvor stor en del af lyskilderne i rummet der er synlig på et givet punkt.

Rent praktisk, især hvor hastighed har betydning (hvilket må siges at være tilfældet i dette projekt, hvor målet jo er en real-time raytracer), er det en fordel at lave en enkelt afgørende ændring ved implementering af raytraceren i forhold til naturen lysstråler. Denne ligger i strålernes retning. Da det kun er en meget lille del af lystes stråler, der rent faktisk når frem til et givet øje-punkt, kan der drages stor fordel af at vende strålernes retning. Det man rent praktisk gør, er således, at der dannes en stråle for hver skærmpixel startende fra kameraet (øje-punktet). For hver stråle bestemmes pixelens endelige farve på følgende vis:

Det nærmeste skæringspunkt mellem kameraet og en overflade beregnes.
Overfladens farve i punktet findes.
Farven justeres i forhold til lyskilderne i rummet, samt, hvis overfladen er reflektiv eller refraktiv, bidrag fra andre objekter i scenen.

Dette er det helt overordnede princip ved raytracing.



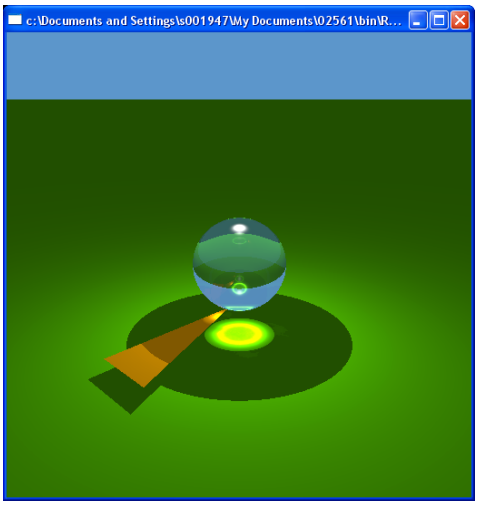
Figur 3: Illustration af hvordan strålerne reflekterer og refrakterer (bryder) objekterne i en scene.

I ovenstående illustration vises det hvordan 3 primære stråler (de røde) skydes ind i scenen. Der er to objekter i scenen, en lille og en stor cirkel. De har begge en reflektiv overflade men den store er også refraktiv. De grønne streger illustrerer de sekundære reflektive stråler og de blå de sekundære refraktive stråler. De gule streger viser vejen til lyskilden i scenen (om punktet er i skygge).

En raytracer opbygges typisk i flere ”bidder”. Først udvikles en helt simpel raytracer, der kan ”oplyse” en simpel scene, ofte kun bestående af en objekttype (f.eks. trekanter eller kugler) som kaster simple ”skarpe” skygger. Herefter vil man typisk tilføje flere lyskilder, og muligvis så kaldte *area* lyskilder, der kaster ”bløde” skygger. Afhængig af formålet kan der så tilføjes flere forskellige objekttyper (måske firkanter el.lign.) eller der kan tilføjes flere effekter så som *caustics*, *colour bleeding* o.lign.

En raytracer kan således udbygges i noget nær det uendelige, og der vil ved hvert implementeret ny effekt opnås et billede der er et skridt nærmere virkeligheden. Det eneste ”men” i denne sammenhæng er, at hver gang en ny effekt implementeres, kan det mærkes i renderingstiden af et billede. Det afhænger således meget af formålet for den enkelte raytracer, hvor vidt man vil gå med implementering af de mange mulige effekter i sin raytracer. – Vil

man have maksimal kvalitet uanset renderingstid, eller vil man gerne give afkald på lidt kvalitet for til gengæld at kunne se resultatet inden for et overskueligt tidsrum?



Figur 4: Eksempel på simpel scene hvor en lyskilde (spotlight) lyser ned på en glaskugle, hvilket resulterer i så kaldt *caustics* på det underliggende objekt.

Strålerne sendes fra øje-punktet og ud i scenen, da det er langt mere effektivt, end hvis strålernes retning vendes, så de følger samme retning som lysets stråler i den virkelige verden[1]. Hvorfor er lettest at forstå, ved at kigge på, hvad der skal til, for at *backward* raytracing skal fungere. Ved *backward* raytracing er der ikke umiddelbart en metode, hvorved det kan afgøres, hvorvidt en given stråle vil bidrage til billedet man ser på skærmen. Man bliver således nødt til at sende stråler fra alle lyskilder mod samtlige objekter i scenen, selv om mange (som oftest klart størstedelen) aldrig vil komme til at bidrage til det billede vi ser på skærmen. – For eksempel vil de stråler der rammer bagsiden af objekter (i forhold til øje-punktet) i scenen kun sjældent (når andre objekter lyses op i form af refleksion) have indflydelse på det endelige resultat.
Ved *forward* raytracing er situationen helt anderledes. Her vil alle stråler bidrage direkte til det endelige resultat i mere eller mindre grad (ved meget reflektive scener skal man være opmærksom på, at bidraget bliver mindre og mindre, for hver refleksion en stråle laver og der bør derfor laves et slags *threshold,* således at strålen afbrydes, når bidraget bliver tilstrækkeligt småt).

[1] Der findes også eksempler på raytracere der starter fra lyskilderne. Denne teknik kaldes normalt for *backward* raytracing. Der er dog en smule forvirring m.h.t. hvad der er *forward* og hvad der er *backward*, hvilket måske er forståeligt nok, da raytracing fra øje-punktet jo umiddelbart er *backward* i forhold til virkeligheden. *Backward* skal i stedet forstås som baglæns i forhold til den første specifikation af en raytracer som blev beskrevet af en fysiker ved navn René Descartes i 1637.

Når det kommer til den rent praktiske implementering, er der en lang række detaljer, der skal tages med i betragtning. Ikke mindst viser det sig, at der skal en stor portion rå computerkraft til for at skabe et højkvalitets billede. Da målet med dette projekt er at opnå realtids gengivelse, må der derfor i høj grad tænkes på optimering af beregninger, effektive datastrukturer samt et godt compilersamarbejde.

1.2 Fokus og begrænsninger

Da alt arbejdet i en raytracer ligger i beregninger relateret til de enkelte stråler der *traces* i scenen, afhænger eksekveringstiden direkte af antallet af stråler der skal *traces*. Der er således, groft beskrevet, to fremgangsmåder til optimering:

1. Minimering af antallet af stråler der skal *traces*.
2. Optimering af beregningstiden for den enkelte stråle.

Punkt 1 kan opnås ved adaptiv sampling, genbrug af samples og ved at formindske antallet af skyggestråler samt andenrangs stråler.
Punkt 2 kan opnås ved *spatial subdivision*, optimering af skæringstests, generel *low-level* optimering (f.eks. udnyttelse af SIMD) og begrænsninger af implementeret realisme (færre effekter betyder færre beregninger).

Jeg vil på baggrund af dette udelade implementering af en del beregningstunge effekter, så som *caustics* og *colour bleeding*. Til gengæld vil jeg fokusere på, at implementere en meget effektiv *spatial subdivision* datastruktur samt en hurtig beregning af skæring mellem stråler og objekter i scenen.

1.3 Shading

Jeg har valgt at benytte The Phong Reflection Model (kapitel 6.3 i [Angel] og [WikipediaI]).
En pixels farve beregnes som summen af de materialefarver som en stråle kommer i kontakt med (via refleksion og refraktion i scenen).

Materialefarven bliver beregnet ud fra strålens indgangsvinkel på materialet, via *Phong*-beregningen, som siger at:

$$I_p = k_a i_a + \sum_{lights} (k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s), \quad \textbf{(1.3.1)}$$

hvor,

I_p er shadeværdien for et overfladepunkt p,

k_a, k_d, k_s er henholdsvis *en* ambient, diffus og spekular konstant,

i_a, i_d, i_s er henholdsvis en ambient, diffus og spekular lysintensitet,

α er en shininess konstant,

L er en retningsvektor fra materialeoverfladen og mod en lyskilde,

N er normalen for materialet,

R er retningvektoren for perfekt refleksion på materialet,

V er retningsvektoren mod øjepunktet.

Ovenstående betyder at materialets intensitet er lig et ambient led plus en summation, over samtlige lyskilder i scenen, af et diffust og et spekulart led. Denne intensitet skal derefter ganges med materialets farve i punktet p, for at få den samlede farve for punktet. Hvis punktet er i skygge, er det kun det ambiente led der bruges.

Der kan tilføjes et afstands led, der giver en så kaldt *fall-off* effekt. - Den effekt at lysbidraget aftager med afstanden fra lyskilden. Dette led skal ganges på det diffuse og spekulare led og ser således ud:

$$dist = \frac{1}{a + bd + cd^2} \quad \textbf{(1.3.2)}$$

hvor *a*, *b* og *c* er konstanter og *d* er afstanden fra lyskilden til skæringspunktet på overfladen.

1.4 Refleksioner og refraktioner

Figur 3 illustrerer at der skal beregnes retningsvektorer ved refleksion og refraktion. Disse beregninger udføres på følgende vis:

Beregning af refleksion:

$$R = V + 2 \cdot N \cdot (-V \bullet N), \tag{1.4.1}$$

hvor R er den reflekterede stråle, V er strålens indgangsvektor og N normalen til overfladen.

Beregning af refraktion (transmission)

$$T = \frac{\eta_i}{\eta_r} V - (\cos\theta_r - \frac{\eta_i}{\eta_r}\cos\theta_i)N, \ \cos\theta_r = \sqrt{1 - \left(\frac{\eta_i}{\eta_r}\right)^2 (1 - \cos^2\theta_i)}, \tag{1.4.2}$$

hvor T er den refrakterede retningsvektor, N er normalen til overfladen, η_i og η_r er henholdsvis incident materialet og refracting materialet og θ_i og θ_r er henholdsvis vinklerne på incident og refraction.

1.5 Skæringstests

Effektive skæringstests har en stor betydning på vejen mod implementering af en hurtig raytracer. Af samme grund har ganske mange lagt et betydeligt stykke arbejde i at optimere disse beregninger. Jeg vil således ikke forsøge at opfinde hjulet på ny, men i stedet sætte min lid til, at de førende forskere på området har gjort deres arbejde, og selv fokusere på at lave effektive implementeringer af deres algoritmer.

1.5.1 Stråle-trekant skæring

Da de fleste, og mest benyttede, standardiserede filformater til 3D-scener, gemmer de forskellige objekter i scenen i form af en masse trekanter[2], er det naturligt at det største arbejde lægges i optimering af beregninger på disse. Faktisk har mange valgt (deriblandt Ingo Wald[3]) kun at understøtte trekanter

[2] Dette er bl.a. gældende for Wavefront formatet (.obj filer), som er meget anvendt.

[3] Betragtes for noget af en guru inden for raytracer verdenen, da han har lavet en lang række gennembrydende raytraceroptimeringer.

i deres raytracere, for derved at opnå maksimal performance (al optimering kan bygges op omkring trekantens fastdefinerede struktur).
Wald giver en god beskrivelse af hans brug af *barycentric coodinates*[4] til sin stråle-trekants beregning. Det er mit indtryk[5] at ingen endnu er kommet på en mere effektiv beregning. Derfor finder jeg det naturligt at implementere denne.

Før vi gør brug af *barycentric coordinates*, kan der laves et hurtigt tjek på om der *traces* i en retning væk fra planen for den pågældende trekant. Dette gøres ved først at beregne afstanden fra strålens origin O til skæringen med planen for trekanten:

Normalen for trekanten:

$$N = (B - A) \times (C - A) \quad \textbf{(1.5.1.1)}$$

Afstanden:

$$afstand = -(O - A) * N / (D * N), \quad \textbf{(1.5.1.2)}$$

hvor D er strålens retning (*direction*).

Hvis *afstand* < 0 kan vi afbryde beregningen nu, da en evt. skæring med trekanten vil ligge bag strålens origin. Ellers beregnes de *barycentric coordinates*.

Kort beskrevet går *barycentric coordinates* ud på følgende. Hvis man har et sæt af punkter: P = [A,B,C,…,N] og et sæt *barycentric coordinates* (en slags vægte til de respektive punkter): [a,b,c,…,n] til beskrivelse af et primitiv[6], kan massecenteret (the *barycenter*) af P vises at ligge i punktet P = aA + bB + cC + … + nN. Yderligere gælder det, at i et givet punkt, at summen af de *barycentric coordinates* a + b + c + … + n = 1 hvis punktet ligger i primitivet.

Dette kan beskrives på følgende vis:
En linie i 2D gående fra P1 til P2 kan beskrives således:

$$P = P1 + a(P2 - P1) \quad \textbf{(1.5.1.3)}$$

[4] Barycentric coordinates blev beskrevet af Franz Ferdinand Möbius i 1827.
[5] Jeg bygger dette på, at der i div. seriøse fora på nettet, omhandlende real-time raytracere, henvises til Walds tesis når der diskuteres effektiv implementering af stråle-trekants beregning.
[6] Kan være en trekant, en firkant eller et andet flerkantet primitiv.

Vi befinder os på linien så længe a er mellem 0 og 1. Dette kan omskrives til:

$$P = a1\ P1 + a2\ P2 \qquad \textbf{(1.5.1.4)}$$

hvor a1 og a2 er *barycentric coordinates* til punktet P.

En trekant kan således beskrives ved tre *vertices* A, B og C og deres *barycentric coordinates*. Et hvert punkt p på denne trekant kan beskrives som den vægtede sum af disse tre *vertices*:

$$P = aA + bB + cC, \qquad \textbf{(1.5.1.5)}$$

hvor $a + b + c = 1$ hvis P ligger i planen for trekanten.

Denne sammenhæng kan bruges til vores skæringstest.

Hvis vi først omskriver reglen for summen af de *barycentric coordinates* således:

$$a + b + c = 1 \Leftrightarrow$$
$$a = 1 - b - c$$

og herefter substituerer a med 1-b-c i ligning 1.5.1.3 får vi:

$$P = (1 - b - c)A + bB + cC \Leftrightarrow$$
$$b(B - A) + c(C - A) = P - A, \qquad \textbf{(1.5.1.6)}$$

hvor $b \geq 0$, $c \geq 0$ og $b + c \leq 1$

Nu kan vi løse ligningerne til de *barycentric coordinates b* og *c*. Der er dog et trick der gør beregningen noget lettere. Vi kan projektere trekanten ind på et hvilket som helst andet plan (undtagen et der ligger retvinklet med trekanten) uden at det ændre på de *barycentric coordinates*. Vi kan således projektere trekanten ind på en af vores akser i koordinatsystemet og derved simplificere beregningen[7].

[7] For at opnå numerisk stabilitet er det en god ide at vælge den dominerende akse for trekantens normal.

Ved at definere *k, j* og *p* ud fra følgende substitutioner:

$$\begin{aligned} k &= \mathrm{B} - \mathrm{A} \\ j &= \mathrm{C} - \mathrm{A} \\ p &= \mathrm{P} - \mathrm{A} \end{aligned} \qquad \textbf{(1.5.1.7)}$$

kan *b* beregnes ud fra ligning 1.5.1.4:

$$\begin{aligned} & bk_x + cj_x = p_x \Leftrightarrow \\ & c = \frac{p_x - bk_x}{j_x} \Rightarrow \\ & bk_y + \frac{p_x bk_x}{j_x} j_y = p_y \Leftrightarrow \\ & bk_y j_x + p_x j_y - bk_x j_y = p_y j_x \Leftrightarrow \\ & bk_y j_x - bk_x j_y = p_y j_x - p_x j_y \Leftrightarrow \\ & b(k_y j_x - k_x j_y) = p_y j_x - p_x j_y \Leftrightarrow \\ & \underline{b = \frac{p_y j_x - p_x j_y}{k_y j_x - k_x j_y}} \end{aligned} \qquad \textbf{(1.5.1.8)}$$

På tilsvarende vis kan *c* beregnes til:

$$\underline{c = \frac{p_y k_x - p_x k_y}{j_y k_x - j_x k_y}} \qquad \textbf{(1.5.1.9)}$$

For at beregne om et givet punkt i planet for trekanten ligger i selve trekanten er det således nok at beregne *b* og *c* og tjekke at $b \geq 0$, $c \geq 0$ og $b + c \leq 1$.

Forudberegning:

Det er en stor fordel hvis beregningstunge udtryk kan forudberegnes, således at de kun udregnes en enkelt gang, frem for ved hver skæringstest. Langt størstedelen af udtrykket for de *barycentric coordinates* kan forudberegnes.

Ved at ændre lidt på udtrykkene for *b* og *c* kan det arrangeres sådan at de kun afhænger af skæringspunktet mellem strålen og planen for trekanten:

$$b = \frac{p_y j_x - p_x j_y}{k_y j_x - k_x j_y} \Leftrightarrow$$
$$b = \frac{1}{k_y j_x - k_x j_y} \cdot (p_y j_x - p_x j_y) \qquad \textbf{(1.5.1.10)}$$

Nu substituerer vi p med p = (P – A), fra tidligere (ligning 1.5.1.5). Dette giver følgende ligning, hvor kun P_x og P_y varierer (den øvrige del af udtrykket kan derfor forudberegnes):

$$b = \frac{1}{k_y j_x - k_x j_y} \cdot (P_y j_x - A_y P_x - P_x j_y + j_y A_x) \Leftrightarrow$$
$$b = \frac{j_x}{k_y j_x - k_x j_y} \cdot P_y - \frac{j_x}{k_y j_x - k_x j_y} \cdot P_x + \frac{j_y A_x - j_x A_y}{k_y j_x - k_x j_y} \qquad \textbf{(1.5.1.11)}$$

1.5.2 Stråle-kugle skæring

En kugle kan udtrykkes således,

$$(p - c) \cdot (p - c) = r^2 \qquad \textbf{(1.5.2.1)}$$

hvor *p* er et vilkårligt punkt på kuglen, *c* er kuglens center og *r* kuglens radius.

Hvis vi har strålens ligning angivet på parameterform,

$$p(t) = o + td \qquad \textbf{(1.5.2.2)}$$

hvor *o* er strålens udgangspunkt og *d* strålens retning og $t \geq 0$.

Ved at substituerer dette udtryk ind i udtrykket for kuglen (1.5.2.2 med 1.5.2.1) fås,

$$(o + td - c) \cdot (o + td - c) = r^2 \qquad \textbf{(1.5.2.3)}$$

Ved at løse dette med hensyn til *t* fås følgende 2. grads ligning,

$$At^2 + Bt + C = 0\,, \qquad \textbf{(1.5.2.4)}$$

hvor,

$$A = d \cdot d$$
$$B = 2(o - c) \cdot d$$
$$C = (o - c) \cdot (o - c) - r^2$$

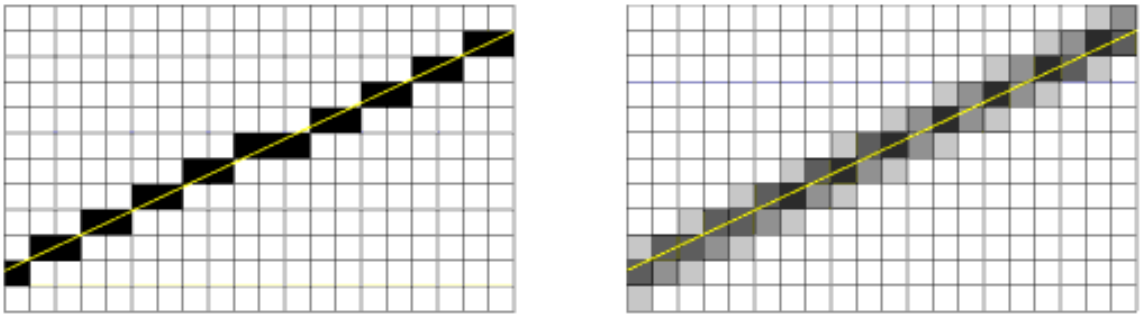
Da *d* i denne sammenhæng er en enhedsvektor kan skæringen beregnes således (i pseudokode),

$$B = (o - c)$$
$$C = (o - c) \cdot (o - c) - r^2$$
$$D = B^2 - C$$

$$Hvis\ \ D > 0,\ \ så\ \ skæres\ \ kuglen\ \ i\ \ \pm B - \sqrt{D}$$

1.6 Supersampling

Supersampling er en *antialiasing* teknik. *Aliasing* ses i billeder i form af kantede linier, som opstår fordi en computerskærm er opdelt i firkantede pixels, som hver kun kan have én farve. Når der tegnes en linie som går på tværs af skærmen, vil denne fremstå hakket.
Man kan forbedre dette ved at forsøge at udglatte disse kanter, ved at lave lidt mere gradvise farveskift:



Figur 5: Billedet illustrer henholdsvis en aliased og en antialiased streg. Se [Schorsch].

Selve supersamplingen foregår ved at der skydes flere stråler (samples) igennem den samme pixel, hvorefter pixelens farve bestemmes ved

interpolation. Teknikken er naturligvis kostbar, da fx 2x2 supersampling giver 4 gange så lang renderingstid, - og det selvom billedet langt fra giver 4 gange så god kvalitet, da der i mange pixels ikke er noget at vinde ved supersampling (hele pixelen har samme farve). For at udbedre denne lineære sammenhæng indføres begrebet adaptiv supersampling, som handler om kun at lave supersampling der hvor der virkelig er noget at hente, - nemlig på kanterne af objekter i scenen. Detektion af kanter kan gøres på flere måder:

1.6.1 Objekt identifikation

Ved at lade hver enkelt stråle indeholde et ID på et evt. objekt den ramte, kan der dannes et 2 dimensionalt kort der indeholder ID'er for hver enkelt pixel. De steder hvor to tilstødende pixels har forskellige ID'er, skal der supersamples.
Dette giver dog ikke antialiasing af skygger fra spotlights, hvorfor ID kortet skal udvides. Dette kan gøres ved at lade strålen returnere en random værdi ved skygger fra spotlights. Dette giver godt nok supersampling af hele skyggeområdet, men vil stadig være langt at foretrække i forhold til supersampling af hele billedet.
Hvor effektiv denne teknik afhænger af antallet af objekter i scenen. I simple scener er denne teknik således hurtig, men ved meget komplekse scener vil det være mere effektivt at benytte traditionel adaptiv supersampling (se 1.6.2).

1.6.2 Pixel-pixel analyse

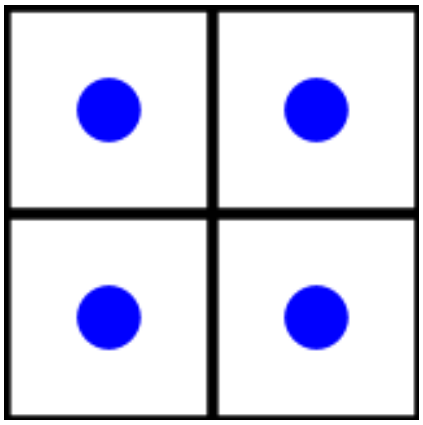
Denne metode er mere traditionel (metoden der normalt henvises til når der tales om adaptiv supersampling*)*. Her laves der typisk først en sampling i hvert pixel hjørne. Herefter testes om forskellen på de fire samples er over et givent *threshold*. Hvis dette er tilfældet opdeles pixelen i et antal subpixels. Proceduren er velegnet til at blive udført rekursivt, sådan at hver pixel opdeles i 4 subpixel som testes op i mod det givne *threshold* og opdeles yderligere, hvis nødvendigt, indtil alle subpixels opfylder *threshold*-kriteriet eller, som "nød stop", den rekursive dybde kommer over en given værdi.

Opdelingen i subpixels

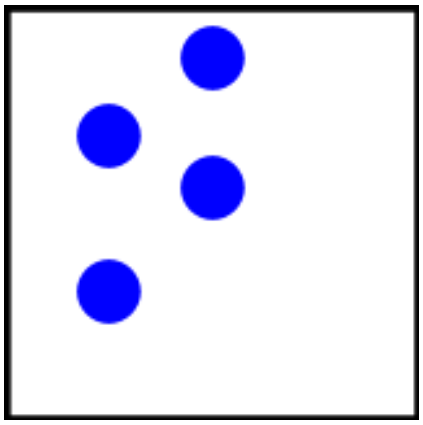
Hvordan de pixels der skal supersamples, opdeles, har ganske stor betydning for resultatet.

Grid:

En pixel opdeles i et jævnt grid, hvorefter farven for hver sub-pixel bestemmes ved at *trace* nye stråler fra midten af hver sub-pixel. Se Figur 6.



Figur 6: Viser en pixel delt op i 4 *sub*-pixels. De blå cirkler, er de nye stråler der *traces*.



Figur 7: Random supersampling.

Random:

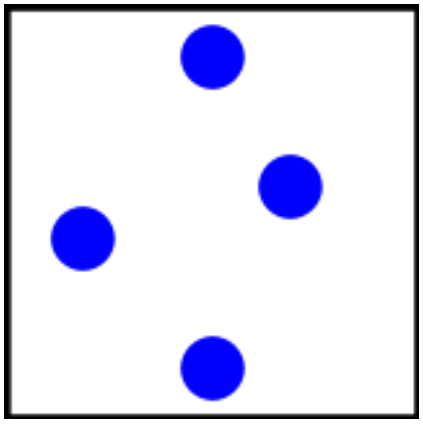
Et antal nye stråler skydes af sted fra tilfældige positioner inden for pixelens fire hjørner. Se Figur 7.

Poison Disc:

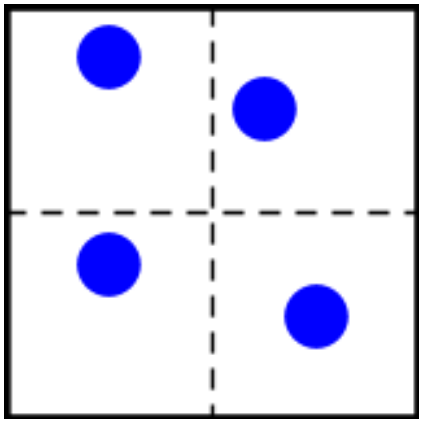
Som random, men hvor der tjekkes at ingen af de nye stråler ligger for tæt på hinanden. Se Figur 8.

Jittered:

Pixelen gridopdeles først i et antal subpixels, hvorefter der skydes nye stråler fra tilfældige positioner inden for disse. Se Figur 9.



Figur 8: Poison Disc supersampling.



Figur 9: Jittered supersampling.

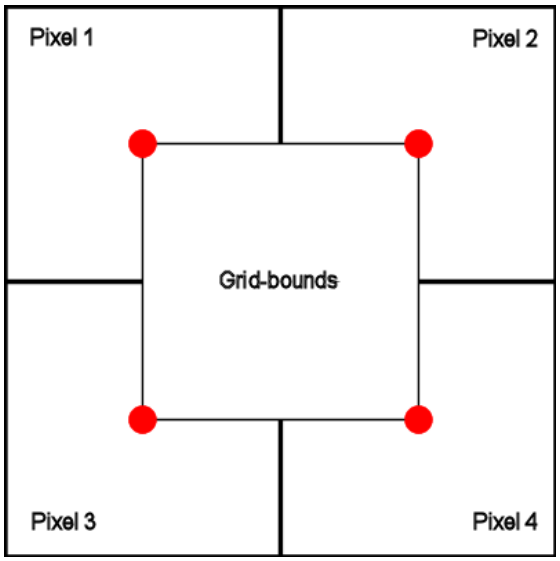
1.6.3 Min implementering

Jeg har implementeret den traditionelle adaptive supersampling, hvor der supersamples hvis farveforskellen mellem nabo-pixels er større end et givent *threshold*. For at være lidt på tværs af den gængse standard, har jeg udviklet min egen version af grid opdelingen.

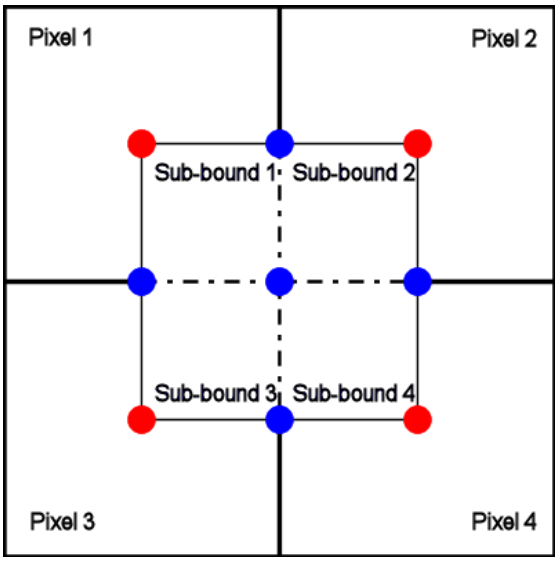
Forløbet er som følger:

Alle skærmens pixels gennemløbes fra en ende af. Hver pixel-farve sammenlignes med dens højre nabo, nedre nabo samt med pixelen nede til højre. Hvis en af disse pixelfarver afviger mere end 0,1 fra en af de øvrige, så foretages der supersampling.

Supersamplingen foretages ved at *trace* 5 nye stråler i området kaldet *grid-bounds* (på Figur 10). Disse, sammenlagt, 9 farveværdier danner nu fire sub-bounds:



Figur 10: Supersampling af pixel 1. Der dannes et grid ud fra pixel 1, 2, 3 og 4.



Figur 11: Supersampling af sub-pixelen (grid-bounds) fra Figur 10.

For hver *sub-bound* startes processen forfra indtil forskellen på de fire farver i hjørnerne på en *sub-bound* bliver mindre end 0,1. Når det er tilfældet, så returneres den interpolerede farve.

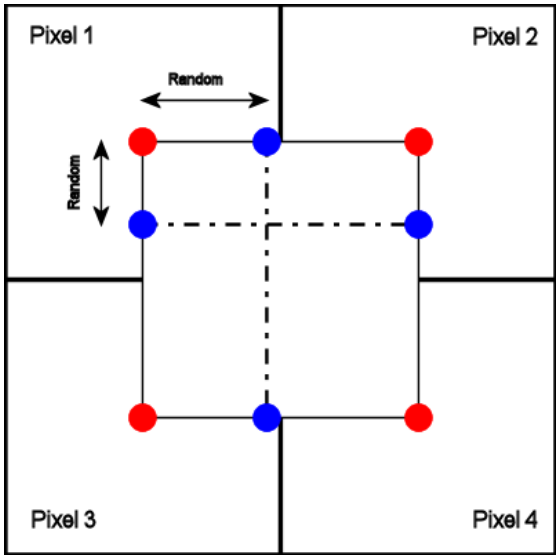
Nedenfor ses koden for den rekursive funktion der foretager adaptive supersampling som beskrevet:

```
Vec3f RenderEngine::GridSupersampling(Quard &q, int depth)
{
   if(!(q.BelowThreshHold(.1f)) && depth < 2)
   {
      //Get coords of new rays
      Vec2f topMiddleCoord = (q.GetTopLeftCoord() +
q.GetTopRightCoord()) / 2;
      Vec2f bottomMiddleCoord = (q.GetBottomLeftCoord() +
q.GetBottomRightCoord()) / 2;
      Vec2f leftMiddleCoord = (q.GetTopLeftCoord() +
q.GetBottomLeftCoord()) / 2;
      Vec2f rightMiddleCoord = (q.GetTopRightCoord() +
q.GetBottomRightCoord()) / 2;
      Vec2f centerCoord = (q.GetTopLeftCoord() +
q.GetBottomRightCoord()) / 2;

      //Get new colors
      Vec3f topMiddleColor = TracePos(topMiddleCoord);
      Vec3f bottomMiddleColor = TracePos(bottomMiddleCoord);
      Vec3f leftMiddleColor = TracePos(leftMiddleCoord);
      Vec3f rightMiddleColor = TracePos(rightMiddleCoord);
      Vec3f centerColor = TracePos(centerCoord);

      //Split new quards
      depth++;
      Vec3f tmpColor = GridSupersampling(Quard(q.GetTopLeftColor(),
q.GetTopLeftCoord(), topMiddleColor, topMiddleCoord, leftMiddleColor,
leftMiddleCoord, centerColor, centerCoord), depth);//top left
      tmpColor += GridSupersampling(Quard(topMiddleColor,
topMiddleCoord, q.GetTopRightColor(), q.GetTopRightCoord(),
centerColor, centerCoord, rightMiddleColor, rightMiddleCoord),
depth);//top right
      tmpColor += GridSupersampling(Quard(leftMiddleColor,
leftMiddleCoord, centerColor, centerCoord, q.GetBottomLeftColor(),
q.GetBottomLeftCoord(), bottomMiddleColor, bottomMiddleCoord),
depth);//lower left
      tmpColor += GridSupersampling(Quard(centerColor, centerCoord,
rightMiddleColor, rightMiddleCoord, bottomMiddleColor,
bottomMiddleCoord, q.GetBottomRightColor(), q.GetBottomRightCoord()),
depth);//lower right

      return tmpColor / 4;
   }
   else return q.GetInterpolatedColor();
}
```

Figur 12: Rekursiv funktion der udfører adaptive supersampling.

Jeg har desuden forsøgsvis implementeret en blanding mellem den ovenfor beskrevne *grid*-opdeling og den *jittered* pixel-opdeling. Den fungerer ved at

opdele *grid-bounden* på et tilfældigt sted vandret og lodret, sådan at udgangspositionen for de 5 nye stråler der traces, placeres således:



Figur 13: Illustration af min blanding mellem en slags grid/jittered pixelopdeling.

Jeg vil ikke beskrive denne metode i detaljer, da den viste sig at være mindre effektiv end *grid* versionen.

1.7 Spatial Subdivision

1.7.1 Generelt

Der findes en lang række datastrukturer der er forsøgt benyttet til raytracing. Den struktur der samlet set (over en lang række forskellige scenetyper), ifølge Vlastimil Havran har vist sig bedst, er det såkaldte kD-træ.
Vlastimil Havran har lavet en større afhandling om spatial subdivision schemes (deriblandt *regular grids, nested grids, octrees* og *kd-trees*) og kom til den konklusion at kD-træer slår de andre strukturer i de fleste tilfælde [HavranPhd2001]. På baggrund af Havrans arbejde, vælger jeg kun at fokusere på kD-træet i denne rapport.

1.7.2 kD-træer

Et kD-træ er en speciel type af Binary Space Partioning (BSP) træet. BSP træer blev først beskrevet af Shumacker m.fl. [Shumacker] i 1969. Som navnet antyder, går teknikken ud på, at opdele ”space” i mindre dele. Opdelingen giver den fordel at kun en lille del af scenens samlede objekter

skal tages med i en given beregning. Hver node i et BSP træ har højst to grene.
Det tager noget tid at bygge et BSP træ, hvorfor træet bygges ved pre-processing, men tiden er givet godt ud, da strukturen giver betydeligt forbedret performance i *run-time*. BSP-træer er kun egnet til håndtering af statiske objekter, da træet ikke umiddelbart kan omstruktureres i *run-time*[8].
I 90'erne begyndte spilindustrien at gøre brug af BSP-træer for at opnå bedre performance samt, som direkte følge deraf, muligheden for at have flere detaljer i scenen.

k'et i kD-træ henviser til at denne type træer kan opbygges i *k* dimensioner. I raytracer sammenhæng vil der således være tale om et 3D-træ, da de scener der arbejdes med er opbygget 3-dimensionelt.

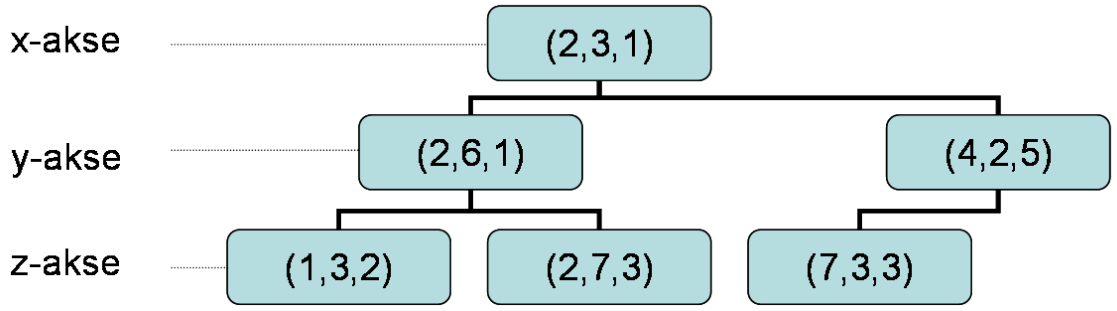
Opbygning af kD-træ er grundlæggende ganske simpelt. Her er et eksempel på hvordan dette kan gøres (i pseudo-kode):

```
void buildkdtree( Node* node )
{
  if (stopcriteria()) return
  splitpos = findsplitposition()
  leftnode = new Node()
  rightnode = new Node()
  for (all primitives in node)
  {
     if (node->intersectleftnode()) leftnode->addprimitive( primitive
)
     if     (node->intersectrightnode())     rightnode->addprimitive(
primitive )
  }
  buildkdtree( leftnode )
  buildkdtree( rightnode )
}
```

Figur 14: Pseudokode til opbygning af et kD-træ. Opbygningen foregår rekursivt og fortsætter indtil et givent stopkriterium er opnået.

I Figur 15 ses et eksempel på hvordan 6 punkter kan fordeles i et kD-træ ved skiftevis at splitte de 3 akser. Først splittes x-aksen, hvilket betyder at alle punkter med en x-værdi mindre eller lig x-værdien (lig 2) i den øverste node (roden) hører til den venstre gren og resten til den højre. På det næste niveau splittes y-aksen og punkterne fordeles efter samme princip som ved x-aksen. På den måde fortsættes der indtil alle punkter har fundet en plads i træet.

[8] For ganske nylig er der dog udviklet et Bounding Interval Hierarchy (se [BIH]) der kan bygges i realtid, hvilket giver mulighed for dynamiske scener.



Figur 15: Illustration af hvordan et kD-træ opbygges ved skiftevis at splitte de 3 akser, x, y og z.

Hver node indeholder information om den nøjagtige splitposition (N_{pos}), hvilken akse der bliver splittet (N_{axis}), samt hvilke objekter der tilhører noden.
Det smarte ved en træ-struktur er, at man meget hurtigt kan bestemme, om en given stråle der "skydes" ind i scenen, med udgangspunkt i en pixel på skærmen, rammer et objekt i scenen. Fremgangsmåden er følgende:

Strålens nærmeste og fjerneste punkt, i forhold til scenens *boundingbox*, bestemmes:

$$P_{near} = ray_{origin} + ray_{direction} \cdot bbox_{near}$$
$$P_{far} = ray_{origin} + ray_{direction} \cdot bbox_{far}$$

Nodens splitakseværdi sammenlignes med splitakseværdien for P_{near} og P_{far}:

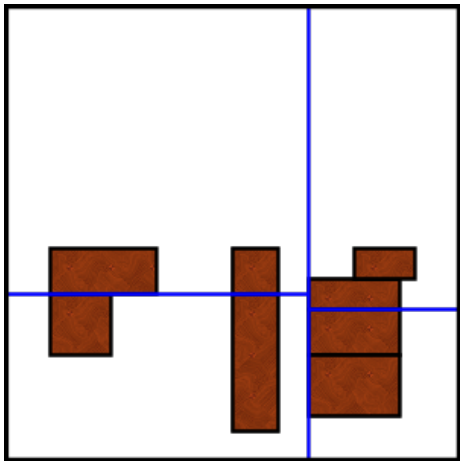
1. Hvis $P_{near}[N_{axis}]$ og $P_{far}[N_{axis}] > N_{pos}$, gå til højre i træet.
2. Hvis $P_{near}[N_{axis}]$ og $P_{far}[N_{axis}] < N_{pos}$, gå til venstre i træet.
3. Ellers, gå til venstre OG til højre i træet.

Denne fremgangsmåde fortsættes indtil man befinder sig på et *leaf* [9] i træet. Her laves der så skæringstests med alle de objekter som noden "peger" på.

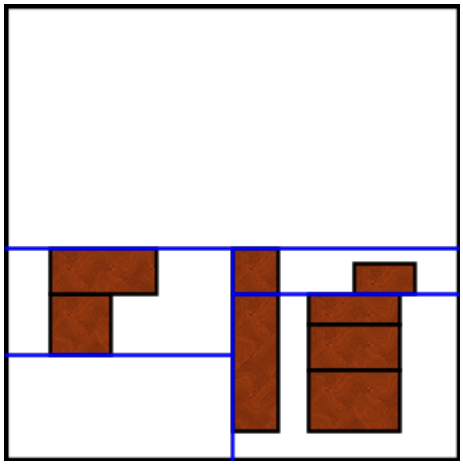
kD-træer adskiller sig fra BSP-træer ved altid at splitte vinkelret i forhold til en af koordinatsystemets akser. Hvilken akse der splittes, kan afgøres på forskellig vis. I nogle sammenhænge vil det være fordelagtigt altid at splitte den "længste akse", i andre sammenhænge splittes akserne på skift (enten blot ved halvering, eller også i medieanen). Der er således mange måder at opbygge et kD-træ på. Ingen af de nævnte "splitstrategier" er dog velegnet i raytracer sammenhæng, idet at de alle typisk vil ende op i ganske

[9] Et "blad" i træet. – Det nederste niveau.

velbalanceret træer, hvor hver *leaf* node peger på et eller flere objekter. Et balanceret træ er naturligvis ingen dårlig ting. Men det er det derimod at alle *leaf* noder peger på et objekt, da dette betyder at en given stråle aldrig vil ramme en tom node, og således altid skal lave skæringstests med et eller flere objekter.
En bedre struktur er at placere splitplanerne på en måde så objekterne isoleres fra evt. tomme områder af scenen. Med denne opdeling vil det ofte slet ikke være nødvendigt at lave de kostbare skæringstests, hvilket selvsagt giver en betydelig hastighedsforbedring. Figur 16 og Figur 17 illustrerer forskellen på de to forskellige "akse-split strategier".



Figur 16: Viser en typisk kD-træ opdeling af et antal objekter i en 2D scene.



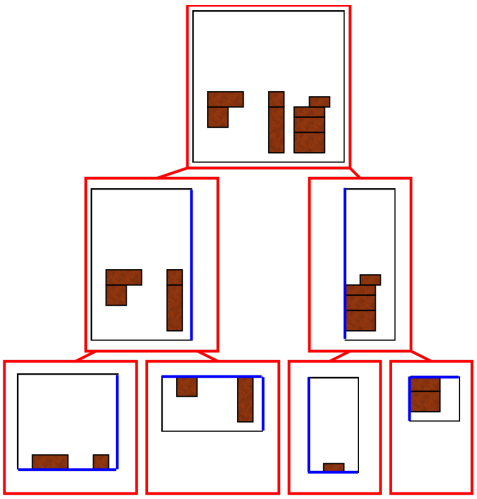
Figur 17: Viser en, i raytracer sammenhæng, bedre egnet fordeling af objekterne i en 2D-scene.

Den yderste sorte ramme specificerer skærmen, de blå streger er splitpositionerne (der afgrænser de enkelte noder) og de brune firkanter er objekter i scenen. Alle pixels i Figur 16, må lave skæringstests med to objekter, uanset hvor i scenen de rammer. I Figur 17 er situationen derimod en helt anden. Her vil alle de pixels der traces i den øverste halvdel af billedet ingen skæringstests skulle lave.

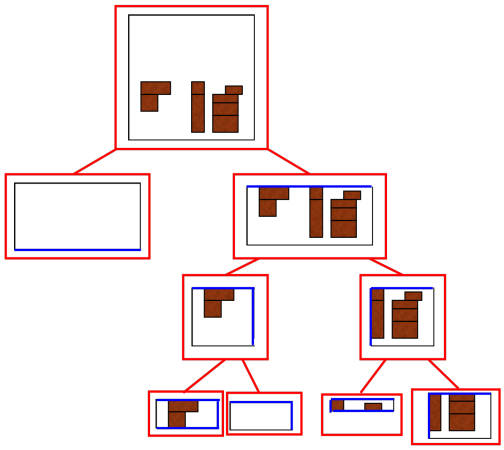
I Figur 18 og Figur 19 illustreres det, hvordan de to ovenstående eksempler på opdeling af en scene vil se ud i et kD-træ.

For at opnå de bedst mulige akse-splitpositioner, må der regnes på hvilken, af alle de mulige splitpositioner, der er den "billigste". Da der som udgangspunkt er uendeligt mange mulige splitpositioner, giver det god mening at begrænse sig til kun at tillade at en akse splittes på hjørner eller kanter (hvis disse ligger parallelt med den pågældende akse) af objekter i

scenen. Da chancen for at ramme en given *voxel*[10] i scenen er relateret til overfladearealet af denne, kan det beregnes hvor 'dyr' en *voxel* er før og efter den splittes, for på den måde at vurdere om et split er nyttigt.



Figur 18: Scenen fra Figur 16 distribueret ud i noderne af et kD-træ.



Figur 19: Scenen fra Figur 15 distribueret ud i noderne af et kD-træ.

Hvis man forudsætter, at alle stråler er jævnt fordelt og ikke blokeres af objekter i scenen (ikke videre realistisk, men stadig ganske anvendeligt), kan det beregnes hvor stor sandsynligheden er for, at hvis en stråle rammer en given *voxel*, hvor stor sandsynligheden da er for at en af dens *sub-voxels* bliver ramt.
Mere matematisk; hvis vi har en voxel V som opdeles i to voxels Vl og Vr, da kan sandsynligheden for at disse voxels rammest beregnes på følgende vis:

$$P(V_L \mid V) = \frac{SA(V_L)}{SA(V)} \quad og \quad P(V_R \mid V) = \frac{SA(V_R)}{SA(V)}, \qquad \textbf{(1.7.2.1)}$$

hvor $SA(V) = 2 \cdot (V_h V_b + V_h V_d + V_b V_d)$ er overfladearealet af V og hvor h, d og b er henholdsvis højden, dybden og bredden af V.

[10] Voxels er en betegnelse for de "klodser" scenen bliver opdelt af, af splitplanerne i kD-træet. Når der tales om at splitte en voxel, svarer det således til at opdele en node i kD-træet i to nye under noder. Voxels kan dog også bruges i andre sammenhænge. F.eks. bruges begrebet ofte til at betagne hvad der svarer til en pixel i 3D.

Hvis man desuden bestemmer gennemsnitsprisen for at bevæge sig fra en voxel til en sub-voxel C_{trav} samt gennesnitsprisen for en skæringstest $C_{intersect}$ kan gennemsnitsprisen for at opdele V til V_L og V_R beregnes:

$$Cost_{split}(V_L, N_L, V_R, N_R) = C_{trav} + C_{intersect}(P(V_L \mid V)N_L + P(V_R \mid V)N_R), \quad \textbf{(1.7.2.2)}$$

hvor N_L og N_R er antallet af objekter i henholdsvis V_L og V_R.

Et naturligt stopkriterium for opbygningen af kD-træet vil være når den estimerede pris for at bevæge sig igennem et *leaf* er mindre end prisen for det billigst estimerede split.

Denne strategi kaldes for *Surface Area Heuristic* (eller blot *SAH*) og blev introduceret af MacDonald og Booth [MacDonald89, MacDonald90].

Bestemmelse af splitpositioner ud fra denne strategi giver et kD-træer der har en yderst effektiv *run-time* performance. Desværre er det en meget kostbar affære at bygge kD-træer på denne måde. For ikke at skulle bygge kD-træet op hver gang en scene skal renderes, vil det derfor være en stor fordel at kunne gemme de byggede kD-træer i en fil, sådan at de hurtigt kan hentes igen uden at træet skal bygges forfra. kD-træet skal således opbygges på en måde der er velegnet til kunne blive gemt på en fil. Dette kan gøres ved at arrangere noderne i kD-træet, samt objekterne i scenen pænt i et array. Dette giver en stor fordel når kD-træet skal gemmes. Hvis kD-træet derimod opbygges rent pointerbaseret (ved blot at lave et *new node(...)* kald for hver gang der skal oprettes en ny node), vil det være yderst vanskeligt at gemme træet i en fil.

1.8 SIMD optimering

SIMD står for Single Instruction Multiple Data, og er således en teknologi indbygget i nyere processorer der muliggør beregning på 4 floating points samtidigt.
Denne teknologi er yderst anvendelig i raytracer sammenhæng, da man ved at samle stråler i pakker af fire, ved stor sandsynlighed vil skulle foretage skæringsberegninger op i mod det samme objekt for alle strålerne i pakken. Derved kan der drages stor fordel af SIMD.

Rent praktisk kunne følgende kode for en ray:

```
struct ray
{
  float posX, posY, posZ;
  float dirX, dirY, dirZ;
};
```

ændres til denne RayPacket struktur:

```
struct RayPacket
{
  union
  {
     struct
     {
       union { float posX[4]; __m128 posX4; };
       union { float posY[4]; __m128 posY4; };
       union { float posZ[4]; __m128 posZ4; };
       union { float dirX[4]; __m128 dirX4; };
       union { float dirY[4]; __m128 dirY4; };
       union { float dirZ[4]; __m128 dirZ4; };
     };
  };
};
```

Denne union mellem SIMD 128bit værdierne[11] og float arrays kan de enkelte værdier tilgås via float arrayet.

Der kan nu regnes på SIMD værdierne, ved brug af forskellige specielle SIMD funktioner, på følgende vis:

```
__m128 v1 = _mm_mul_ps( RP->dirX4, RP->dirX4);
__m128 sum = _mm_add_ps( v2, RP->dirX4 );
__m128 sqrroot = _mm_sqrt_ps( sum );
__m128 reciprocal = _mm_div_ps( one, sqrroot );
RP->dirX4 = _mm_mul_ps( RP->dirX4, reciprocal );
```

_mm_mul_ps, _mm_div_ps og _mm_add_ps henholdsvis, ganger, dividerer og plusser to __m128 variabler og _mm_sqrt_ps beregnere kvadratroden.
'one' er foruddefineret/-initialiseret SIMD variabel der består af fire 1.0f'er.

At "klatre" rundt i kD-træet bliver noget mere kompliceret ved brug af RayPackets, idet at alle stråler ikke altid vil følge samme vej ned gennem træet. Dette problem kan løses ved, at lade alle stråler passere en gren i træet, hvis blot en enkelt rent faktisk skal den vej og så indføre en form for gyldighedsmarkering af de enkelte stråler i pakken (de stråler som ikke

[11] __m128 er en SIMD data type som automatisk bliver tilpasset til 16byte blokke i hukommelsen.

skulle have været samme vej ned gennem træet erklæres for ugyldige, og må siden hen traces selvstændigt). Så snart strålerne i en RayPacket ikke følger den samme vej ned gennem træet, vil der selv sagt være begrænset eller ingen forbedring i performance for netop denne pakke. Heldigvis vil langt størstedelen af RayPackets følges ad, - i hvert fald indtil et objekt rammes, hvorfor RayPackets her vil give en væsentlig performance forbedring. Ved efterfølgende *tracing* (reflekterede og transmiterede stråler) er der større tendens til at strålerne bliver adskilt.

1.9 Afrunding

Jeg vælger at implementere en raytracer med følgende funktionalitet:

- Mulighed for at indlæse obj (*wavefront*) filer, således at komplekse scener kan indlæses automatisk.
- Brug af kD-træ for hurtig identifikation af relevante objekter. En effektiv datastruktur har afgørende betydning ved implementering af en hurtig raytracer. Den mest effektive er ifølge [Havran] kD-træet ved brug af *surface area huristics*. Jeg vil derfor implementere [Havran]'s, i pseudokode beskrevne, kD-træ.
- Mulighed for at gemme kD-træ i fil. Da det tager en rum tid at opbygge et effektivt kD-træ, vil jeg gøre det muligt at gemme det færdigbyggede kD-træ til en fil, således at det hurtigt siden hen kan indlæses.
- Mulighed for at indlæse kD-træ fra fil.
- Kan håndtere et ubegrænset antal lyskilder, - *spotlights*- såvel som *arealights*.
- Dynamisk til-/fraslutning af adaptiv supersampling. Da det ikke er altid supersampling er nødvendigt eller ønskværdigt, skal det være muligt at til og fraslutte supersamplingen, i form af et kommandolinieargument eller ved tryk på en tast under rendering.
- Mulighed for at navigere rundt i scenen (ved at flytte kameraet) i realtid ved brug af interpolation. Billedet vil straks vise et interpoleret billede som dannes ved at rendere et antal pixels i et grid og herefter interpolere mellem disse pixels. Dette billede skal gradvist forbedres når kameraet er stillestående, for efter få sekunder at opnå maksimal billedkvalitet.

Den raytracer jeg implementerer, er således forholdsvis simpel. En del af de begrænsninger jeg har foretaget er beskrevet i det følgende:

- Ingen *caustics*. *Caustics* nødvendiggøre brug af et *photon* kD-træ, som kræver en del ekstra prerendering samt en hel del ekstra belastning i selve renderingen. Der ligger desuden en hel del arbejde i selve implementeringen af *caustics* som ikke har direkte relevans for dette projekt.
- Ingen *colour bleading*. For at opnå denne effekt skal der kastes en hel del ekstra stråler, når en diffus overflade rammes, hvilket forøger antallet af sekundære stråler betydeligt. I forhold til hvad denne effekt har af betydning for billedkvaliteten, vurderes det ikke at være den ekstra renderingstid værd.
- Ingen bevægelige objekter. Det vil ikke være muligt at flytte rundt på objekter efter at kD-træet er bygget. Denne begrænsning er et direkte resultat af valget af datastruktur.
- Ingen brug af SIMD. Dette valg er foretaget pga. den meget komplekse og uigennemskuelige kode en SIMD venlig struktur medfører. For udenforstående (eksempelvis andre ønsker at fortsætte udviklingen af min raytracer hvor jeg slap) vil det være meget vanskeligt at forstå hvad der sker i koden. Desuden findes der en unik SIMD struktur til hver enkelt PC arkitektur, hvorfor implementering af SIMD vil sætte en stopper for platformuafhængigheden i projektet.
- Kun trekanter. Da de fleste ”scene filer” kun definerer trekanter, finder jeg det ikke umiddelbart betydningsfuldt at understøtte flere objekttyper. Desuden simplificerer det koden i flere sammenhænge. Bl.a. i forbindelse med at gemme og hente kD-træ’er til/fra fil.

2 Parallelisering

2.1 Introduktion

Parallelisering går ud på at skrive kode der kan udnytte flere CPU'er så programmet performer bedre. Det handler således om at lokalisere de beregningstunge dele af koden og distribuere disse dele til flere CPU'er, så koden kan afvikles parallelt (samtidigt).
Der er grundlæggende to former for parallelisering. Der er parallelisering målrettet til systemer der har flere CPU'er der deler den samme hukommelse (SMP systemer). Og så er der parallelisering målrettet til systemer der er sammensat af et antal computere over en form for netværk (DMPP). Desuden kan det lade sig gøre at lave kode der fungerer på begge systemtyper.
Før man går i gang med at parallelisere sin kode, bør man således overveje hvilket system koden skal afvikles på. Meget krævende programmer, som oftest programmer til udregning af meget komplekse problemer (så som klimaprognoser o.lign.), vil som regel blive udviklet til DMPP systemer, da det er her de største systemer findes (med mest regnekraft), da store SMP systemer er meget dyre at fremstille.
Mindre SMP systemer er til gengæld ved at blive ganske almindelige, da de førende chip-producenter har stort fokus på udvikling af flerkernede CPU'er. Dette betyder at det har stigende relevans at parallelisere programmer generelt.

API'er

Der findes mange forskellige API'er til parallelisering:

Pthreads, som er en lav-niveau API, der giver mulighed for en fleksibilitet som høj niveau API'er ikke kan måle sig med. Til gengæld skal tungen holdes lige i munden ved *low-level* parallelisering, så der ikke opstår *deadlocks* eller *race conditions* (se beskrivelse nedenfor).
MPI er et andet eksempel på en lav-niveau API, som er velegnet til SMP såvel som DMPP systemer. API'en kan siges primært at være et interface til kommunikationen mellem processerne. – Der er således funktioner til at sende og modtage data mellem processorne.

OpenMP fungerer kun på SMP systemer, men er umiddelbart klart lettere at have med at gøre, end Pthreads og MPI og giver desuden den fordel at koden stadig kan afvikles, selvom der ikke er support for OpenMP[12]. I stedet for at opbygge sit program direkte til API'en (som man skal til Pthreads og MPI), skal man til OpenMP blot specificere hvordan man ønsker koden paralleliseret. OpenMP understøtter desuden både Unix, Microsoft Windows m.fl.

Det kan være yderst vanskeligt at lave debugging af parallel kode og der kan desuden opstå problemer, som man ikke finder i sekventielt afviklet kode.

Race conditions:

Race condistions er en betegnelse for, hvad der sker, hvis to processor forsøger at ændre en variabel samtidigt med et uventet (fejlagtigt) resultat til følge. Et eksempel kunne være hvis to processor samtidigt forsøger at inkrementere et register i hukommelsen med værdien 0. Først læser de to processorer samtidig den oprindelige registerværdi, hernæst inkrementerer de den hver især og skriver den nye værdi til hukommelsen. Dette vil resultere i at resultatet bliver 1 i stedet for 2.
For at forhindre at *race conditions* opstår, må der laves et system til at låse kritiske sektioner i koden, så kun en proces af gangen kan skrive til delte variable og datatyper.

Deadlock:

Opstår hvis to (eller flere) tråde i et program venter på at den anden bliver færdig og dermed aldrig kommer videre.

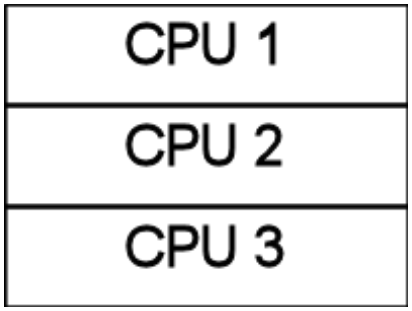
2.2 Strategier

Der er flere mulige indgangsvinkler til parallelisering af raytracer-koden. De mest åbenlyse er dog parallelisering af opbygningen af kD-træet samt af selve raytracingen, da disse er de to områder der virkelig tager tid. Opbygningen af kD-træet vil dog være vanskeligt at parallelisere pga. den rekursive opbygning, og vil kun komme til gavn i pre-renderingen, hvilket ikke har fokus i denne opgave. Selve raytracingen er til gengæld yderst relevant og der er desuden flere spændende strategier til, hvordan dette kan gøres.

Da en raytracer fungerer ved at "skyde" et stort antal **uafhængige** stråler ud i scenen, er parallelisering grundlæggende yderst trivielt. Skærmen kan

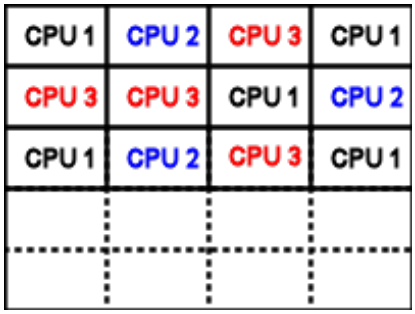
[12] Gælder kun hvis specifik OpenMP kode pakkes ind i #ifdef _OPENMP … #endif.

således blot deles op, så hver CPU, der er til rådighed, får ansvar for hver en blok af skærmen (illustreret i Figur 20).



Figur 20: Helt statisk opdeling af skærmen i 3 lige store dele.

Denne simple strategi vil virke, men det vil dog langt fra være optimalt. Problemet er, at de enkelte stråler langt fra bliver lige hurtigt afviklet. Hvor hurtigt en stråle afvikles afhænger af, om strålen rammer objekter i scenen og i så fald hvilke type overflader disse objekter har. Stråler der ingen objekter rammer, bliver naturligvis ganske hurtigt afviklet, hvorimod stråler der rammer eksempelvis højreflektive overflader kan være ganske beregningsmæssigt tunge, da de risikerer at bande mange gange rundt i scenen. Nogle områder af scenen vil således typisk tage langt længere tid at rendere end resten af scenen.
Den umiddelbart simpleste måde at imødekomme dette problem er blot at opdele skærmen i mindre blokstørrelser og distribuere disse over flere omgange til de enkelte CPU'er. De CPU'er der bliver hurtigt færdige vil således få tildelt en ny blok.



Figur 21: Dynamisk opdeling af skærmen i mange små dele. Når en CPU bliver færdig får den tildelt en ny blok.

Denne strategi er betydeligt bedre end den første. Problemet her er, at jo flere blokke skærmen opdeles i, jo mere *overhead*[13] vil der være. Det er

[13] Selvom det ikke er nødvendigt at synkronisere de forskellige tråde, vil der stadig være overhead i og med at det tager tid hver gang en ny tråd skal oprettes.

således en fin balance mellem at holde alle CPU'er konstant beskæftiget og at undgå for meget *overhead*.

Det optimale vil være, hvis man på forhånd ved, hvor lang tid de enkelte stråler skal bruge. Da man så kan fordele det arbejde der skal udføres, helt jævnt ud på de CPU'er der er til rådighed. Dette kan man naturligvis ikke vide. Til gengæld kan man starte med at foretage en indledende analyse af scenen, der kan udnyttes til en bedre distribuering af scenens pixels. Mere konkret kan der tages tid på, hvor hurtigt et antal tilfældige pixel-farver bliver beregnet. Andre pixels nær en af disse målte pixels vil med stor sandsynlighed være omtrent lige så længe om at blive beregnet, da stråler som *traces* fra de omkringliggende pixels, som oftest, vil ramme de samme overflader i scenen. Dette kan så bruges til at lave et kvalificeret bud på en arbejdsfordeling, hvor alle CPU'er bliver færdige samtidig.

I tilfælde af at der var mulighed for bevægelige objekter, ville det desuden være yderst relevant, at kigge på, hvordan man kunne genbruge dele af tidligere renderede *frames* og kun genrendere områderne omkring de bevægelige objekter. Men da den valgte spatiale datastruktur (kD-træet) ikke giver mulighed for bevægelige objekter, er disse overvejelser ikke relevante i denne sammenhæng.

2.3 To former for rendering

Jeg vil muliggøre to former for rendering. I mangel på bedre betegnelser kalder jeg disse for henholdsvis visuel og en ikke visuel rendering. Hvad dette præcist indbefatter, bliver beskrevet i det følgende. Hvilken renderingsform der skal benyttes, bestemmes ud fra en parameter i kommandolinien.

Uanset renderingsform skal billedet, hver gang det bliver helt færdigrenderet, gemmes til en fil, som navngives ud fra navnet på den scene der renderes efterfulgt af et nummer, der inkremeteres hver gang et nyt billede dannes. På den måde vil der automatisk blive dannet en billedserie, når man bevæger sig rundt i scenen.

Den første renderingsform skal baseres udelukkende på brug af OpenMP. Løsningen er ganske simpel. Jeg paralleliserer blot de to *for-loops* som *tracer* skærmens pixels. Som *schedule* vil jeg blot benytte *runtime*, således at brugeren, ved brug af kommandolinie parametre, kan bestemme hvilken form for *scheduling* der skal benyttes, ligesom der også kan angives det antal tråde der skal anvendes. Jeg vil siden hen lave forskellige testscenarier (se

test afsnit), der skal hjælpe til at fastslå, hvor mange tråde der er nyttige, samt hvilken *schedule* type der er den mest effektive.

```
#pragma omp parallel for default(none) private(x,y) shared(canvas,engine,width,height) schedule(runtime)
    for(x=0; x<width; x++)
        for(y=0; y<height; y++)
            canvas->draw_pixel(x,y,engine->TracePixel(x,y));
```

Figur 22: Kodeudsnit der paralleliserer renderingen af skærmens pixels ved brug af OpenMP.

Den anden renderingsform (visual rendering), går ud på at sikre, at det er muligt, at navigere ubesværet rundt i scenen, selv ved lave framerates. Dette vil ikke fungere godt ved den rene OpenMP løsning, da OpenMP vil rendere billedet helt færdigt før der reageres på eventuelle brugerinputs. Dette kan betyde store forsinkelser fra at brugeren f.eks. flytter kameraet og til at billedet opdateres.
Visuel rendering går således ud på, at lave en mere dynamisk renderingsform der garanterer en umiddelbar reaktion på brugerinput, men stadig ender op i et højkvalitets billedoutput.
Dette skal opnås ved en gradvis billedforbedring. Der *traces* således til at starte med kun ganske få skærmpixels, hvorefter der interpoleres farver for de mellemliggende pixels, for at opnå et fuldt farvelagt billede. På denne måde kan der meget hurtigt tegnes et noget uklart (udetaljeret), men dog genkendeligt, billede på skærmen. Der vil, med andre ord, med det samme ske noget, når brugeren navigerer rundt i scenen. Når brugeren til gengæld ikke navigerer, vil billedet gradvist forbedres, indtil den maksimale kvalitet er opnået (se Figur 23).

Figur 23: Illustrerer hvordan scenen renderes trin for trin og interpolerer mellem de renderede pixels.

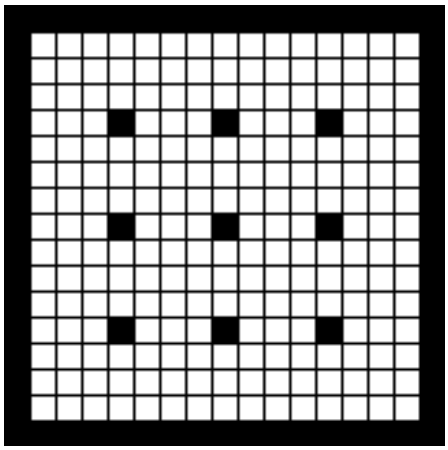
Rendering af et billede består af følgende trin:

Først *traces* pixels i et groft grid (hver 32. pixel *traces*), hvor der tages tid på hvor længe de enkelte pixels er om at blive færdige. De resulterende pixel-farver gemmes i en *canvas*-buffer og *trace*-tiden gemmes i en *cost*-buffer.

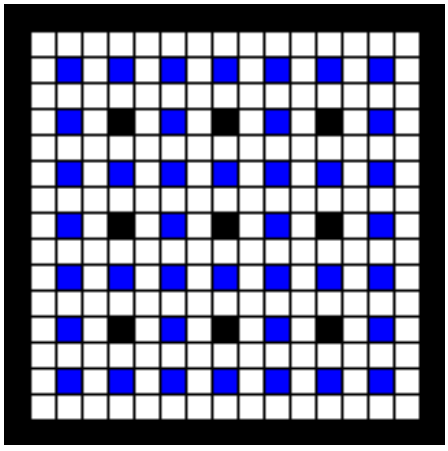
Herefter interpoleres der farveværdier for alle de ikke *tracede* pixels ud fra de *tracede* pixels i *canvas*-bufferen. Disse interpolerede pixels skrives direkte til canvas, således at alle skærmens pixels får en farve.
Cost-bufferen interpoleres, sådan at der dannes et estimat af, hvor lang tid det tager at *trace* hver enkelt pixel. Dette estimat bruges efterfølgende til at

uddelegere tidsmæssigt lige store dele af skærmens pixels til de enkelte CPU'er. Mere om dette senere.

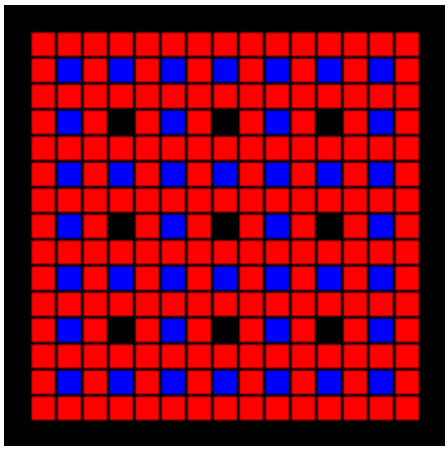
Efter at det første grid er renderet og *cost*-bufferen dannet, så oprettes en tråd for hver halvering af blokstørrelsen. Der oprettes således en tråd med blokstørrelse på 16, 8, 4, 2 og 1. Hver tråd *tracer* så pixels der bestemmes ud fra blokstørrelsen og gemmer disse pixel-farver i hver sin separate *canvas*-buffer. For at eksempelvis tråden der *tracer* pixels ud fra en blokstørrelse på 2 ikke skal spilde tid på at *trace* de pixels, som de andre tråde allerede har bestemt, *tracer* tråden skiftevis pixelrækker med "blokstørrelse" mellemrum og "2 x blokstørrelse" mellemrum. Dette illustreres på Figur 24, Figur 25 og Figur 26.



Figur 24: Startsituationen for tråden med blokstørrelse 2. De farvede pixels skal naturligvis ikke farves på ny.



Figur 25: De blå firkanter viser de pixels som tråden med blokstørrelse 2 *tracer*.



Figur 26: Her vises det hvilke pixels den sidste tråd (med blokstørrelse 1) tracer. – Alle de røde pixels.

Når en tråd er færdig, opdaterer den sin *canvas*-buffer med pixel-farver fra *canvas*-bufferen med et niveau grovere grid end den pågældende tråd, hvorefter *canvas* opdateres ud fra interpolerede farver fra den pågældende *canvas*-buffer.

Med ovenstående arbejdsfordeling er det klart at de forskellige tråde vil blive færdige på helt forskellige tidspunkter. Den tråd der står for at *trace* et grid med blokstørrelse 16 vil være langt hurtigere færdig end den der laver et grid med blokstørrelse 1. Her er det at *cost*-bufferen kommer ind i billedet. Ved hjælp af denne, kan jeg lave en effektiv fordeling af arbejdet fra de tætteste grids ud på flere tråde. Se et eksempel på effekten af dette på Figur 32 og Figur 33.

Rent praktisk vælger jeg som udgangspunkt at gøre brug af følgende tråde i den visuelle implementering:

- En hovedtråd (*main*) der står for at al initialisering samt for håndtering af OpenGL. OpenGL får specificeret en *idle* funktion der sørger for at opdatere skærmen ud fra en bitmap i *canvas* (billedet opfattes af OpenGL som en textur der klistres på en firkant der fylder hele skærmen). *Idle* funktionen sørger desuden for at lave screendump af billedet, hver gang det er færdigrenderet. Se JSP diagram af *idle* tråden på Figur 53 i Appendiks A.
- En tråd man kunne kalde *scheduler* (funktionsnavnet er `MultithreadRendering2` i koden), der først *tracer* den det første grid og laver den omtalte *cost-buffer*. Ud fra *cost-bufferen* oprettes så de forskellige "arbejdertråde" som står for rendering af de resterende grids (se JSP diagram over *scheduler* tråden på Figur 55 i Appendiks

A). Der oprettes således en eller flere tråde for hver blokstørrelse. Hvilke gridstørrelser der skal deles ud på flere tråde afgøres i afsnittet ”4.2 Opdeling af trådene”. Opdeling af trådene sker med funktionen *MakeThreads*:

```
extern "C" void MakeThreads(int startThreadNo, int splitNo, int bSize,
int bufID)
{
  int width=canvas->get_width(), height=canvas->get_height();
  //find optimal split pos
  double costPart = renderInfo.totalCost / splitNo;
  double tmpCost;
  int i, lastSplitPos = bSize, splitPos, tmpSplit, tmpLastSplit =
bSize;
  for(i = startThreadNo; i<startThreadNo+splitNo-1; i++)
  {
     tmpCost = 0;
     for(splitPos=lastSplitPos; splitPos<width*height; splitPos++)
      if((tmpCost +=
renderInfo.costBuffer[(int)splitPos%width][(int)splitPos/height]) >
costPart)
            break;//break when enough pixels has been assigned
     tmpSplit = splitPos / height;//do only distribute complete lines
     tmpSplit -= ((tmpSplit % (2*bSize))+bSize);//make sure startpixel
is a part of the grid to be rendered
     //create thread
     pthread_create(&thread[i],NULL,TightenGrid,new ThreadParameters(
tmpLastSplit,tmpSplit,bSize, bufID, false, false));

     tmpLastSplit = tmpSplit;
     lastSplitPos = splitPos;
  }
  //create last thread
  pthread_create(&thread[i],NULL,TightenGrid,new ThreadParameters(
tmpLastSplit,width-1,bSize, bufID, true, false));
}
```

Figur 27: Kode for funktionen MakeThreads. Funktionen opretter et antal tråde specificeret med parametren splitNo. Hver tråd oprettes med funktionen pthread_create. Det sidste argument i pthread_create definerer det arbejde som tråden skal udføre (i form af en pointer til en instans af klassen ThreadParameters).

- Mindst 5 ”arbejder tråde” der renderer et grid specificeret af *schedular* tråden (se JSP diagram for denne tråd på Figur 56 i Appendiks A):

```
extern "C" void *TightenGrid(void *par)//worker thread
{
  int x,y,h;
  int width = canvas->get_width();
  int height = canvas->get_height();
  //get work instructions with static cast from void*
  ThreadParameters* p = static_cast<ThreadParameters*>(par);
```

```
  //Trace new grid-points
  for(x=p->bSize,h=0; x<width-p->bSize; x+=p->bSize,h++)
    for(y=p->from; y<p->to; y+=(h%2==0 ? p->bSize : 2*p->bSize))
      renderInfo.canvasBuffer[p->bufPtr][x][y] = TracePixel(x, y);

  if(p->doUpdate)//update higher order buffers with newly rendered grid
points
  {
    //Wait for "lower" thread(s) to finish
    pthread_mutex_lock(&mut);
    while(renderInfo.bufReady < p->bufPtr-1)
    {
      if(redraw) {pthread_mutex_unlock(&mut); return NULL;}//exit if
redraw
      pthread_cond_wait(&cond, &mut);
    }
    //Join with previous grid-points
    if(p->bufPtr > 1)
      for(x=0; x<width; x++)
        for(y=0; y<height; y++)
          if((renderInfo.canvasBuffer[p->bufPtr-1][x][y][0] != 1.0))
            renderInfo.canvasBuffer[p->bufPtr][x][y] =
renderInfo.canvasBuffer[p->bufPtr-1][x][y];

    //broadcast that bufReady has been updated
    renderInfo.bufReady = p->bufPtr;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mut);
  }

  return NULL;
}
```

Figur 28: Kode for funktionen TightenGrid. Funktionen *tracer* et antal pixels specificeret i *p*, hvorefter der ventes på at alle arbejdertrådene med mindre *bufPtr* bliver færdige, hvorefter *canvas-bufferen* opdateres med *canvas-bufferen* for *bufPtr*-1. Til sidst laves der et *broadcast*, så evt. ventende tråde får besked om at *bufReady* er blevet opdateret.

- En tråd der står for at skrive interpolere farver fra *canvas-bufferen* til *canvas*, hver gang en ”arbejdertråd” bliver færdig (se JSP diagram over denne tråd på Figur 54 i Appendiks A):

```
extern "C" void *UpdateCanvas(void *X)
{
  int lastBuf = -1, x, y, width = canvas->get_width(), height =
canvas->get_height(), bSize;
  while(true)
  {
    //wait for a worker-thread to finish
    pthread_mutex_lock(&mut);
    while(renderInfo.bufReady == lastBuf)
```

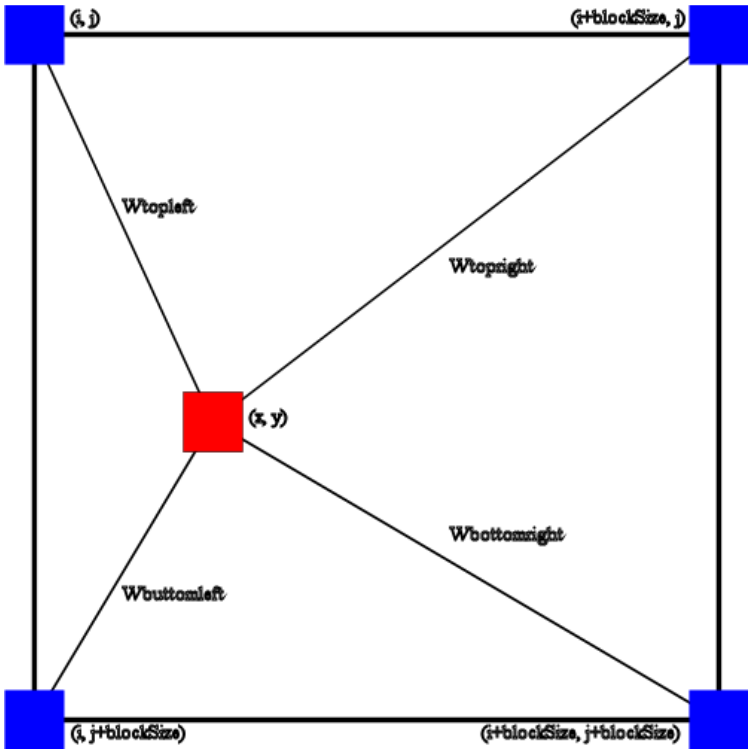
```
pthread_cond_wait(&cond, &mut);
     lastBuf = renderInfo.bufReady;
     //find bSize from bufferid
     switch(lastBuf){case 0: bSize = 32; break; case 1: bSize = 16;
break; case 2: bSize = 8; break; case 3: bSize = 4; break; case 4:
bSize = 2; break; case 5: bSize = 1; break;}
     pthread_mutex_lock(&canvasMutex);
     //Interpolate if needed
     if(lastBuf < 5)
        for(x=0; x<width-bSize; x+=bSize)
           for(y=0; y<width-bSize; y+=bSize)
                InterpolateBuffer(x, y, bSize, lastBuf);
     else
        for(x=0; x<width; x++)
           for(y=0; y<height; y++)
                canvas-
>draw_pixel(x,y,renderInfo.canvasBuffer[lastBuf][x][y]);
     //tell canvas thread to update
     drawCanvas = true;
     pthread_mutex_unlock(&canvasMutex);
     pthread_mutex_unlock(&mut);
   }
   return NULL;
}
```

Figur 29: Kode for funktionen UpdateCanvas. Funktionen startes som en selvstændig tråd og kører i et uendeligt loop (while(true)). Tråden venter på at der bliver *broadcasted* et signal om at *bufReady* er blevet opdateret, hvorefter tråden interpolerer værdier for ikke renderede pixels og giver besked til *idle* funktioen om at *canvas* skal opdateres (ved at sætte *drawCanvas = true*).

2.4 Interpolation

Til at bestemme pixelværdierne i mellem fire kendte pixelværdier med *blockSize* mellemrum, skal jeg definere en vægtfunktion, således at de interpolerede pixels får et vægtet farvebidrag fra hver af de fire kendte pixels. De pixels der ligger tættest på den interpolerede pixel skal vægte mest og den længst fra mindst. Hver kendt pixel skal således have sin egen vægt. Se Figur 30

Vægtværdien for pixels interpoleret oven i den tilhørende pixel skal have maksimalt bidrag. Bidraget skal aftage lineart med længden fra den pågældende pixel indtil den slet ikke bidrager, hvilket skal ske når længden er lig længden til den diagonalt modsatte pixel.



Figur 30: Billedet viser hvordan en ny pixel (rød) skal bestemmes ud et vægtet bidrag fra de fire kendte pixels (blå) ved interpolation.

Vægtene bygges op hver ud fra to ligninger, ved brug af Pythagoras sætning, af en vægt-værdi for x-retningen og en vægt-værdi for y-retningen der hver giver en værdi mellem 0 og 1. Disse to vægt-ligninger har følgende form (ligninger af første grad):

$$W_x = ax + b \tag{2.4.1}$$

og

$$W_y = ay + b \tag{2.4.2}$$

I x-retningen skal ligningen give 1 når x er lig den kendte pixel-værdis x-koordinat og 0 når x er lig pixelen der ligger *blockSize* væk langs x-aksen. Disse to kendte tilstande resulterer i følgende to ligninger:

$$0 = a(i + blockSize) + b \tag{2.4.3}$$

og

$$1 = ai + b$$

$$\Leftrightarrow$$

$$b = 1 - ai \tag{2.4.4}$$

a kan bestemmes ved substitution af ligning 2.4.4 med ligning 2.4.3:

$$0 = a(i + blockSize) + 1 - ai$$
$$\Leftrightarrow$$
$$a = -\frac{1}{blockSize} \quad \textbf{(2.4.5)}$$

Nu kan b bestemmes ved substitution af ligning 2.4.5 med ligning 2.4.4:

$$b = 1 - \frac{-1}{blockSize} i$$
$$\Leftrightarrow$$
$$b = \frac{blockSize + i}{blockSize} \quad \textbf{(2.4.6)}$$

Vi får således, ved at substituere ligning 2.4.5 og 2.4.6 med 2.4.1, at:

$$W_x = -\frac{1}{blockSize} x + \frac{blockSize + 1}{blockSize}$$
$$\Leftrightarrow$$
$$W_x = \frac{blockSize + 1 - x}{blockSize} \quad \textbf{(2.4.7)}$$

W_y kan bestemmes på samme vis. Resultatet bliver følgende:

$$W_y = \frac{blockSize + 1 - y}{blockSize} \quad \textbf{(2.4.8)}$$

De samlede vægte kan nu beregnes med Pythagoras sætning:

$$W_{TopLeft} = \sqrt{W_x^2 + W_y^2}$$
$$W_{TopRight} = \sqrt{(1 - W_x)^2 + W_y^2}$$
$$W_{BottomLeft} = \sqrt{W_x^2 + (1 - W_y)^2} \quad \textbf{(2.4.9)}$$
$$W_{BottomRight} = \sqrt{(1 - W_x)^2 + (1 - W_y)^2}$$

For stadig at holde vægtene mellem 0 og 1 normaliseres de:

$$
\begin{aligned}
W_{TopLeft} &= \frac{W_{TopLeft}}{W_{TopLeft} + W_{TopRight} + W_{BottomLeft} + W_{BottomRight}} \\
W_{TopRight} &= \frac{W_{TopRight}}{W_{TopLeft} + W_{TopRight} + W_{BottomLeft} + W_{BottomRight}} \\
W_{BottomLeft} &= \frac{W_{BottomLeft}}{W_{TopLeft} + W_{TopRight} + W_{BottomLeft} + W_{BottomRight}} \\
W_{BottomRight} &= \frac{W_{BottomRight}}{W_{TopLeft} + W_{TopRight} + W_{BottomLeft} + W_{BottomRight}}
\end{aligned}
\qquad \textbf{(2.4.10)}
$$

En vilkårlig pixelværdi mellem de fire kendte pixels kan nu beregnes:

$$
\begin{aligned}
Pixel(x,y) &= W_{TopLeft} Pixel(i,j) \\
&+ W_{TopRight} Pixel(i + blockSize, j) \\
&+ W_{BottomLeft} Pixel(i, j + blockSize) \\
&+ W_{BottomRight} Pixel(i + blockSize, j + blockSize)
\end{aligned}
\qquad \textbf{(2.4.11)}
$$

2.5 Afrunding

Jeg har valgt udelukkende at parallelisere selve *run-time* afviklingen, da fokus i dette projekt ligge på at få netop denne del til at forløbe hurtigst muligt.
OpenMP og Pthreads benyttes til parallelisering. OpenMP benyttes hvor det er muligt/givtigt og Pthreads benyttes i mere komplekse situationer som OpenMP ikke understøtter.
Ved at lave gradvis rendering af billedet, ved brug af interpolation, gøres det muligt at navigere rundt i scenen (ved at flytte kameraet) i realtid.

3 Interaktivitet

En naturlig følge af, eller et direkte mål, ved at implementere en realtids raytracer er at kunne interagere i scenen. Man kan forestille sig mange interessante former for interaktivitet:

1. Den mest simple form for interaktion er at gøre det muligt at bevæge kameraet rundt i scenen. Dette vil være yderst praktisk, da det gør det muligt at se scenen fra alle sider og vinkler, samt tæt på såvel som langt fra.
2. At kunne indsætte og fjerne lyskilder fra scenen kunne også være praktisk. Dette vil f.eks. kunne anvendes til at oplyse dele af scenen der som udgangspunkt ligger i mørke.
3. I visse sammenhænge vil det være yderst anvendeligt at kunne klippe i objekterne i scenen. Man kunne f.eks. forestille sig en avanceret 3D model, hvor kun en lille del af denne er synlig set udefra. Det ville her være veldigt praktisk hvis man kunne ”skære” i objekterne, for på den måde at gøre det muligt at se hvordan objektet ser ud inden i.
4. Hvis raytraceren skal bruges i animationssammenhæng, vil det være nødvendigt at kunne definere bevægelige objekter.
5. En anden interessant indgangsvinkel ville være at anvende raytraceren til udvikling af 3D objekter i realtid. Det ville være ganske spændende at kunne bevæge sig rundt i en 3D og tegne objekter, hvor man med det samme så et realistisk resultat, med skygger, refleksion, transmission, *colourbleeding, caustics*, osv.

Disse tanker om interaktion med scenen synliggør en del af de mange muligheder, en raytracer der kan rendere i realtid giver. Desværre så viser det sig, at der i praksis, er en hel del forhindringer, der skal overvindes, førend at disse tanker kan virkeliggøres.
Med den valgte kD-træ datastruktur, er det kun punkt 1 og 2 der er praktisk muligt. Dette kommer sig af, at det er nødvendigt at genopbygge kD-træet, hver gang et objekt fjernes fra eller indsættes i scenen, - hvilket tager meget lang tid. Der skal, som tidligere nævnt, en hel del ”prerendering” til før realtidsrenderingen kan påbegyndes.

For nylig er der dog blevet udviklet kD-træer der rent faktisk gør det muligt at "genopbygge" træet i realtid. Disse træer muliggør realisering af alle de nævnte interaktions muligheder, hvilket puster endnu mere liv i den stigende interesse for realtids raytracere.
Da jeg først læste om disse nye kD-træer ganske sent i dette projektforløb, er jeg nødsaget til at holde mig til det valgte kD-træ, og få det bedste ud af dette. I øvrigt ligger det primære fokus i dette projekt på, hvilke muligheder der er ved parallelisering af raytracere på SMP systemer og i mindre grad om funktionaliteten af raytraceren eller den praktiske anvendelse af denne. Uanset hvor spænde disse emner end måtte være!
Jeg er således begrænset til punkt 1 og 2.

4 Resultater

4.1 Introduktion

Dette kapitel fokuserer udelukkende på parallelisering af raytraceren.

Alle tests er udført ved brug af programmet *Sun Studio Performance Analyzer*. Primært vil jeg kigge på *Analyzer*'ens *timeline*. Denne viser hvordan de forskellige tråde i programmet arbejder. Øverst vil der være en *timeline* der viser den forløbne tid i sekunder. Baren under *timeline* angiver hvor stor del af den samlede CPU kraft programmet benytter. Dernæst er der et antal rækker kaldet 1.x, hvor x angiver et trådnummer. Det er disse rækker der er de rigtig interessante. Til højre for hver af disse har jeg angivet et navn for den respektive tråd. Den første er *Main*, som er hovedtråden der foretager al initialisering og prerendering. *Main* vil desuden starter de to efterfølgende tråde, - *Scheduler* og *UpdateCanvas* og dernæst stå for at opdatere *canvas* ud fra en bitmap-textur (hvilket foregår i *idle* funktionen). *Scheduler* tråden står primært for at oprette de resterende tråde, der vil blive omtalt som arbejdertråde. *UpdateCanvas* sørger for at interpolere farve-værdier for ikke renderede pixels ud fra de forskellige arbejdertrådes buffer og danne bitmap-texturen. Arbejdertrådene foretager selve raytracingen. De renderer hver et grid af forskellig grovhed. Gridstørrelsen for de enkelte arbejdertråde er angivet ud for hver af arbejdertrådene. Når der er flere arbejdertråde der er angivet til at rendere samme gridstørrelse, betyder det, at de pixels der skal renderes for den pågældende gridstørrelse er blevet delt ud på flere tråde. Se i øvrigt afsnittet "2.3 To former for rendering" for yderligere beskrivelse af de forskellige tråde eller Appendiks A, hvor der er JSP diagrammer for trådene.

I alle tests, på nær i testen "4.4 Ikke nok CPU'er", benyttes serveren Bohr (består af 48 1200MHz *dual-core* UltraSPARC CPU'er), som har nok CPU'er til at alle tråde kan afvikles samtidigt, til rendering af scenerne. Der holdes desuden øje med at Bohr ikke er overbelastet, da det kan give et forkert billede, hvis flere tråde må deles om én CPU.
Testen "4.4 Ikke nok CPU'er" bliver udført på serveren Sunq05 der består af 4 1062MHz UltraSPARC IIIi CPU'er.

For samtlige *timeline screendumps* (fra *Analyzer*) gælder det, at der går ca. 1 sekund, før programmets tråde bliver startet (se eksempelvis Figur 32). På dette sekund foregår al initialisering samt prerendering. Hoveddelen af prerenderingen ligger i indlæsning af kD-træet fra en fil. Denne del af *timelinen* vil ikke blive kommenteret yderligere, da den ikke har indflydelse på realtidsafviklingen af raytraceren.

Jeg vil benytte følgende formler til henholdsvis beregning af *speedup* og effektivitet:

$$S(P) = T(1) / T(P)$$
$$E(P) = S(P) / P$$

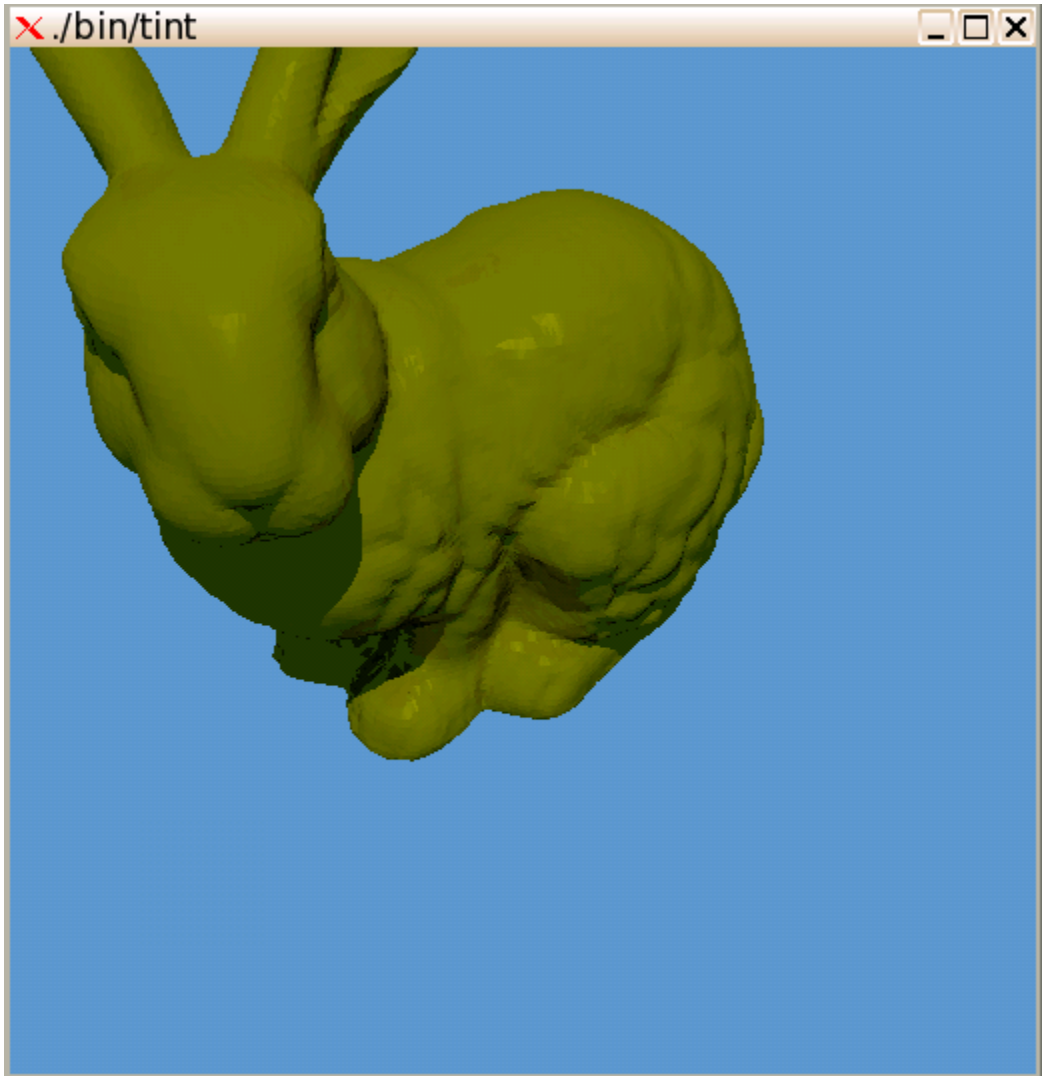
Hvor P er antallet af processer/tråde, T(P) er eksekveringstiden for P processer, S(P) er *speedup* for P processer og E(P) er effektiviteten for P processer.

4.2 Opdeling af trådene

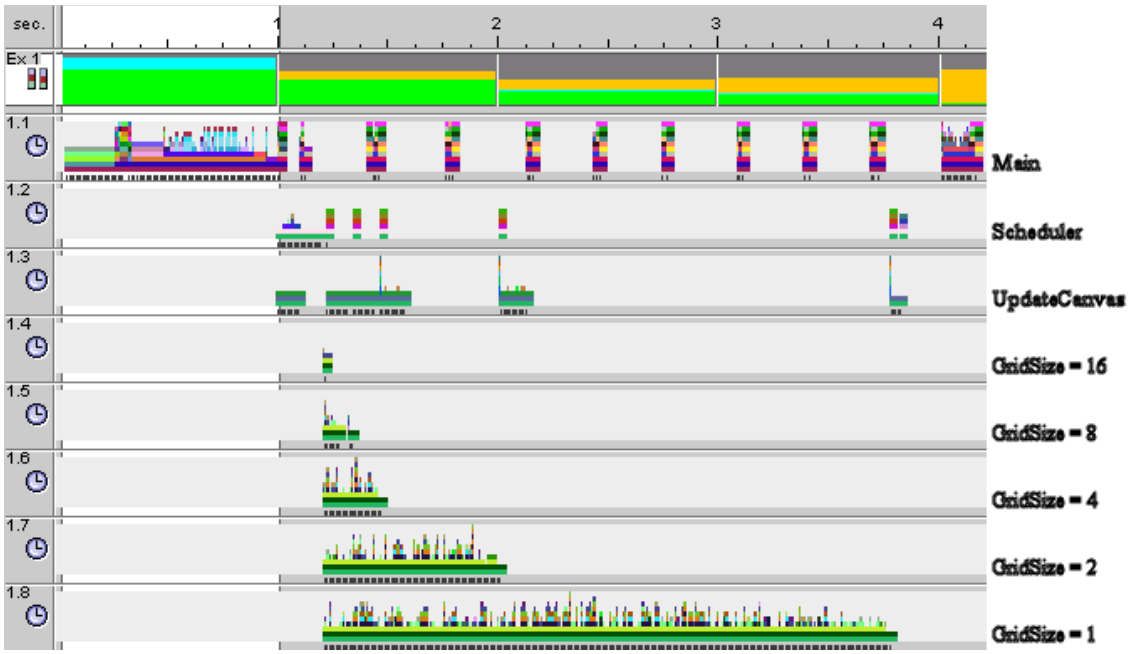
4.2.1 Formål

At undersøge hvordan det vil være effektivt at opdele arbejdertrådene. Målet er at få alle arbejdertrådene til at afslutter nogenlunde samtidig.

4.2.2 Resultat



Figur 31: Output efter rendering af bunny.kdtree uden supersampling. Scenen består af 69.451 trekanter og kD-træet er bygget af 262.384 noder. Trekanterne har følgende overfladespecifikationer: k_highlight = 0.3, phong_exponent = 80, k_reflected = 0.3, k_diffuse = 0.7.



Figur 32: Screendump af Analyzer efter rendering af kaninbilledet, der tydeligt viser den skæve arbejdsfordeling trådene i mellem.

Som de ses på Figur 32 er der en meget skæv arbejdsfordeling trådene i mellem. Samlet set så afvikles arbejdertrådene (1.4-1.8) på ca. 3,7 sekunder. Som beskrevet i afsnit ”2.3 To former for rendering” så var det ganske forventet, at de sidste to arbejdertråde (1.7 og 1.8) har væsentligt mere at lave end de 3 første (1.4-1.6).

På ses det at Figur 46 ses det at figuren kan renderes med en tråd på ca. 3,3 sekunder. Ved brug af 5 tråde opnås et *speedup* på:

$$S(5) = T(1) / T(5)$$
$$\Rightarrow$$
$$S(5) = 3,3 / 2,6 = 127\%$$

Hvilket resulterer i en effektivitet på:

$$E(5) = S(5) / 5$$
$$\Rightarrow$$
$$E(5) = 1,27 / 5 = 24\%$$

For at forbedre *speedup* såvel som effektiviteten vurderes det ud fra Figur 32, at det vil være rimeligt at distribuere arbejdet fra tråd 1.7 på 3 tråde og arbejdet fra 1.8 på 5 tråde.

Figur 33 viser resultatet af denne opdeling.

Figur 33: Screendump af Analyzer efter rendering af kaninbilledet, der viser den væsentligt bedre arbejdsfordeling i mellem trådene efter at de to sidste tråde fra Figur 32 er blevet delt op i henholdsvis 3 og 5 tråde.

Det er tydeligt at der vindes ganske meget ved den bedre fordeling af arbejdet mellem trådene. Selve afviklingen af arbejdstrådene går fra at tage små 3 sekunder til ca. 1 sekund. Der opnås således et *speed-up* på:

$$S(11) = T(1) / T(11)$$
$$\Rightarrow$$
$$S(11) = 3{,}3 / 1 = 330\%$$

Hvilket resulterer i en effektivitet på:

E(11) = S(11) / 11
=>
E(11) = 3,3 / 11 = 30%

Beregning viser at løsningen ved brug af 11 tråde giver et ganske pænt *speedup*. Effektiviteten er dog stadig ganske lav. Dette er dog ikke overraskende da de første arbejdertråde kun er aktive i ganske kort tid før de afsluttes og frigives. Den tid der går før de øvrige tråde er færdige indgår således i beregningen, selvom CPU'erne reelt er gjort ledige og således kan sættes til at udføre andre processer.

Umiddelbart vil det kunne være givtigt at distribuere tråd 1.8 fra Figur 32 ud på yderligere et par tråde. Dog vælger jeg at holde mig til disse 11 arbejdertråde, da der ved brug af endnu flere tråde let kan opstå problemer med at Bohr ikke har det fornødne antal ledige CPU'er, hvilket kan forstyrre de efterfølgende tests.

4.2.3 Opsummering

Det besluttes at raytracingen af gridstørrelse 2 bliver distribueret på 3 tråde og raytracingen af gridstørrelse 1 på 5 tråde. Dette gøres da der opnås en ganske pæn performance forbedring. Trådene opdeles ikke yderligere for at undgå problemer med for få ledige CPU'er på Bohr, hvilket vil forstyrre ensartetheden i de øvrige tests.

4.3 Fordeling af arbejdsbyrden

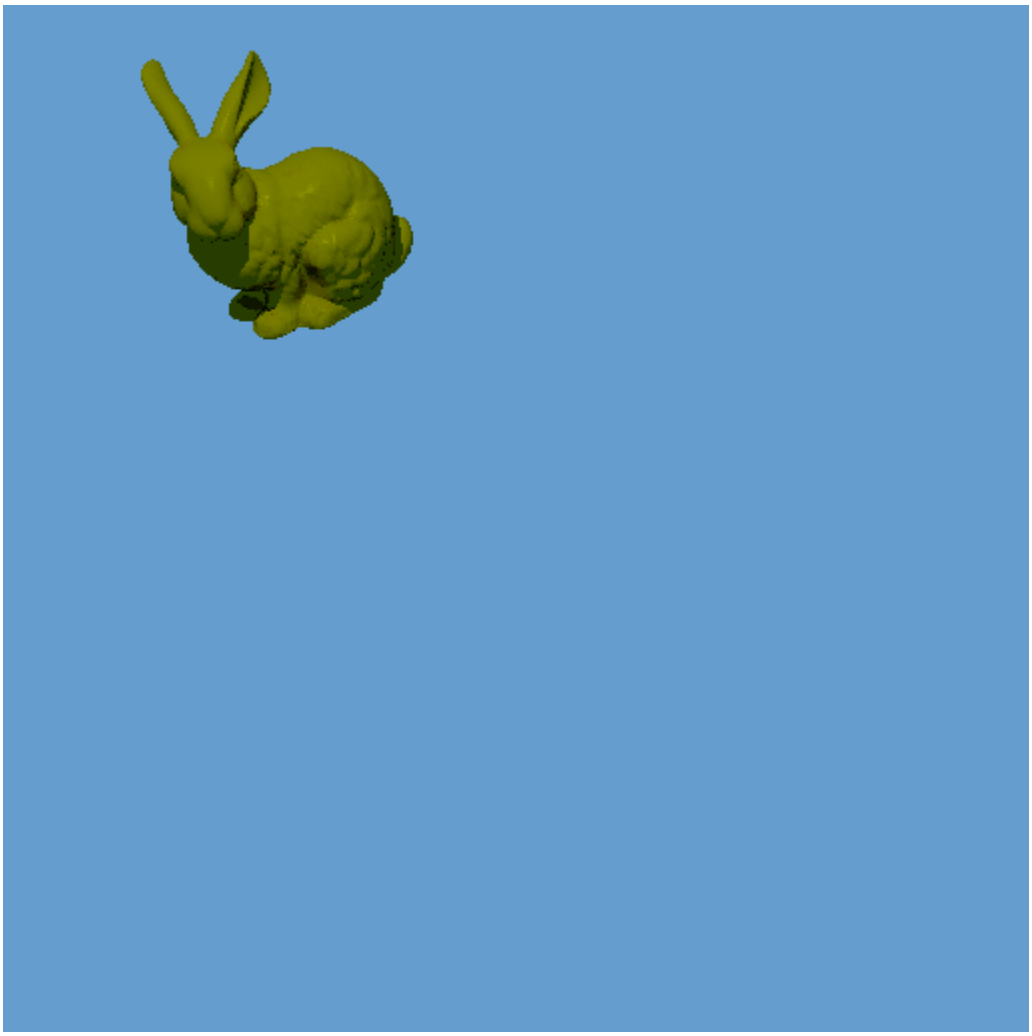
4.3.1 Formål

At vurdere hvor velegnet den interpolerede *cost-buffer* er som udgangspunkt for opdeling af de "tunge" tråde. Dette gøres ved at analysere *timelines* fra rendering af forskellige figurer og figurstørrelser. En god opdeling kan ses på at trådene med samme grid-størrelse slutter samtidigt. Hvis dette ikke er tilfældet skyldes det at der ikke blev brugt nok pixel-målinger til dannelse af *cost-bufferen*. Da der er benyttet interpolering, er *cost-bufferen* kun et estimat for det reelle *cost*. Jo flere pixels der benyttes til dannelse af *cost-bufferen* jo bedre vil den således svare til det reelle *cost* og jo bedre vil opdeling af trådene blive.

Testen udføres ved analyse af 4 forskellige figurtyper i programmet A*nalyzer*. I programmet analyseres hvordan de forskellige tråde arbejder, med fokus på om de sidste to grid-størrelser 2 og 1, som er distribueret på henholdsvis 3 og 5 tråde, afsluttes samtidigt.

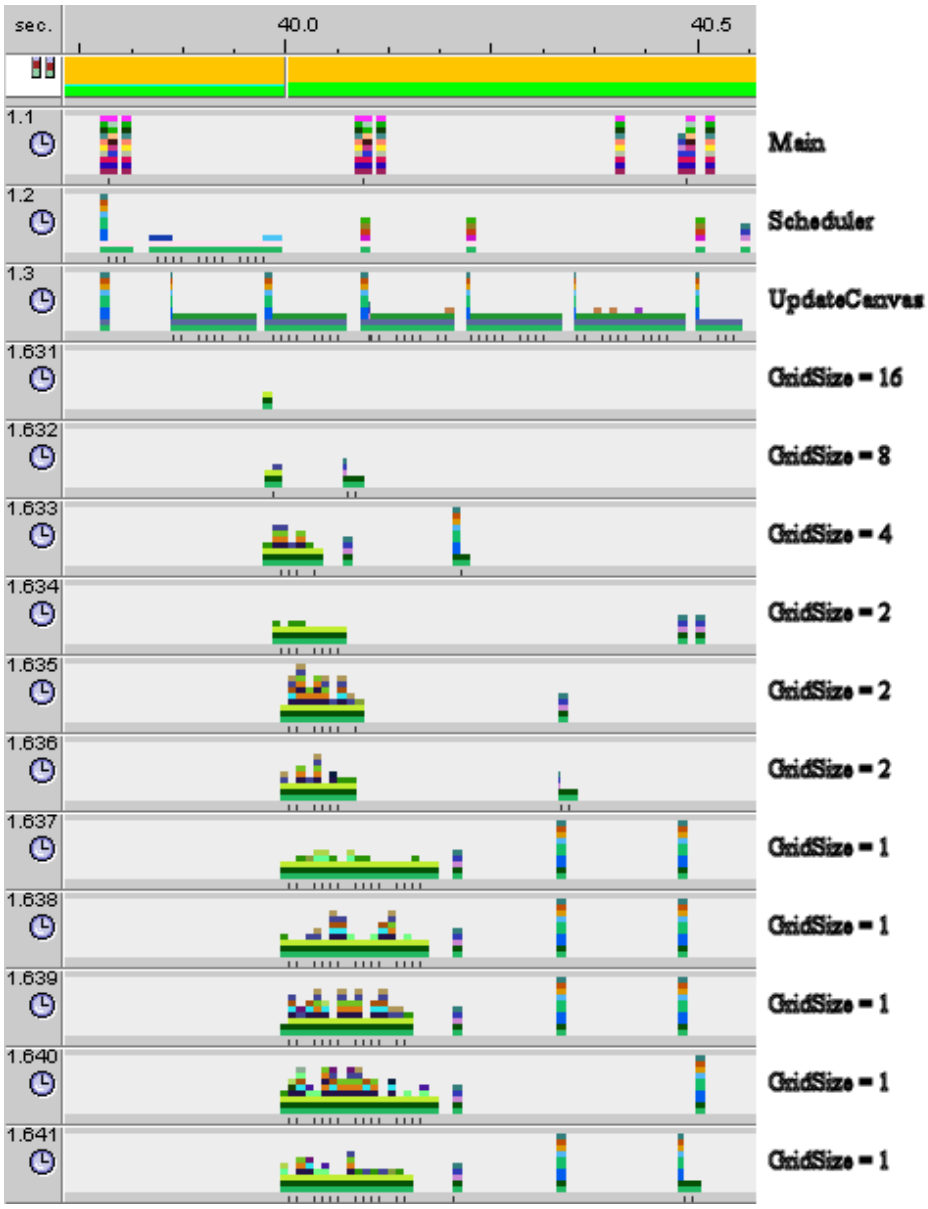
4.3.2 Resultat

Lille bunny



Figur 34: Output efter rendering af bunny.kdtree, hvor der er zoomed ud og navigeret rundt så figuren er placeret i hjørnet.

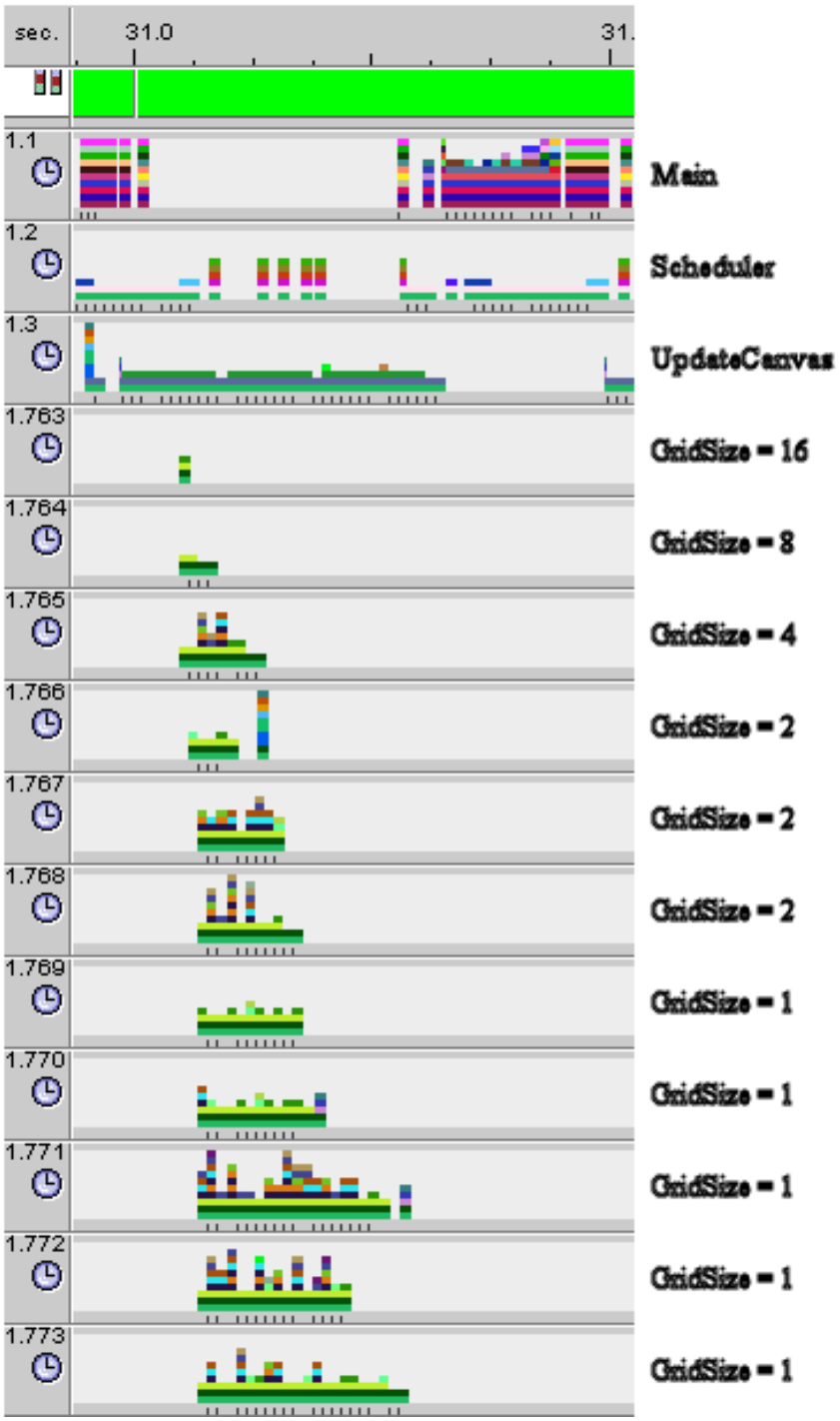
Det forventes at denne figur bliver renderet betydeligt hurtigere end den store bunny på Figur 30, da der kun vil blive udført stråle-trekantskæringsberegninger for de stråler der kommer ganske tæt på figuren (inden for figurens *bounding box*).



Figur 35: ***Timeline screendump*** **der viser renderingsforløbet af den lille Bunny.**

Figuren blev som forventet renderet noget hurtigere end den store bunny blev det (omkring dobbelt så hurtigt), men ikke helt så hurtigt som forventet. Den lidt langsommere renderingstid end forventet satte mig på sporet af et problem ved rendering af små figurer. Problemet ligger i at *UpdateCanvas* tråden låser *canvas-bufferen* imens der interpoleres, hvilket betyder at trådene ikke kan lave den sidste opdatering af *canvas-bufferen* og afslutte. Dette betyder at der går over 0,5 sekunder før billedet bliver færdigrenderet, selvom trådene reelt har renderet alle pixels allerede efter godt 0,2 sekunder. For at løse dette problem laver jeg en lille ændring af *UpdateCanvas* funktionen. I stedet for at interpolere direkte ud fra *canvas-bufferen*, starter jeg med at kopiere *canvas-bufferen* over i et andet array. På den måde kan

jeg nøjes med at låse *canvas-bufferen* imens bufferen kopieres (hvilket går betydeligt hurtigere end interpoleringen). Resultatet ses på nedenstående figur (Figur 36):



Figur 36: ***Timeline screendump*** **der viser renderingsforløbet af den lille Bunny.**

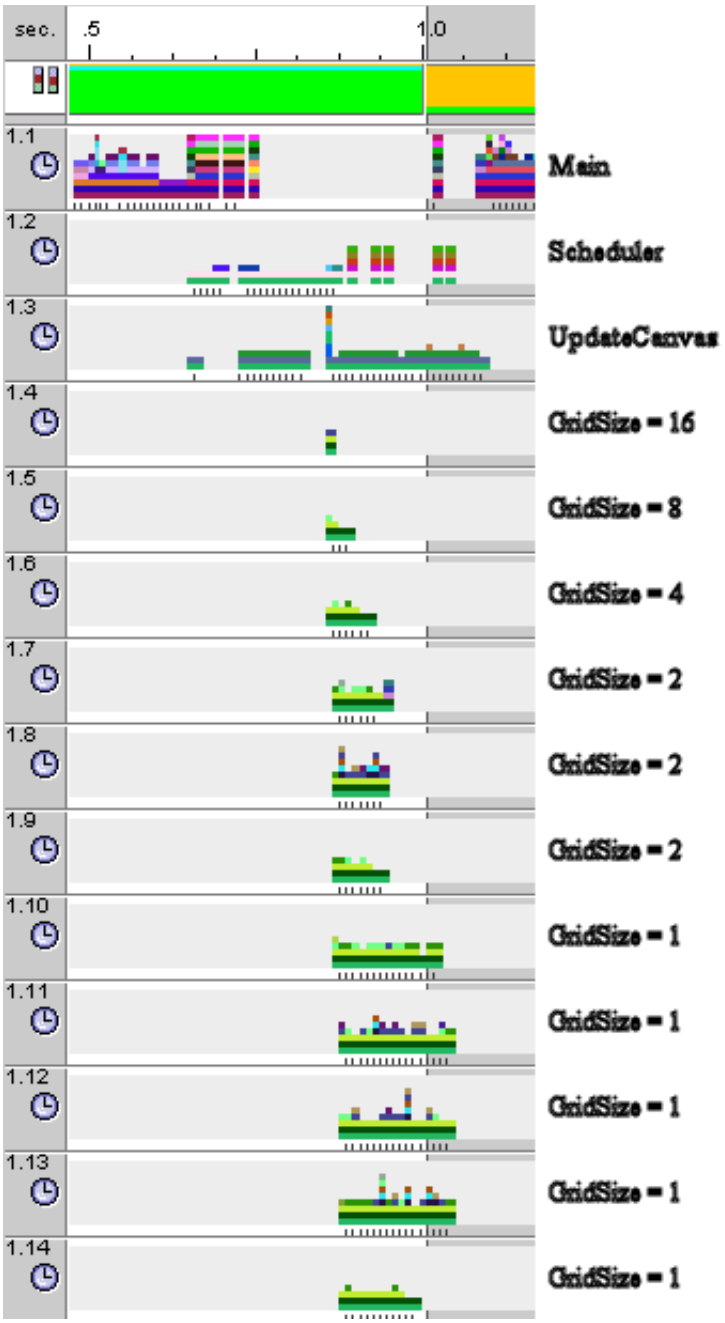
Efter rettelsen blev samme figur renderet på ca. 0,4 sekunder, hvilket som forventet er betydeligt bedre end før fejlen blev rettet. Nu er der kun et par

enkelte huller (et i tråd 766 og et i 771). Disse skyldes at gridstørrelse 2 og 1 hver har en hovedtråd, der sørger for at opdatere *canvas-bufferen* med pixelværdier fra trådene med større gridstørrelser. Disse hovedtråde må vente på at trådene fra den foregående gridstørrelse samt andre tråde med samme gridstørrelse afsluttes før denne opdatering kan foretages.

Lille bolter

Figur 37: Output efter rendering af bolter.kdtree. Figuren består af 4.138 trekanter og kD-træet er bygget af 9.938 noder. Overfladespecifikationerne var angivet i tilhørende mtl fil.

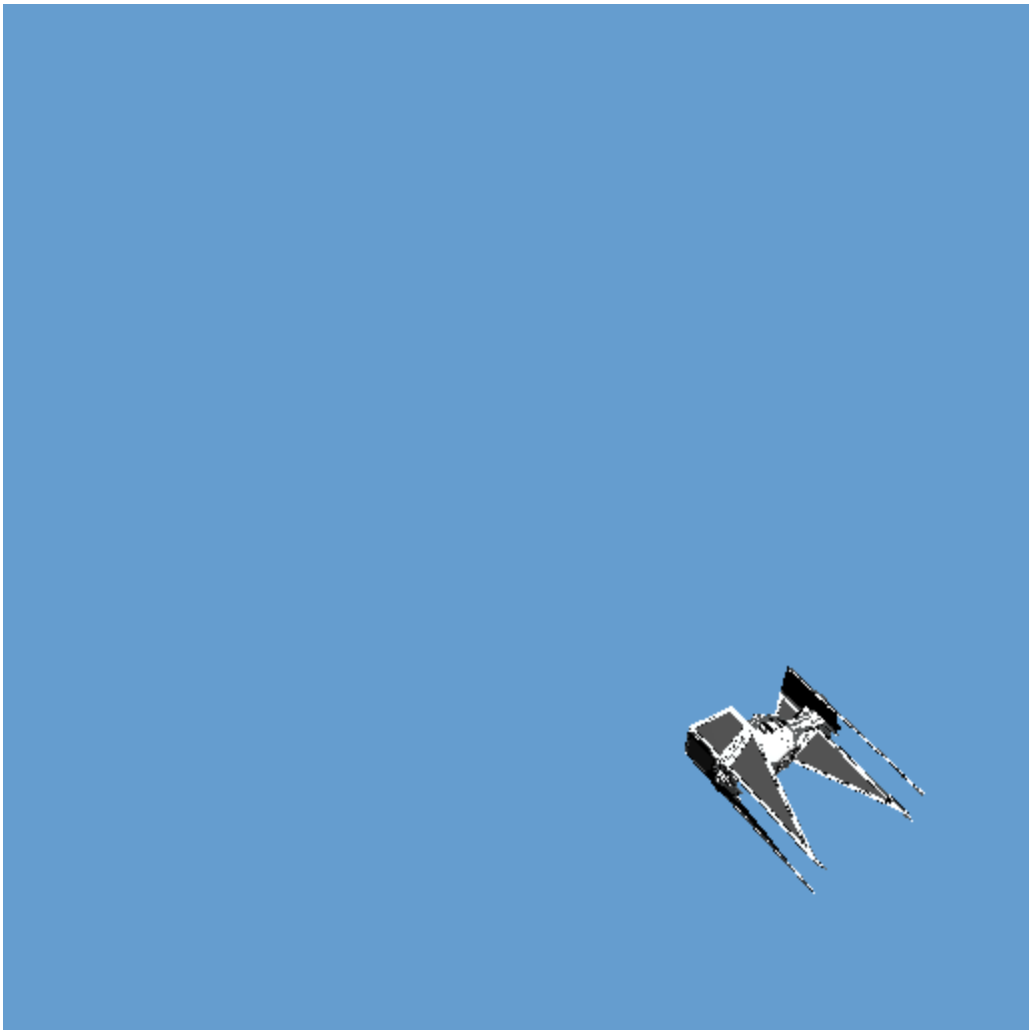
Denne figur består af betydeligt færre trekanter end bunny'en. Dette forventes dog ikke at give betyde det store fordi kD-træet kun giver en logaritmisk forøgelse i søgetiden i forhold til antallet af objekter i træet. Til gengæld forventes omtrent samme, eller lidt hurtigere, pga. de mere diffuse overflader, renderingstid, som ved rendering af den lille bunny, da figuren har omtrent samme størrelse.



Figur 38: ***Timeline screendump*** **der viser renderingsforløbet af Bolter.**

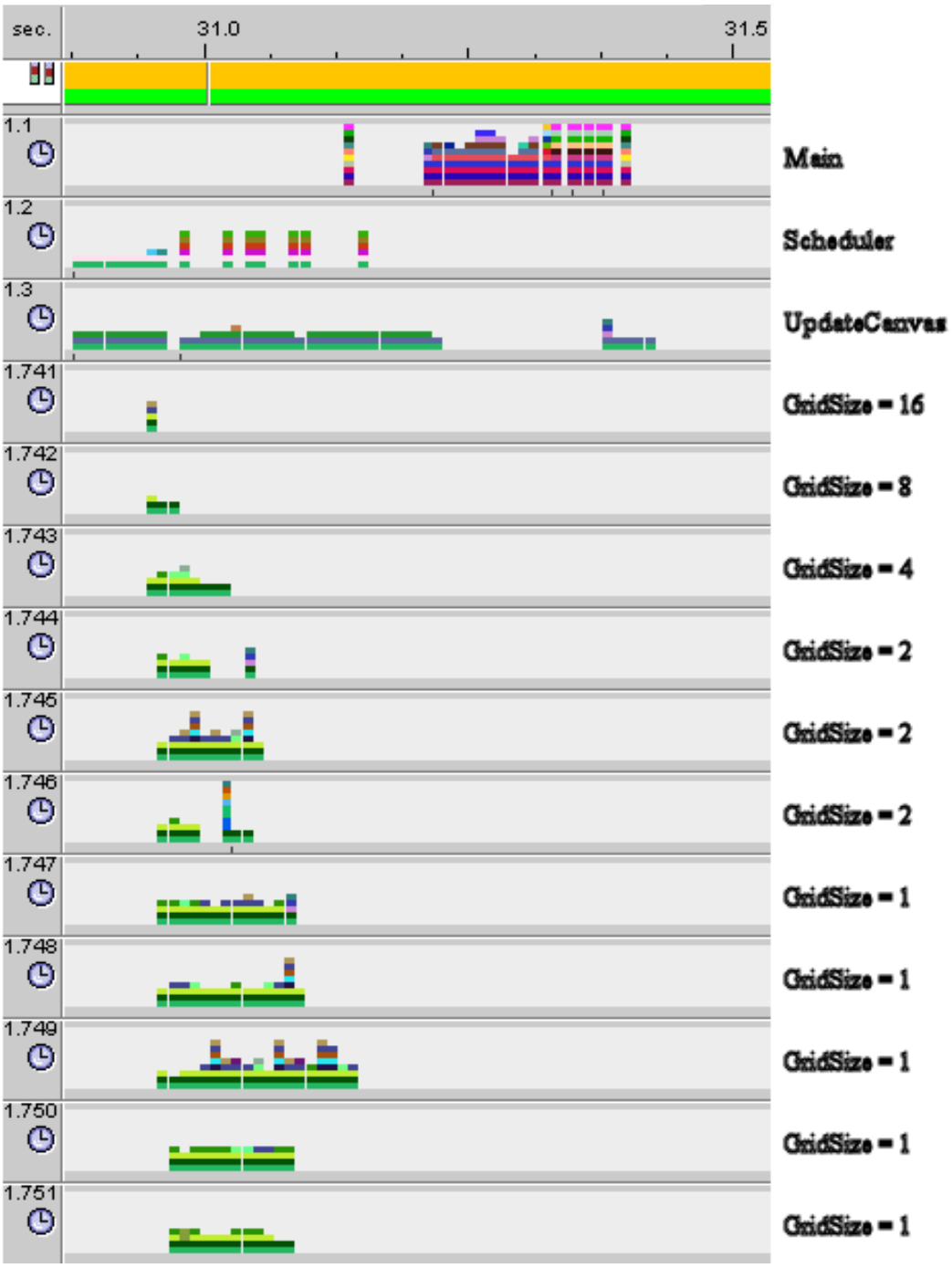
Resultatet viser en renderingstid på ca. 0,2 sekunder, hvilket som forventet en lidt hurtigere renderingstid end for den lille bunny.

Lille TieFighter



Figur 39: Output efter rendering af TieFighterMed.kdtree, hvor der er zoomed ud og navigeret rundt så figuren er placeret i hjørnet.

Igen forventes omtrent den samme renderingstid som for den lille bolter og bunny.



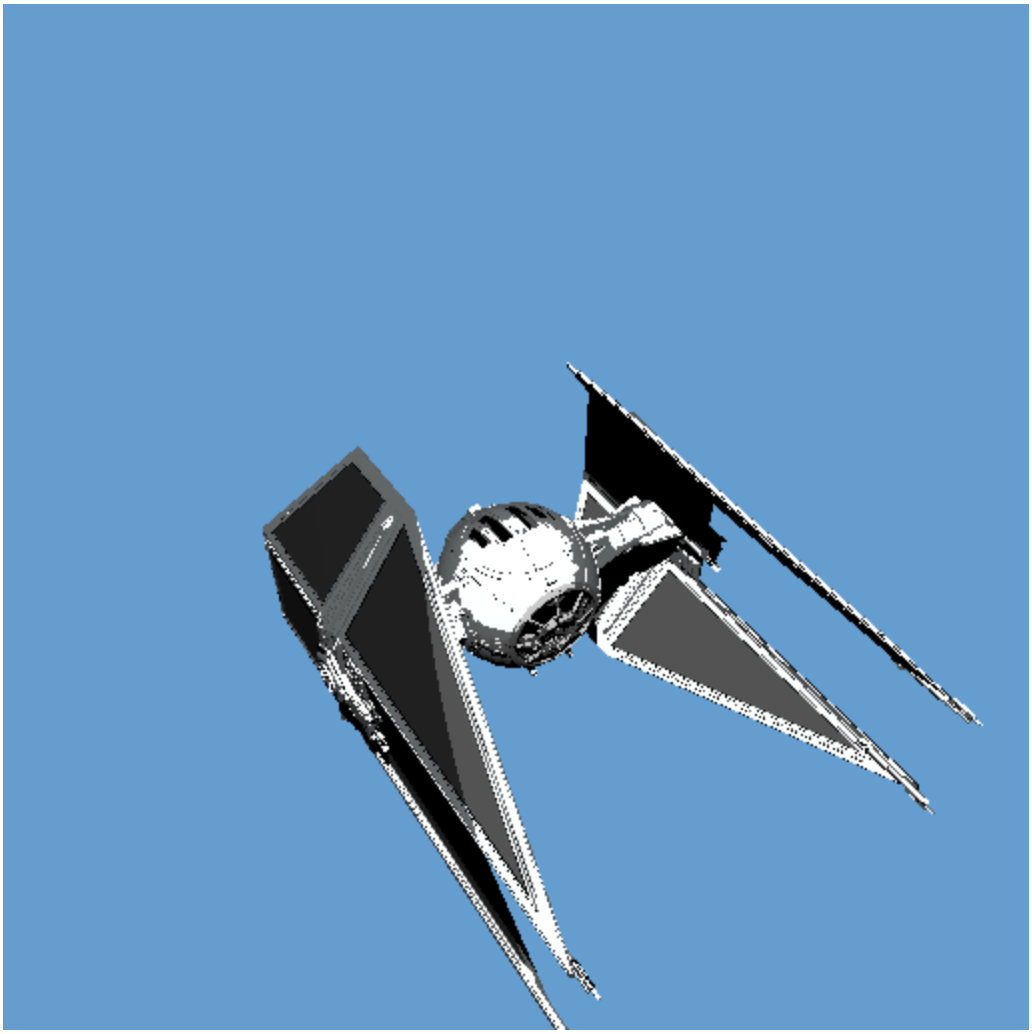
Figur 40: ***Timeline screendump*** **der viser renderingsforløbet af den lille TieFighter.**

Ovenstående resultat viser en renderingstid på omkring 0,2 sekunder. Resultatet er som forventet.

Stor Bunny

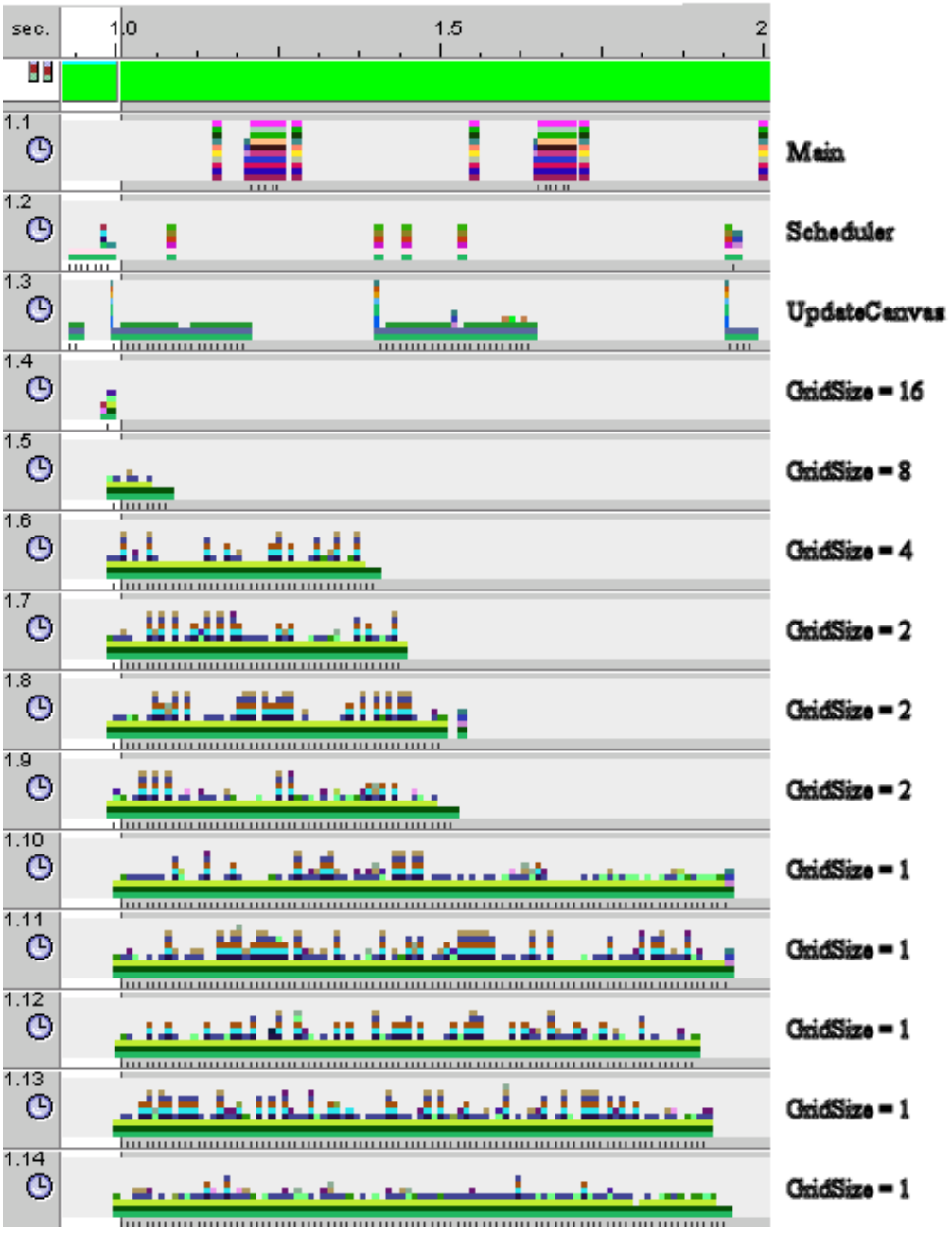
Se Figur 31 og Figur 33.

Stor TieFighter



Figur 41: Output efter rendering af TieFighterMed.kdtree uden supersampling. Scenen består af 34.786 trekanter og kD-træet er bygget af 77.828 noder. Trekanternes overfladespecifikationer var angivet i tilhørende mtl fil.

Der forventes en noget længere renderingstid end for de små figurer, da betydeligt flere stråler vil ramme indenfor figurens *bounding-box*.



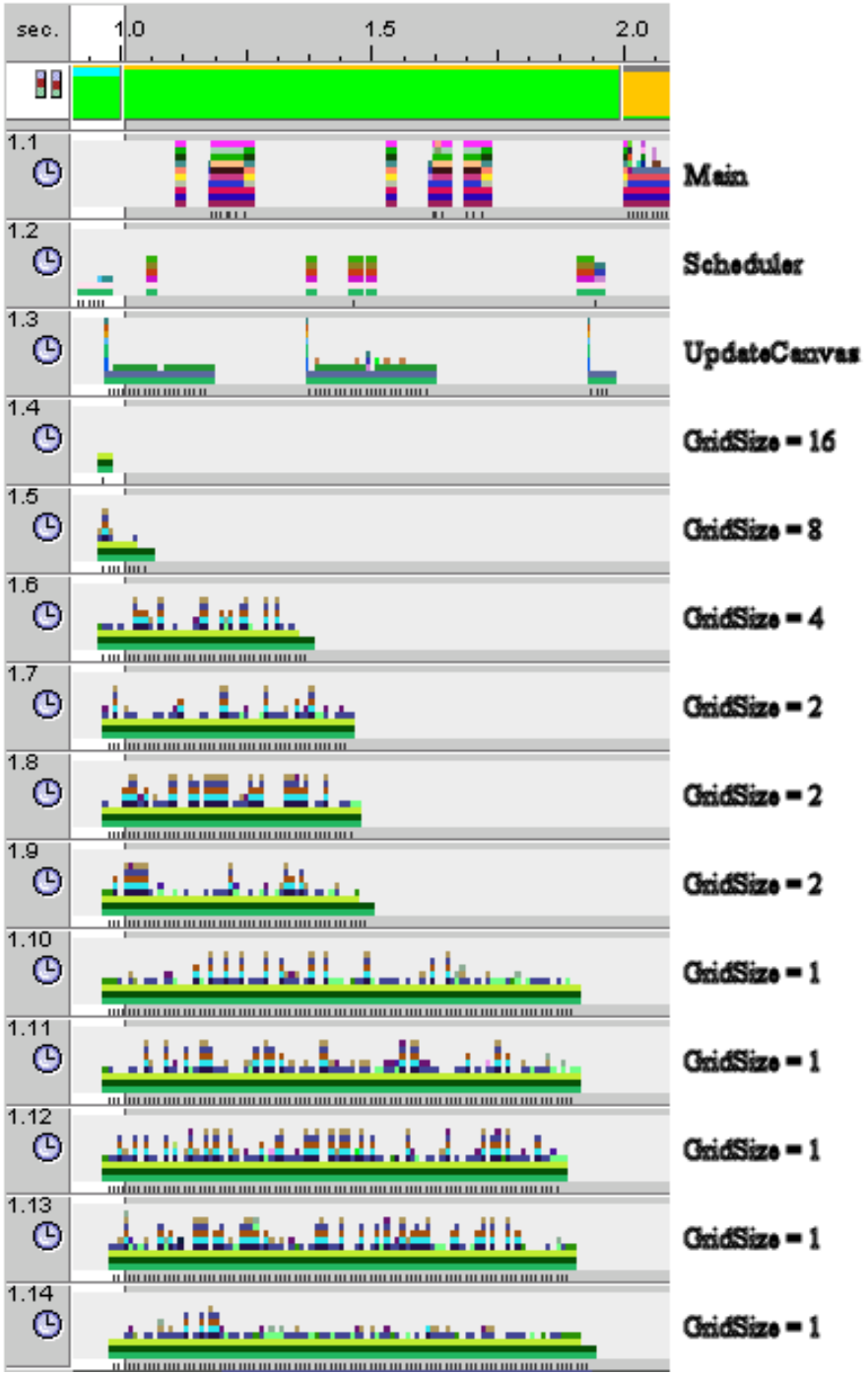
Figur 42: ***Timeline screendump*** **der viser renderingsforløbet af TieFighter.**

Figuren blev renderet på ca. 1 sekund, hvilket vil sige ca. 5 gange langsommere end for den lille TieFighter. Dette svarer ganske godt til den forøgede størrelse af figuren. Resultatet er derfor som forventet.

Stor Armadillo

Figur 43: Output efter rendering af Armadillo_orig.kdtree. Figuren består af 345.944 trekanter og kD-træet er bygget af 412.604 noder. Overfladespecifikationerne er de samme som for Figur 31.

Det forventes at denne figur vil blive renderet på omtrent samme tid som den store TieFighter, da denne figur er lidt mindre, men til gengæld har en spekular overflade, hvilket resultere i at flere stråler skal *traces*.



Figur 44: ***Timeline screendump*** **der viser renderingsforløbet af Armadillo.**

Figuren blev, lige som den store TieFighter, renderet på ca. 1 sekund. Dette var som forventet.

4.3.3 Opsummering

Ved rendering af små figurer viste der sig et problem med at et *mutex* i *UpdateCanvas* funktionen spærrede for arbejdertrådene. Problemet blev løst ved brug af en ekstra temporær buffer.

Testen viste desuden at distribution af arbejdet for gridstørrelse 1 og 2 ud fra *cost-bufferen* fungerer ganske glimrende. Ved de små figurer er opdelingen dog ikke helt så god. Dette skyldes at det grid der blev brugt til at danne *cost-bufferen* ikke her er helt fint nok. Det ville nok være værd at overveje at ændre det første grid (som benyttes til at danne *cost-bufferen*) til *GridSize = 16*. Dette ville så betyde at den første arbejdertråd ville have *GridSize = 8*. Denne tråd laver i forvejen ganske lidt, hvilket bekræfter at det kunne være en god ide at benytte denne gridstørrelse til at danne *cost-bufferen* ud fra. Det eneste der umiddelbart taler i mod at foretage denne ændring, er at SMP systemer med langsommere CPU'er vil være længere tid om at vise det første interpolerede output på *canvas*, hvilket kunne virke forstyrrende ved interaktion med scenen.
Det skal desuden bemærkes, at der tydeligvis kan opnås et endnu højere *speedup* ved brug af flere CPU'er, da især *GridSize = 1* med fordel kan distribueres på flere tråde. Og hvis *GridSize = 4* også blev distribueret til 2 eller 3 tråde ville *GridSize = 2 og 1* kunne distribueres på endnu flere tråde.
En sekundær ting i denne test som er værd at bemærke er den ganske lille forskel på renderingstiden af komplekse figurer og simple figurer (figurer med mange trekanter vs. få trekanter). Det der har betydning er således hovedsagelig hvor stor del af *canvas* figuren dækker. Dette passer godt med analogien af kD-træstrukturen, der jo netop er genial fordi *traversal-time* kun forøges ganske lidt når antallet af objekter forøges betragteligt.

4.4 Ikke nok CPU'er

4.4.1 Formål

I dette afsnit kigges der på, hvordan programmet opfører sig, hvis der ikke er nok CPU'er til rådighed til, at de enkelte tråde kan afvikles på hver deres individuelle CPU. Formålet er således at teste at programmet stadig fungerer korrekt på systemer med "få" CPU'er og om der stadig opnås en forbedret performance i forhold til, hvis hele billedet blev renderet af en enkelt CPU. Dette er interessant, da PC'er der reelt er små SMP systemer[14] bliver mere og mere normale. Det vil således være glædeligt hvis raytraceren også opnår

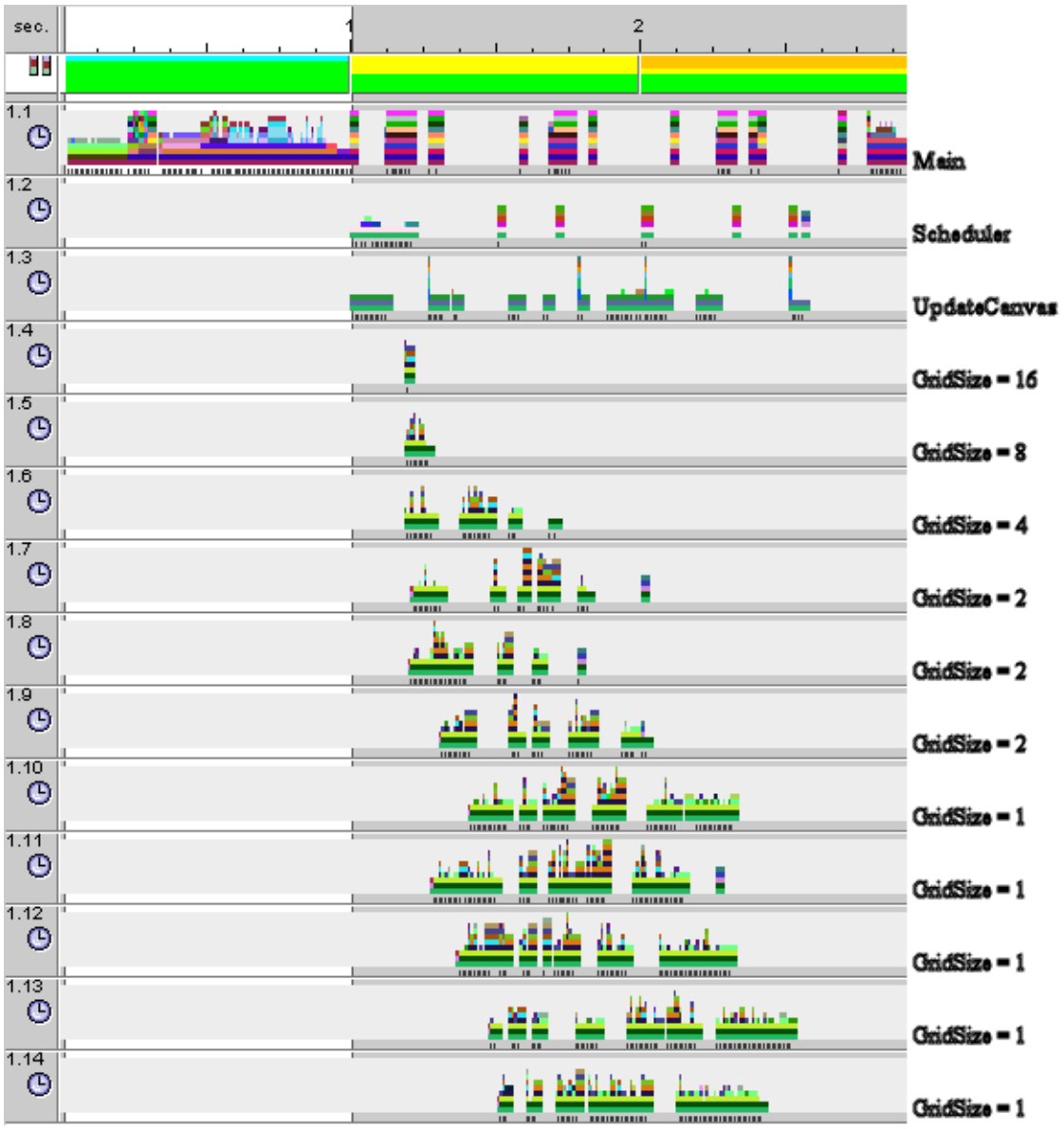
[14] som tendensen er nu vil de fleste PC'er inden for få år have 2, 4 eller flere kærner.

forbedret performance på systemer med færre CPU'er end det ideelle (14, en for hver tråd).

4.4.2 Resultat

Testen blev udført på flere forskellige figurer, men da tendensen var den samme, vælger jeg kun at vise *screendump* fra en enkelt.

Stor Bunny



Figur 45: ***Timeline screendump*** **der viser renderingsforløbet af bunny ved brug af færre CPU'er end der er tråde.**

På Figur 45 ses det tydeligt at trådene må deles om de tilgængelige CPU'er. Dette kan ses på "hullerne" i arbejdertrådene (sammenlign evt. ovenstående

figur med Figur 33, hvor den samme figur er renderet med tilstrækkeligt mange CPU'er).

Arbejdertrådene bliver færdige på ca. 1,5 sekunder. Sammenlignet med Figur 33, hvor trådene blev færdige på 1 sekund ved brug af tilstrækkeligt mange CPU'er, betyder de for få CPU'er således en forsinkelse på 0,5 sekunder.

Brug af 4 CPU'er giver et *speedup* på (der benyttes stadig 11 tråde):

$$S(11) = T(1) / T(11)$$
$$\Rightarrow$$
$$S(11) = 3{,}3 / 1{,}5 = 220\%$$

Og en effektivitet på:

$$E(11) = S(11) / 11$$
$$\Rightarrow$$
$$E(11) = 2{,}2 / 11 = 20\%$$

4.4.3 Opsummering

Resultaterne viste at de 4 CPU'er giver et ganske pænt *speedup* på 220%. I øvrigt er det værd at bemærke, at ud over at Sunq05 har færre CPU'er til rådighed, så kører de kun med 1062MHz imod Bohrs 1200MHz. Denne test er ganske positiv, da raytraceren således kan udnytte ganske små SMP systemer (det vil ikke vare længe før almindelige PC'er har fire kærner) til forbedret *speedup*. Effektiviteten er naturligvis lavere, end når hver tråd har sin egen CPU, fordi der altid vil være et vist *overhead* når trådene "pauses" under eksekvering (hullerne i Figur 45).

4.5 Ren OpenMP

4.5.1 Formål

At undersøge hvor mange tråde der skal bruges for at opnå bedst mulig performance ved brug af OpenMP, samt at finde den mest velegnede *scheduling*.

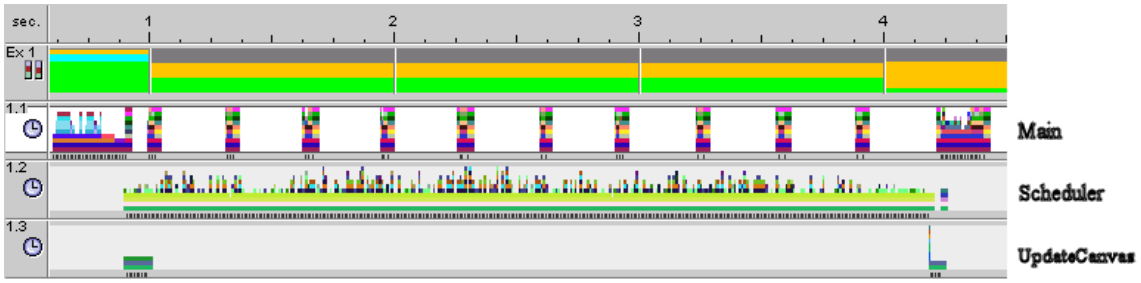
De bedst mulige OpenMP indstillinger findes ved at rendere et billede ved brug af fra 2 til 10 tråde ved alle tre former for OpenMP *scheduling* (*static*, *dynamic* og *guided*).

Forskellen på de tre *scheduling* typer er følgende:

- *Static scheduling* med *default chunksize* opdeler det paralleliserede *for-loop* i lige store dele til hver tråd.
- *Dynamic scheduling* distribuerer dynamisk et element af gangen (hvilket her vil sige en søjle af pixels) indtil hele billedet er renderet.
- *Guided scheduling* distribuerer gradvist eksponentielt mindre og mindre dele til de enkelte tråde.

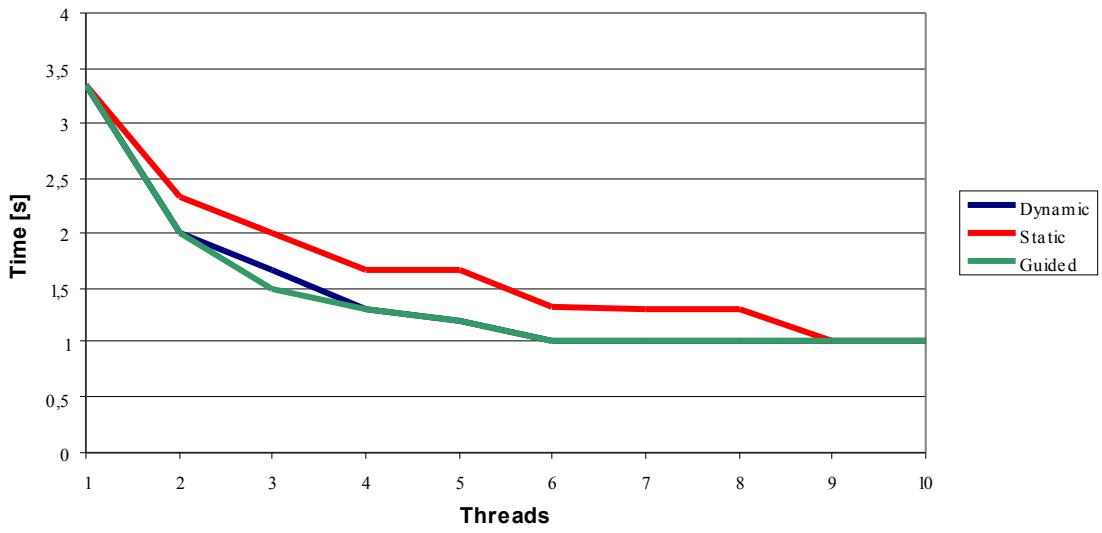
Jeg vælger at lave alle tests på figuren ved navn bunny (samme som Figur 31).

4.5.2 Resultater



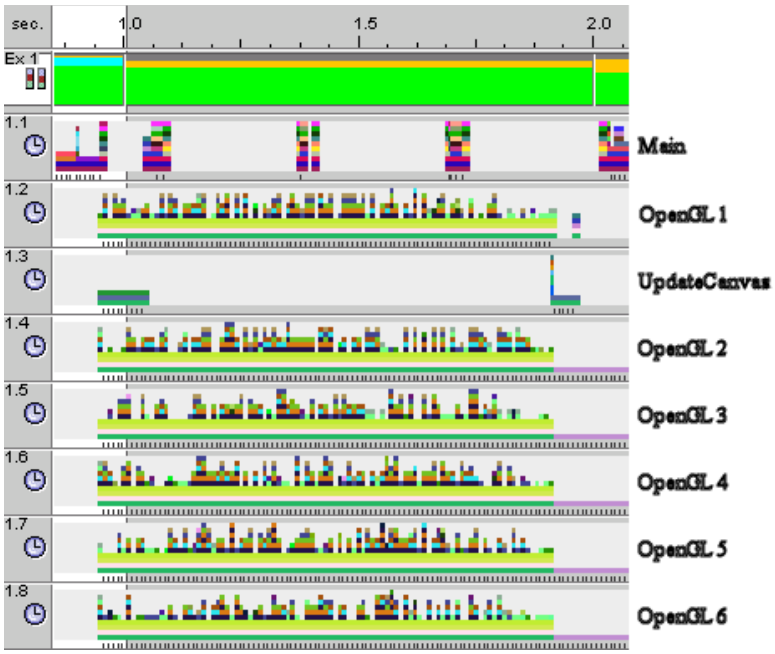
Figur 46: Kun en enkelt tråd til rendering. Canvas bliver opdateret når alle pixels i billedet er renderet.

Ovenstående figur illustrerer arbejdet i scheduler tråden, der skal distribueres ud på x antal tråde ved brug af OpenMP, så der opnås bedst mulig performance. En enkelt tråd kan således rendere bunny billedet på ca. 3,3 sekunder.



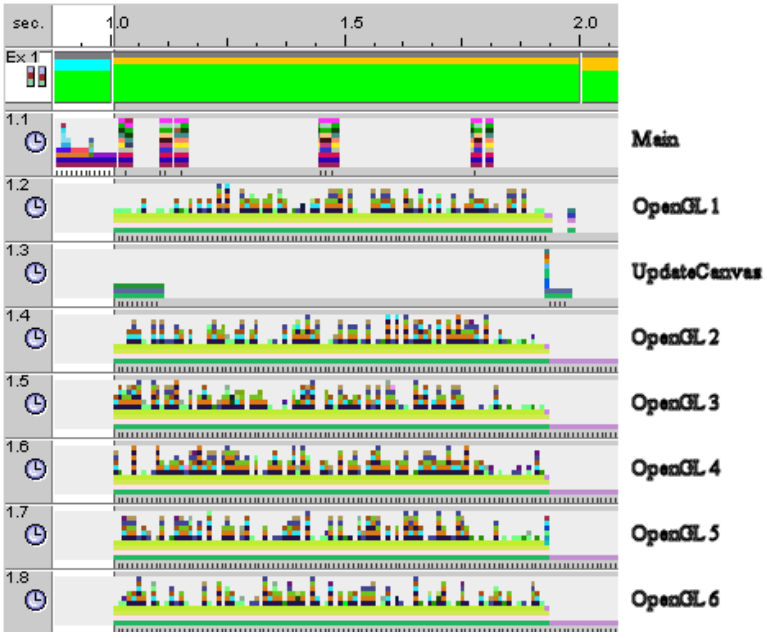
Figur 47: Grafen viser forholdet mellem renderingstid og antallet af OpenMP tråde ved default chunksize for henholdsvis dynamic, static og guided scheduling.

På grafen i Figur 47 ses det, hvordan performance forbedres, jo flere tråde der benyttes. Det ses desuden at, ved *default chunksize,* opnås den bedste performance for *static* og *guided scheduling* ved brug af 6 tråde, men for *static* først ved 9 tråde. Det viste sig desuden at performance også topper ved 6 tråde for *static scheduling*, hvis *chunksize* sættes til et sted mellem 4 og 16 (se resultatskema fra denne test i Appendiks B). Den bedste performance for alle tre *scheduling* typer, ved rendering af bunny figuren, var på ca. 1 sekund. Det lykkedes ikke at forbedre dette uanset hvilken *chunksize* der blev benyttet.



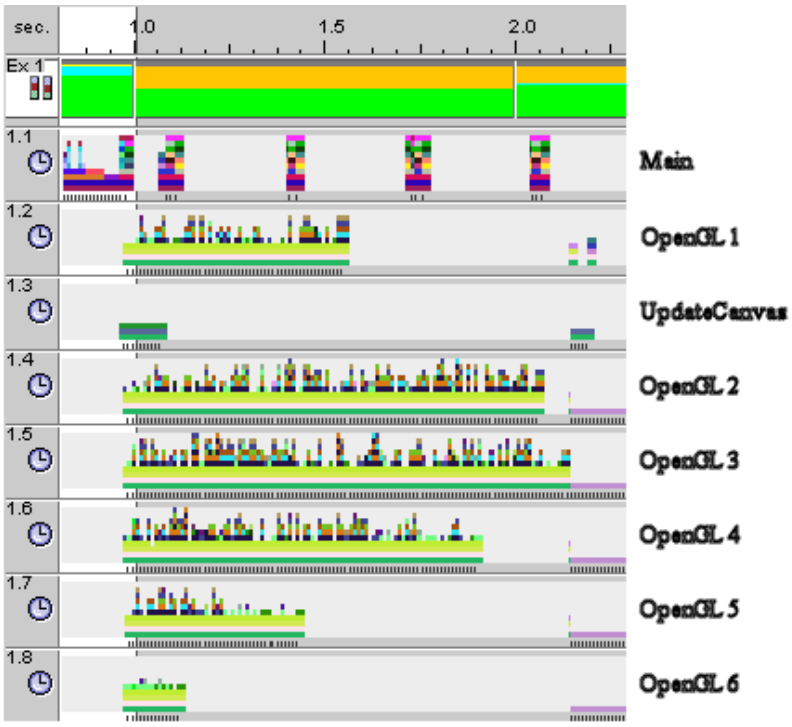
Figur 48: 6 OpenMP tråde, dynamic scheduling.

Ovenstående figur viser at arbejdet distribueres ganske jævnt mellem trådene. Dette er ganske som forventet pga. den lille *default chunksize*.



Figur 49: 6 OpenMP tråde, guided scheduling.

Også ved *guided scheduling* distribueres arbejdet ganske pænt mellem trådene. Dette er som forventet. Da *chunksize* formindskes under renderingsforløbet, vil det kun være i ekstreme tilfælde, hvor der er områder i scenen der har betragteligt længere renderingstid end resten af scenen, der kan resultere i en skæv fordeling ved *guided scheduling*.



Figur 50: 6 OpenMP tråde, static scheduling.

Ved *static scheduling* ses det at arbejdet fordeles skævt mellem trådene. Dette er ikke overraskende, da den renderede figur ikke fylder hele billedet. Dermed vil der være områder af billedet der tager længere tid at rendere end andre.

Figur 48 til Figur 50 viser hvorfor *static scheduling* har dårligere performance end *dynamic* og *guided scheduling* ved *default chunksize* som man så det på grafen i Figur 47.

Det er således ikke overraskende at *static scheduling* adskiller sig ved at have den dårligste fordeling af trådene. Til gengæld overrasker det mig noget, at specificering af *chunksize* generelt har så lille betydning (så godt som ingen, på nær lige for *static scheduling*). Jeg havde forventet, at det overhead der vil være når trådene ”fodres” med meget små *chunks,* ville betyde en dårligere performance, end hvis man distribuerede lidt større *chunks*. Hvis man beregner det arbejde der bliver udført for henholdsvis *dynamic* og *static scheduling* kan man dog se at der er noget overhead.

En tråd: 3,3 sekunder
Dynamic: 6 x 0,9 = 5,4 sekunder
Static: 0,6 + 1,1 + 1,2 + 0,9 + 0,5 + 0,1 = 4,4 sekunder

Dynamic scheduling har således 1 sekunds mere *overhead* end *static scheduling* og 2,1 sekunds *overhead* i forhold til når alt arbejdet bliver gjort af en enkelt tråd.

Alt i alt giver brug af 6 tråde med OpenMP et *speedup* på:

S(6) = T(1) / T(6)
=>
S(6) = 3,3 / 1 = 330%

Og en effektivitet på:

E(6) = S(6) / 6
=>
E(6) = 3,3 / 6 = 55%

4.5.3 Opsummering

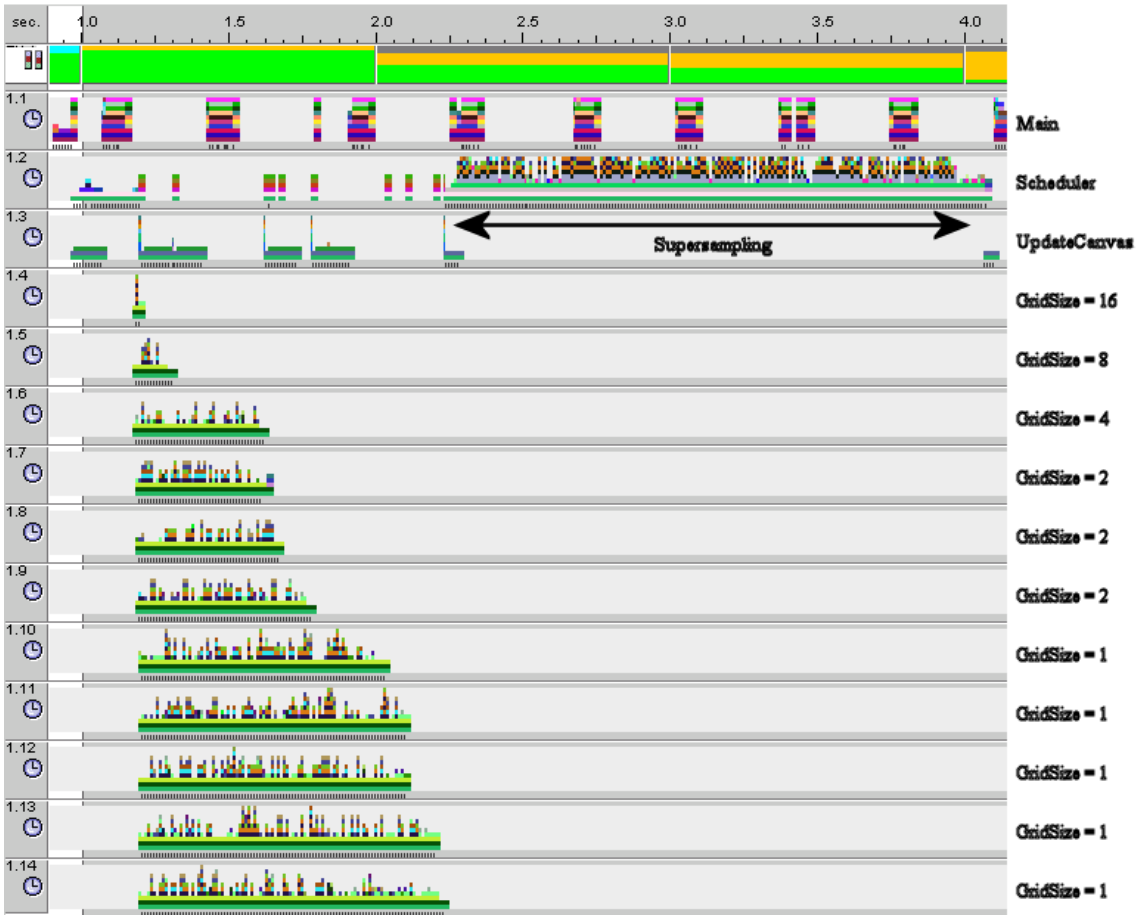
Testen viste at det bedst mulige *speedup,* på 330% (den samme som ved Pthreads løsningen), blev opnået ved brug af 6 tråde og enten *dynamic* eller *guided scheduling* med *default chunksize* eller ved *static scheduling* og en *chunksize* på 4-16.
Effektiviteten er desuden noget bedre end ved Pthreads løsningen.

4.6 Supersampling

4.6.1 Formål

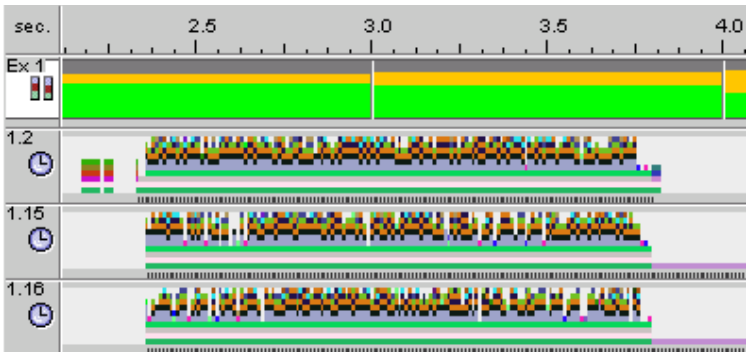
Her kigges der på supersampling henholdsvis med og uden brug af OpenMP. Der vil blive eksperimenteret med antallet af tråde samt med *scheduling* for at finde de bedst mulige indstillinger.

4.6.2 Resultater



Figur 51: ***Timeline screendump*** **der viser Bunny renderet med supersampling.**

Ovenstående figur illustrerer det stykke af *Scheduler* tråden, hvor supersamplingen bliver udført. Det er dette stykke der skal paralleliseres ved brug af OpenMP. Det forventes at der kan opnås omtrent samme resultat for parallelisering af supersampling som ved brug af OpenMP til rendering generelt (se afsnittet ”4.5 Ren OpenMP”). Når en tråd foretager supersampling alene tager det således for denne figur ca. 1,8 sekunder.



Figur 52: Bunny med supersampling ved brug af OpenMP med 3 tråde og dynamic sheduling. Kun OpenMP trådene er vist. De øvrige tråde er uinteressante i denne sammenhæng og ligner i øvrigt dem fra Figur 51.

Ovenstående resultat viser at supersampling ved brug af 3 tråde tager ca. 1,8 sekunder, hvilket er ganske skuffende da det er samme tid som det tog for en enkelt tråd at lave supersampling. Der opnås næsten eller slet ingen forbedring ved parallelisering. Billedet var det samme uanset antallet af tråde og variation af *scheduling* samt *chunksize.* Det blev desuden forsøgt at sætte *SUNW_MP_THR_IDLE = spin* for at forhindre at trådene evt. skulle gå i *sleep mode.* Heller ikke dette hjalp.

4.6.3 Opsummering

Resultaterne viste at supersamplingen er meget dårlig paralleliserbar. Præcis hvad dette skyldes står uklart, men et gæt vil være at OpenMP ikke synes om rekursiviteten i den adaptive supersampling.

5 Konklusion

Der er blevet udviklet en raytracer med understøttelse af refleksioner, transmissioner, bløde såvel som hårde skygger, *area*- og *point*- lyskilder. Raytraceren er bygget op omkring et kD-træ med *SAH* (*Surface Area Heuristic*) der tilstræber optimale splitpositioner, så *traversal-time* bliver hurtigst mulig. At kD-træet virker efter hensigten, ses på den ganske lille betydning antallet af objekter i scenen har for renderingstiden (*traversal-time* vokser logaritmisk i forhold til antallet af objekter i træet). Det der har betydning er derimod hvor stor en del af skærmen der dækkes af objekter.
Raytraceren kan bygge kD-træet ud fra *wavefront* filer og gemme det færdigbyggede træ i en fil, således at den langsommelige proces det er at opbygge kD-tæet kun skal foretages første gang en ny scene renderes. Næste gang hentes kD-træet blot ind fra den fil der blev dannet ud fra *wavefront* filen.
Ved brug af en enkelt CPU renderer raytraceren billeder på 512x512 pixels på op til ca. 10 sekunder, på en UltraSPARC IIIi 1062MHz CPU (når størstedelen af billedet er dækket af objekter). Typisk kan et billede dog renderes på omkring 3 sekunder.
Som forventet er raytraceren således alt for langsom til, at mennesker med almindelig tålmodighed vil kunne holde ud at navigere rundt i scenen. At skulle vendte op adskillige sekunder hver gang kameraet flyttes, er grunden til at rasterisering er det foretrukne valg, når målet er real-time rendering.
Fundamentet for dette projekt - at en raytracer som udgangspunkt langt fra er hurtig nok til afvikling i real-time – er således realt nok. Derfor gav det god mening at fortsætte projektet, med fokus på essensen, - at parallelisere koden til afvikling på et SMP system, for på den måde at opnå real-time rendering.

Ved brug af OpenMP fandt jeg at 6 UltraSPARC IV 1200MHz CPU'er giver et *speedup* på omkring 330%. Rendering af en typisk scene kom således ned på omkring 1 sekund. Dette er dog stadig ikke godt nok til, at man kan bevæge sig rundt i scenen uden at finde det ganske langsommeligt. Derfor var det nødvendigt at udvikle en mere kompleks renderingsstrategi, der hurtigere giver et billedoutput. Jeg valgte at rendere billedet i steps, hvor der for hvert step blev renderet pixels i et grid der blev gradvist finere. Efter

hver gridrendering, interpoleres de resterende pixels. På denne måde kan der hurtigt dannes et tilnærmet resultat, der gradvis forbedres indtil billedkvaliteten bliver optimal. Brugeren kan således hurtigt se et tilnærmet billede, som er godt nok til, at man kan orientere sig i scenen og flytte kameraet rundt i scenen. Når så kameraet er flyttet til en ønsket position vil brugeren naturligt lade kameraet stå stille, hvilket resulterer i at billedkvaliteten forbedres indtil den maksimale kvalitet er opnået. Denne strategi er OpenMP ikke fleksibel nok til at kunne håndtere. Derfor blev den i stedet implementeret ved brug af Pthreads.

Pthreads løsningen giver ved brug af 11 arbejdertråde (hver med individuel UltraSPARC 1200MHz CPU) det samme *speedup,* på 330%, som OpenMP løsning gør ved brug af 6 CPU'er. Pthreads løsningen leverer til gengæld sit første billedoutput efter omkring 0,1-0,3 sekunder og giver således en betydelig bedre oplevelse ved navigation i scenen.
Resultatafsnittet viste desuden, at Pthreads løsningen fungerer ganske godt, selvom der ikke er tilstrækkeligt mange CPU'er til rådighed, til at alle tråde kan få sin egen. Ved kun 4 tilgængelige UltraSPARC 1062MHz CPU'er blev der opnået et *speedup* på ca. 220%. Pthreads løsningen er desuden kun omkring 13% langsommere end OpenMP løsningen med samme antal CPU'er til rådighed (se Figur 45 og Figur 47). Dette mener jeg er ganske godt gået, idet at Pthreads løsningen leverer væsentligt flere billedoutputs og dermed laver en del ekstra arbejde.
Pthreads løsningen giver desuden mulighed for yderligere *speedup* ved at distribuering renderingen over endnu flere CPU'er, hvorimod det maksimale *speedup* for OpenMP blev opnået allerede ved 6 CPU'er.

Der var to ting der gik mindre godt i dette projekt. Den ene var parallelisering af supersampling og den anden et problem med at firkantede områder af billedet nogle gange ikke blev færdigrenderet, - især ved afvikling på Erlang serveren. Se Appendiks E for en beskrivelse af disse problemer.

Udviklingsmuligheder:
Selvom den implementerede statiske løsning til distribution af arbejdet mellem trådene viste sig ganske god uanset scenetype og -størrelse, vil det være nærliggende at lave automatisk dynamisk distribution af arbejdet mellem trådene, ud fra hvor mange CPU'er der er til rådighed og hvor meget renderingsarbejde der reelt er i billedet (ud fra *cost-bufferen*). Med en sådan automatisk dynamisk distribution ville man kunne tilstræbe, at den optimale distribution altid blev opnået uanset antallet af CPU'er til rådighed.

Med baggrund i den nyudviklede *BIH (Bounding Interval Hierarchy)* datastruktur [BIH], ville det være spændende at udskifte det brugte kD-træ med denne. Dette ville give mulighed for at bygge datastrukturen i real-time med kun en ganske lille forringelse af *traversal-time* (i forhold til kD-træet med *SAH*) og dermed åbne op for en masse muligheder såsom animerede scener eller real-time modellering af objekterne i scenen. *BIH* datastrukturen skulle desuden være ganske simpel at implementere.

Ordforklaring

3DNow!:	AMDs SIMD extension.
AltiVec:	IBM/Motorolas SIMD extionsion.
Analyzer:	En GUI til performance analyse af programmer udviklet i C, C++, Fortran og Java.
API:	API står for Application Programming Interface og er som navnet antyder et interface til et program eller et bibliotek.
BIH:	Bounding Interval Hierarchy. En helt ny (2006) meget lovende datastruktur, der kan bygges i realtid.
Bohr:	Sun Fire E6900 SMP med 48 UltraSPARC IV CPU'er (1200 MHz/dual-core/8 MB L2-cache per core), 96 GB memory.
Boundingbox:	En boundingbox benyttes til at effektivere geometriske beregninger. Ved at pakke objekter (f.eks. defineret ved en union af mange små trekanter) ind i et mere simpelt primitiv (kaldet en boundingbox), kan der regnes på boundingboxen i stedet for på objektet selv og derved simplificere beregningen.
BSP:	Binary Space Partitioning.
Colour bleeding:	Ses eksempelvis i hjørnet af et rum med to forskelligfarvede vægge. Væggene afgiver så at sige farve til hinanden pga. diffus refleksion.
Caustics:	På dansk bedst beskrevet som forstørrelsesglaseffekten. Altså når lyset samles efter f.eks. brydning i et glas el. lign.

DMPP:	Distributed Memory Parallel Processing. Et antal computere der arbejder sammen i et netværk som var de en enkelt kraftig computer.
MMX og SSE:	MultiMedia eXtensions og Streaming SIMD Extensions. Intels SIMD extension.
MPI:	Message Parsing Interface.
OBJ filer:	OBJ fil formatet er et simpelt data format til beskrivelse af 3D geometri. OBJ filer kaldes også for *Wavefront* files, da formatet er udviklet af Wavefront Technologies.
OpenMP:	Open Multi Processing.
Pthreads:	POSIX (Portable Operating System Interface for uniX) threads.
Prerendering:	De indledende beregninger der foretages inden selve renderingsprocessen.
Primitiv:	Et simpelt geometrisk objekt. Kan f.eks. være en trekant, firkant en kugle eller lign.
Rendering:	Processen der genererer et billede pixel for pixel ved brug af software der beskriver objekter og lyskilder i billedet.
Renderingstiden:	Den tid det tager at rendere et billede.
SIMD:	Single Instruction, Multiple Data (kan udføre flere, typisk fire, floating point operations på instruktion).
SAH:	Surface Area Heuristic.
Shading:	En process hvor farven på et objekt bestemmes ud fra vinklen og afstand til lyskilder, så der opnås en fotorealistisk effekt.

SMP: Symmetric MemoryProcessing. Et system der giver høj performance ved at udfører individuelle processer samtidigt. SMP systemer bruger ét operativ system og deler hukommelse og harddisk kapacitet.

VIS: Visual Instruction Set. Suns SIMD extensions til SPARC V9 instruktions sættet, som er understøttet af alle UltraSPARC processorer. Kan udføre op til 10 operationer i parallel og giver op til 7 gange performance boost. http://www.sun.com/processors/vis/. Ved brug af SunONE compileren skal der tilføjes –xvis hvorefter der kan benyttes inline macros. Der er også lavet et Development Kit VSDK. Sun's VIS library: mediaLib. Kan hentes gratis fra Sun (har Ray Casting funktioner).

Kilder

[Analyzer]: Sun Studio Performance Analyzer: http://developers.sun.com/sunstudio/analyzer_index.html

[Angel]: Interactive Computer Graphics: "A Top-Down Approach Using OpenGL, 3rd edition, Addison-Wesley 2003, by Edward Angel.

[BIH]: Bounding Interval Hierarchy. http://graphics.uni-ulm.de/BIH.pdf http://en.wikipedia.org/wiki/Bounding_Interval_ Hierarchy

[Carsensen]: Image analysis, vision and computer graphics, 2nd edition, DTU 2002, by Jens Michael Carstensen.

[DevmasterI]: Raytracingtutorial: http://www.devmaster.net/articles/raytracing_series/ part1.php

[DevmasterII]: Om Ray-triangle intersections: http://www.devmaster.net/wiki/Ray-triangle_intersection

[DevmasterIII]: Om stråle-kugle skæring: http://www.devmaster.net/wiki/Ray-sphere_intersection

[Gamedev]: Om vector beregninger: http://www.gamedev.net/reference/articles/ article1443.asp

[Gbar]: Information om de forskellige SMP systemer på DTU: http://www.gbar.dtu.dk/index.php/Hardware# HPC_servers

[GDB]: Om debugging med GDB:

http://www.gnome.org/~newren/tutorials/developing-with-gnome/html/ch03.html

[GEL]: GEometric and Linear algebra tools udviklet af Jackob Andreas Baerentzen:
http://www2.imm.dtu.dk/~jab/GEL/index.html

[GLT]: OpenGL/GLUT documentation:
http://www.nigels.com/glt/doc/index.html

[Havran]: Vlastimil Havran, Assistant Professor, Computer Graphics Group, Department of Computer Science and Engineering, Faculty of Electrotechnical Engineering, Czech Technical University in Prague.:
http://www.cgg.cvut.cz/~havran/

[HavranPhd2001]: Havrans thesis about Heuristic Ray Shooting Algorithms:
http://www.cgg.cvut.cz/~havran/DISSVH/phdthesis.html

[Intersect]: Om 3D object intersections:
http://www.realtimerendering.com/int/

[MacDonald89]: J. David MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing using Space Subdivision. In Proceedings of Graphics Interface '89, pages 152–63, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[MacDonald90]: J. David MacDonald and Kellogg S. Booth. Heuristics for Ray Tracing using Space Subdivision. Visual Computer, 6(6):153–65, 1990.

[Makefiles]: Om makefiles:
http://www.hsrl.rutgers.edu/ug/make_help.html
http://www.opussoftware.com/tutorial/TutMakefile.htm

[OMPF]: Fantastisk forum om real-time raytracing:
http://ompf.org/

[Padley]: Refraction on a Spherical Interface by Paul Padley:

http://cnx.org/content/m13083/latest/

[Phantom]: a.k.a. Jacco Bikker, Overloaded. http://www.devmaster.net/articles/raytracing_series/part1.php

[RRTIGL]: Thesis about Realtime Ray Tracing and Interactive Global Illumination by Ingo Wald.

[RTNews]: Nyhedsside om real-time raytracing: http://www.acm.org/tog/resources/RTNews/html/

[Schorsch]: Om aliasing: http://www.schorsch.com/kbase/glossary/aliasing.html

[Shumacker]: Shumacker, R., Brand, R., Gilliland, M., Sharp, W. Study for Applying Computer-Generated Images to Visual Simulation

[SimpleRT]: Imponerende real-time raytracer: http://homepages.paradise.net.nz/nickamy/raytracer/raytracer.htm

[Slusallek]: Links til mange gode artikler om raytracing af Philipp Slusallek: http://graphics.cs.uni-sb.de/~slusallek/publications.html

[Threads]: Gode links til sider om threads i C++: http://www.linuxselfhelp.com/HOWTO/C++Programming-HOWTO-18.html

[ThreadsII]: Diskussion om threads på forskellige platforme: http://www.2cpu.com/forums/showthread.php?t=72955

[ThreadsIII]: Forum om multithreading: http://www.codeguru.com/forum/forumdisplay.php?f=63

[ThreadsIV]: Glimrende beskrivelse af pthreads: http://www.phptr.com/articles/article.asp?p=169479&seqNum=1&rl=1

[ThreadsV]: Tutorial til OpenMP:
http://www.topcoder.com/tc?module=Static&d1=features&d2=091106

[ThreadsVI]: Implementation om producer consumer problematikken med pthreads:
http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pc.c

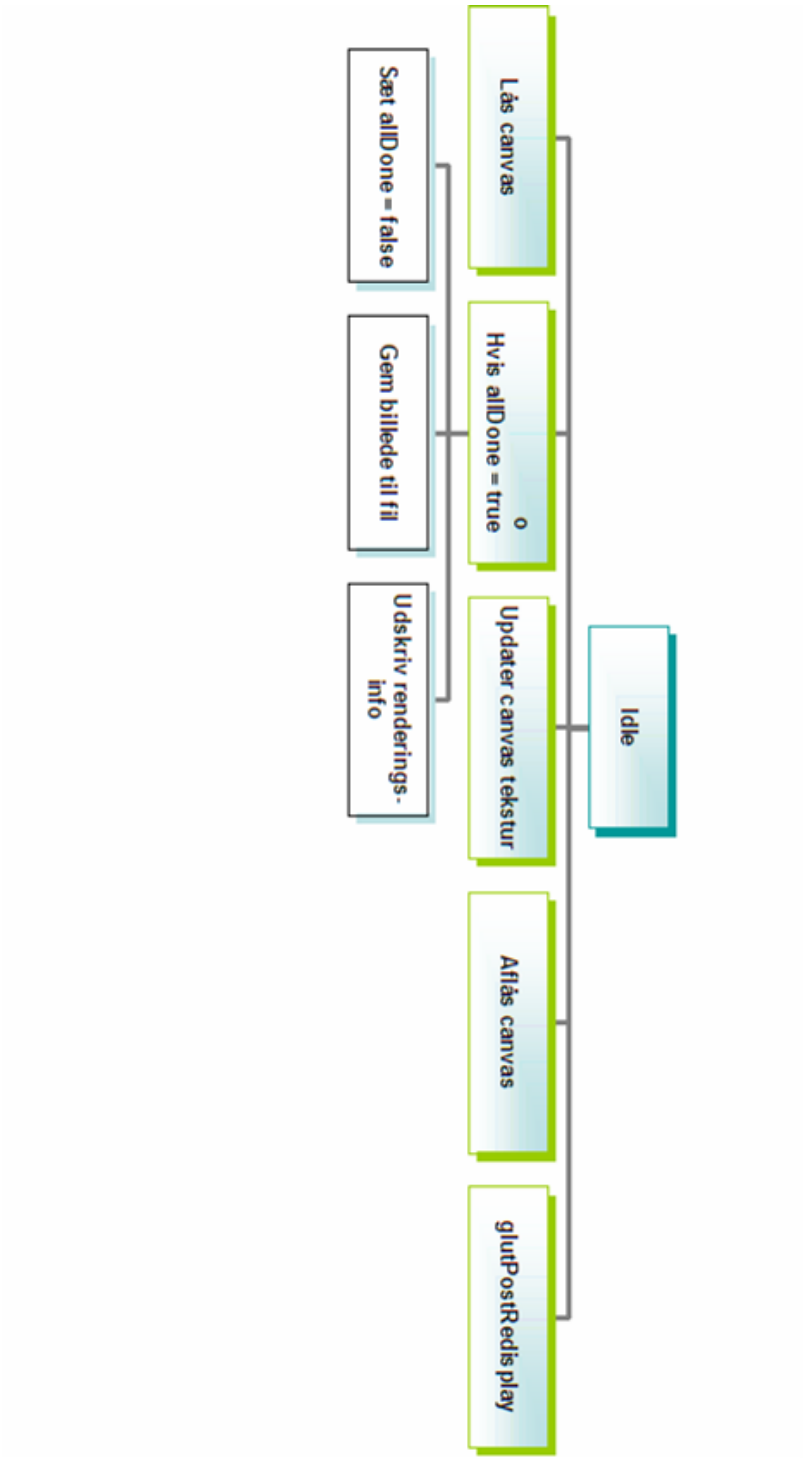
[ThreadsVII]: Multi-Threading på Linux og Win32. Om Mutex Objects og Critical Sections:
http://www.gamasutra.com/features/20051205/paquet_01.shtml

[Watt]: Advanced Animation and Rendering Techniques: "Theory and Practice", Addison Wesley Longman Limited 1992, by Alan & Mark Watt.

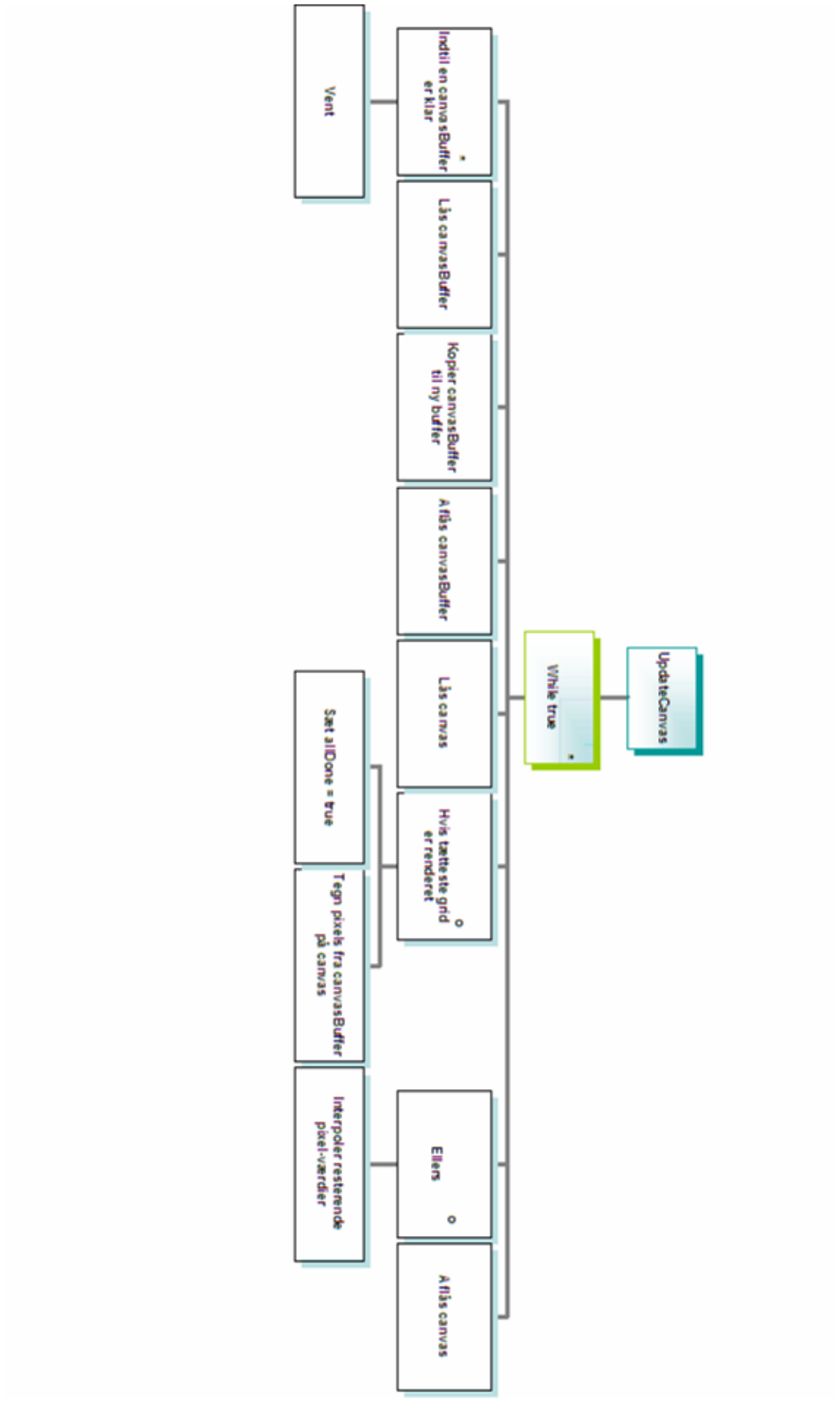
[WikipediaI]: Phong Reflection Model:
http://en.wikipedia.org/wiki/Phong_reflection_model

[Yoon]: Om refleksion og refraktion af Sung-Eui Yoon:
http://gamma.cs.unc.edu/QVDR/temp_homepage/project/comp236/RA4..htm

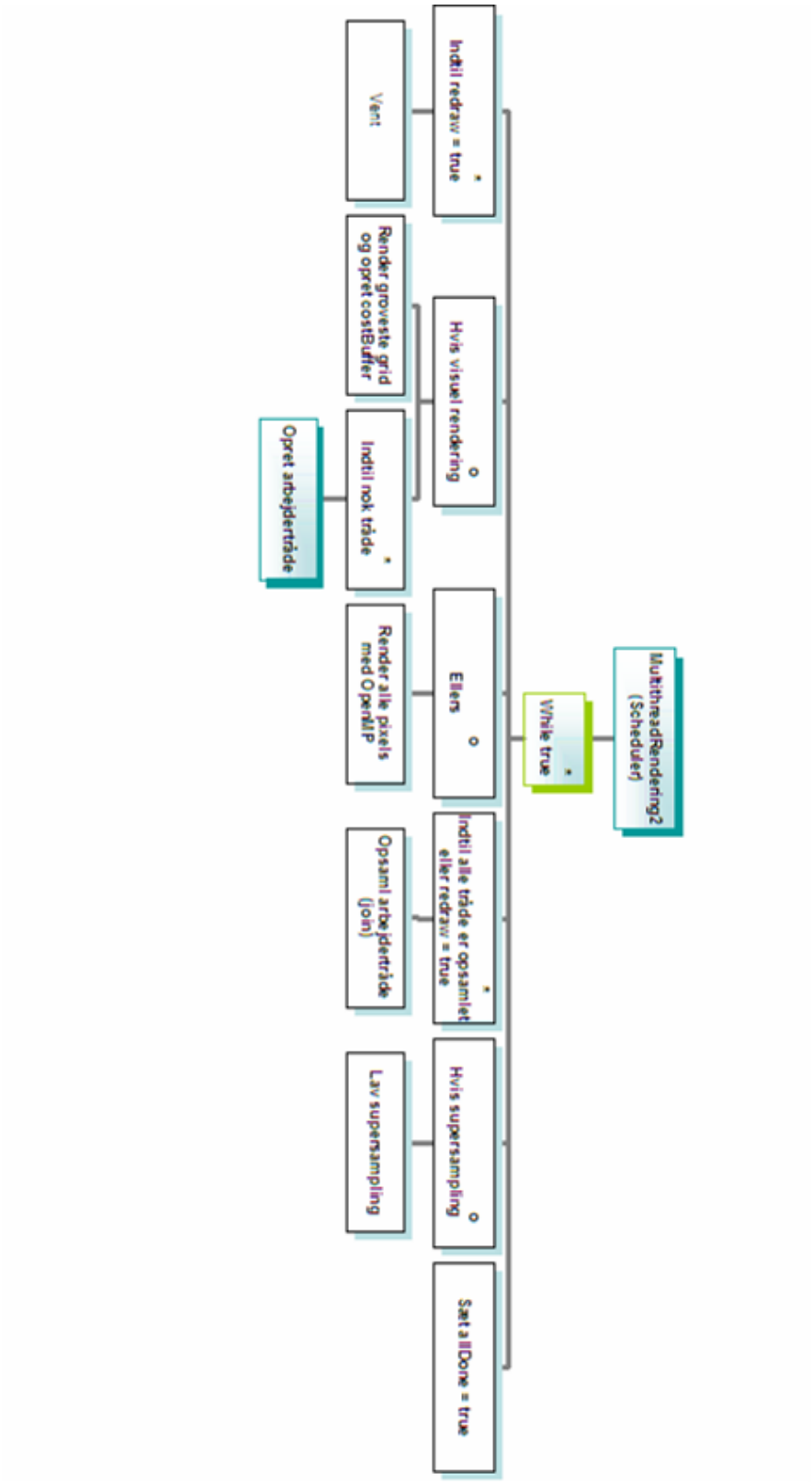
Appendiks A



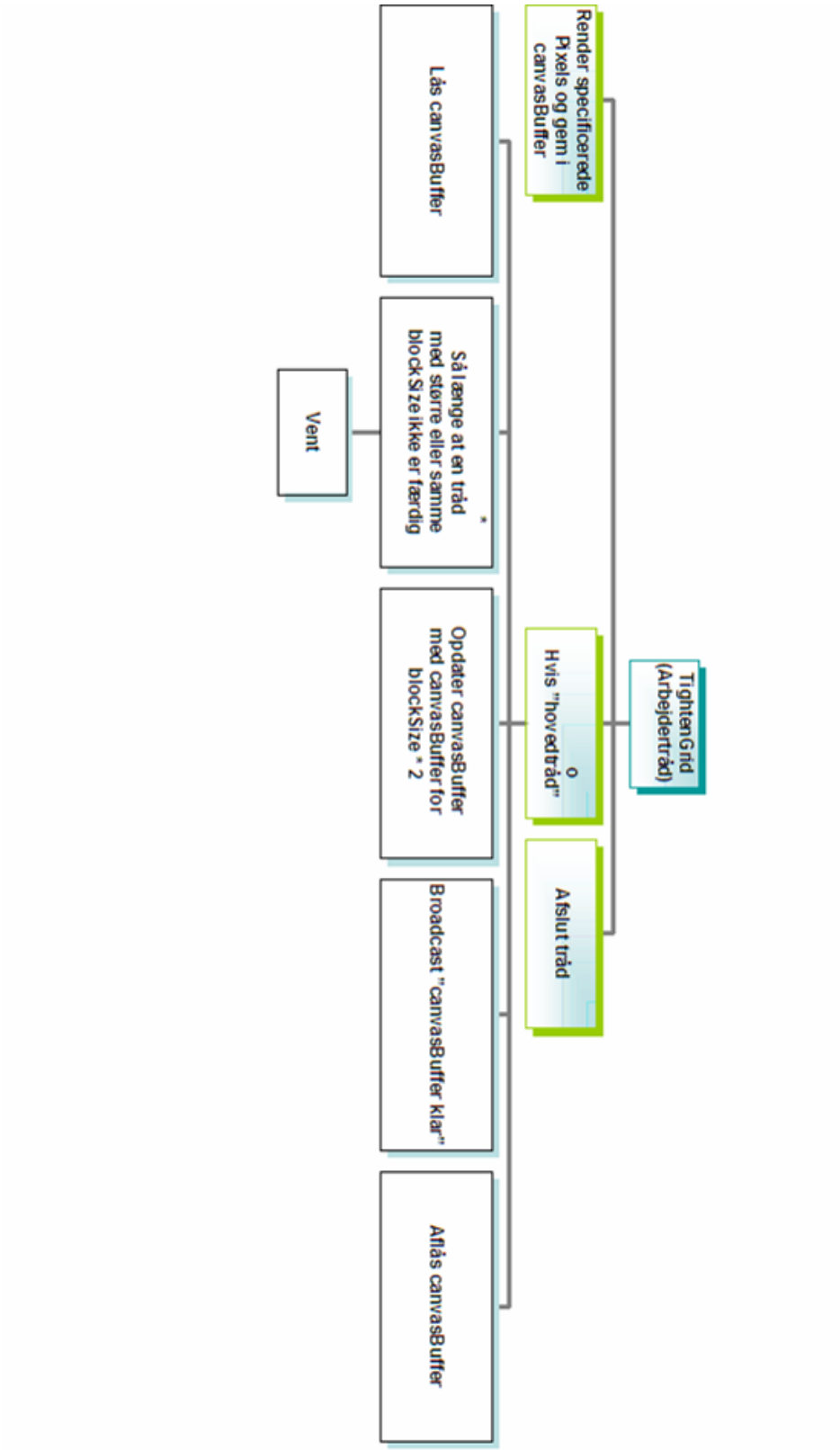
Figur 53: JSP diagram over OpenGL *Idle* funktionen, der reelt er en del af hovedtråden (*main*), men som OpenGL sørger for bliver kaldt jævnligt, så *canvas* holdes opdateret.



Figur 54: JSP diagram over *UpdateCanvas* tråden der sørger for interpolation og opdatering af *canvas* ud fra *canvasBuffer* (der er en *canvasBuffer* til hver arbejdertråd).



Figur 55: JSP diagram over *Scheduler* tråden der sørger for at danne *costBuffer*, distribuere arbejdet for de tætteste grids ud på flere tråde, starte alle arbejdertrådene og lave supersampling.



Figur 56: JSP diagram over *TightenGrid* arbejdertrådene, der renderer et af *Scheduler* tråden specificeret antal pixels og gemmer disse pixel-værdier i *canvasBuffer*.

Appendiks B

Måledata fra OpenMP test med forskellige former for scheduling.

Schedule	Chunksize	Threads	Time [s]	Speed-up [%]	Effektivitet [%]
Dynamic		1	3,35	100	100
		2	2	168	84
		3	1,65	203	68
		4	1,3	258	64
		5	1,2	279	56
		6	1	335	56
		7	1	335	48
		8	1	335	42
		9	1	335	37
		10	1	335	34
	4	6	1	335	56
	8	6	1	335	56
	16	6	1	335	56
	32	6	1	335	56
	64	6	1,3	258	43
Static		1	3,35	100	100
		2	2,32	144	72
		3	2	168	56
		4	1,65	203	51
		5	1,65	203	41
		6	1,33	252	42
		7	1,3	258	37
		8	1,3	258	32
		9	1	335	37
		10	1	335	34
	4	6	1	335	56
	8	6	1	335	56
	16	6	1	335	56
	32	6	1	335	56
	64	6	1,3	258	43
Guided		1	3,35	100	100
		2	2	168	84
		3	1,5	223	74
		4	1,3	258	64
		5	1,2	279	56
		6	1	335	56
		7	1	335	48
		8	1	335	42
		9	1	335	37
		10	1	335	34

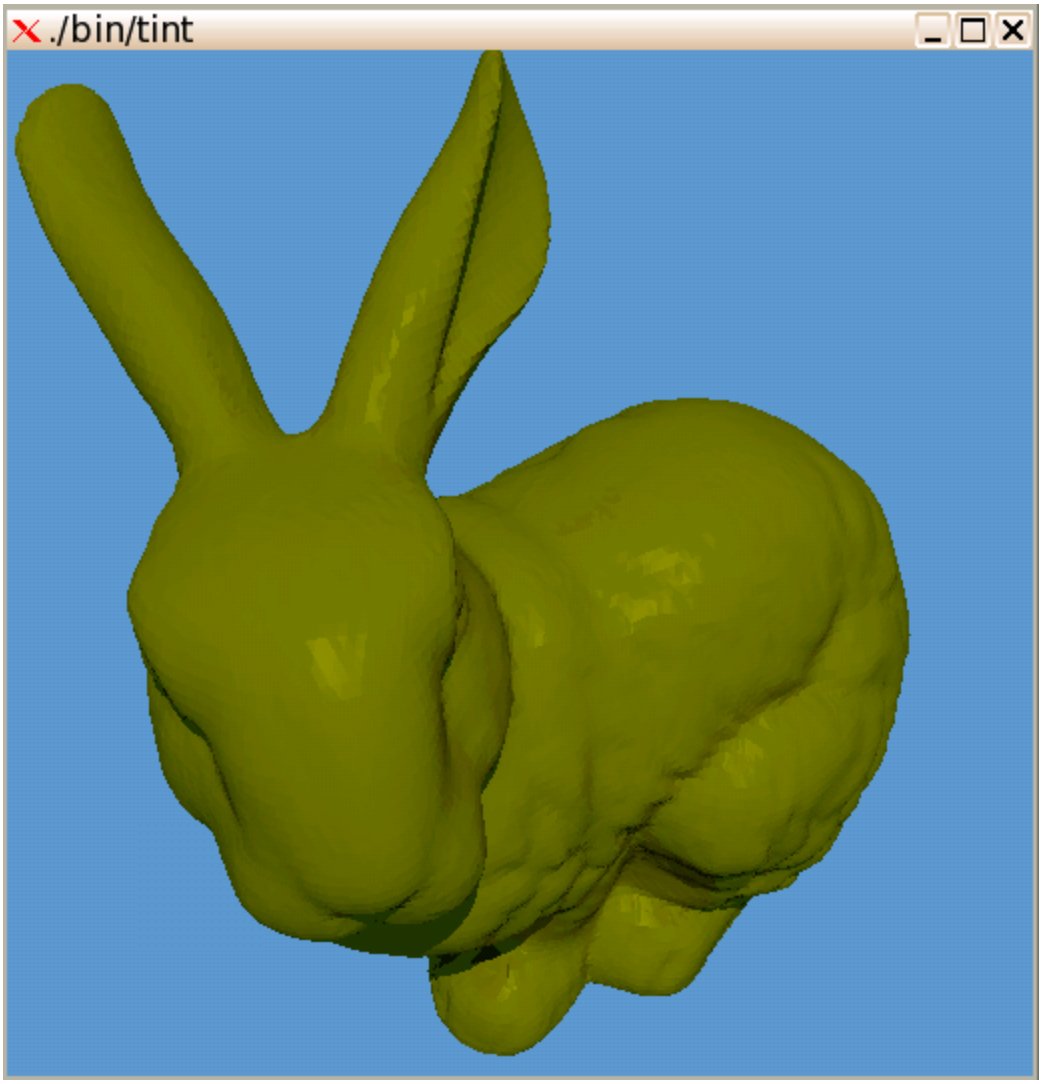
4	6	1	335	56
8	6	1	335	56
16	6	1	335	56
32	6	1	335	56
64	6	1,3	258	43

Appendiks C

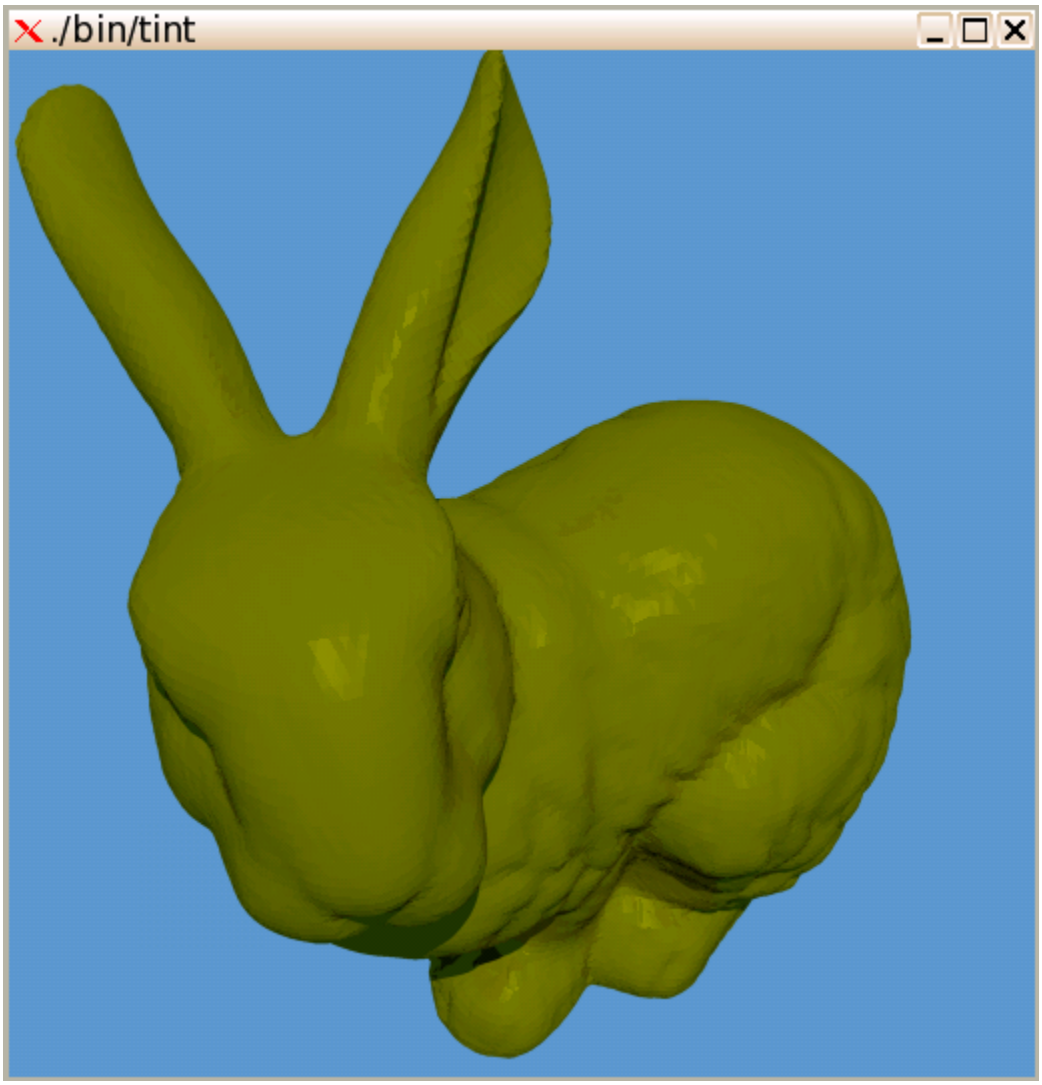
I dette appendiks kan der findes *screendump* fra en række scener samt en kort beskrivelse af hver. Renderingstiderne for de forskellige figurer svarer til dem der kan findes i resultat afsnittet i rapporten. Dog skal der lægges et par sekunder oven i, da der er anvendt supersampling på de fleste figurer. At navigere kameraet rundt så billedet bliver som ønsket er dog ganske uproblematisk, da programmet reagerer prompte på brugerinput. Man skal således ikke vente til et billede er færdigt, før man kan se figuren og navigere på kameraet.
På den vedlagte CD kan .obj samt .kdtree filer til figurerne findes. Se vejledningen i Appendeks D om hvordan disse filer anvendes.

Bunny

kD-træ bygget ud fra bunny.obj. Figuren består af 69.451 trekanter og kD-træet er bygget af 262.384 noder. Nedenstående to *screendumps* af bunny figuren viser forskellen på et billede renderet med og uden supersampling (forskellen er tydeligst i omridset af figuren). Der zoomet lidt ind og *tiled* lidt op, så figuren fylder hele billedet.



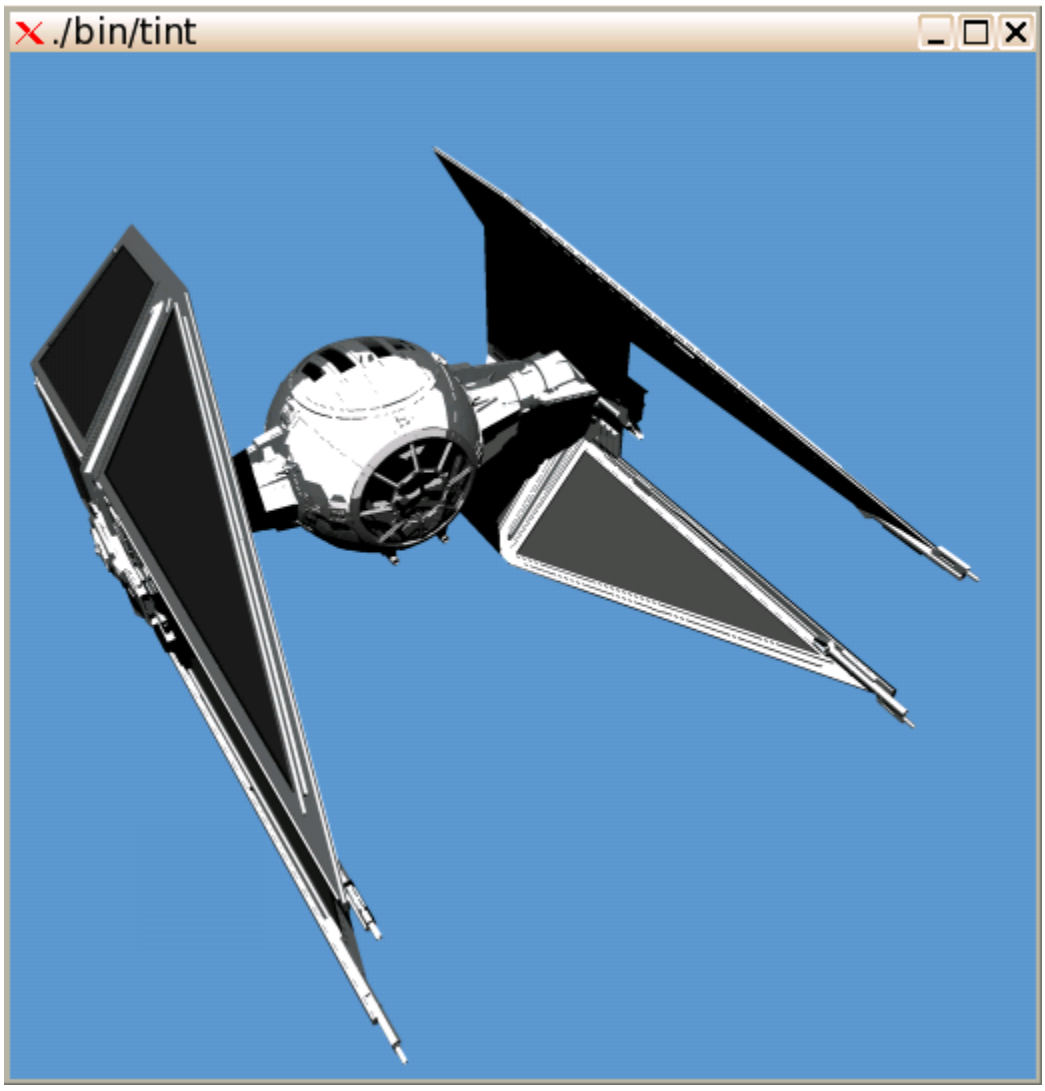
Figur 57: Bunny renderet uden supersampling.



Figur 58: Bunny renderet med supersampling.

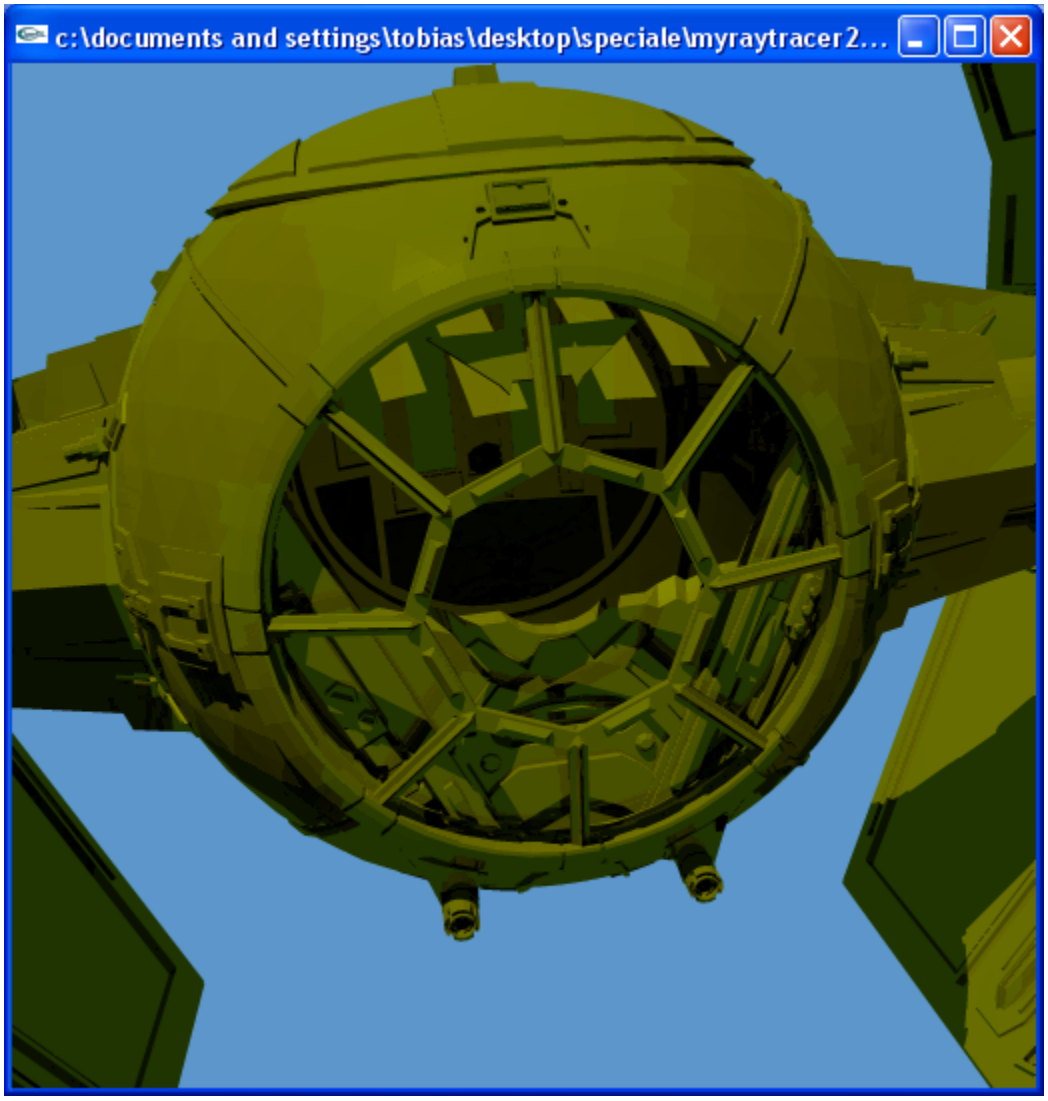
TieFighter

kD-træ bygget ud fra TieFighterMed.obj og TieFighterMed.mtl (overflade specifikationer). Figuren består af 34.786 trekanter. Kameraet er flyttet lidt rundt og der er zoomet en anelse ind, så figuren fylder hele billedet. Der er anvendt supersampling.



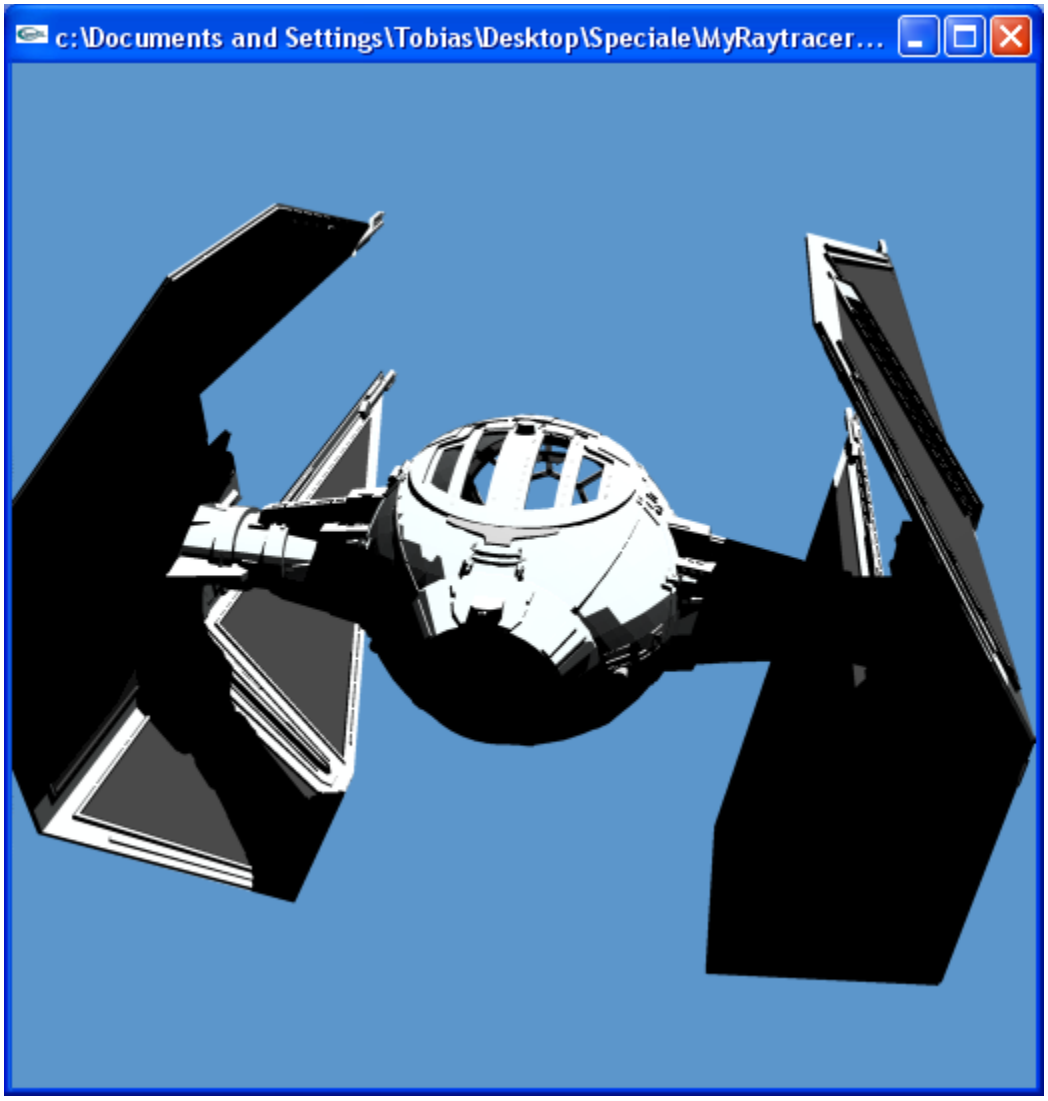
Figur 59: TieFighter.

Nedenstående figur er den same som den ovenfor. Her har figuren bare fået en gul reflektiv overflade og der er blevet zoomed ind på cockpittet.



Figur 60: TieFighter closeup.

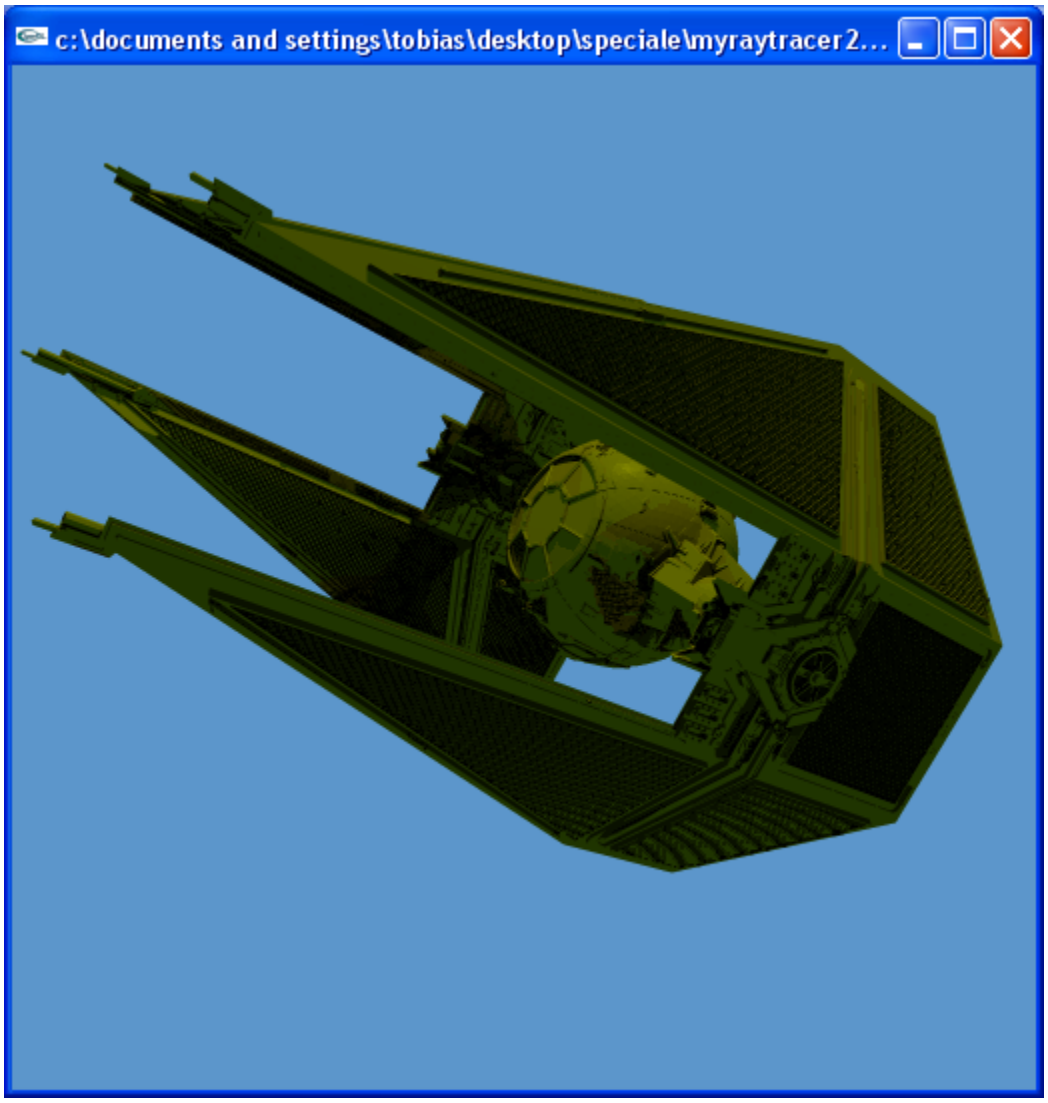
Igen samme figur. Denne gang er kameraet roteret rundt om figuren. Læg mærke til at man kan se igennem hullerne i taget på cockpiettet.



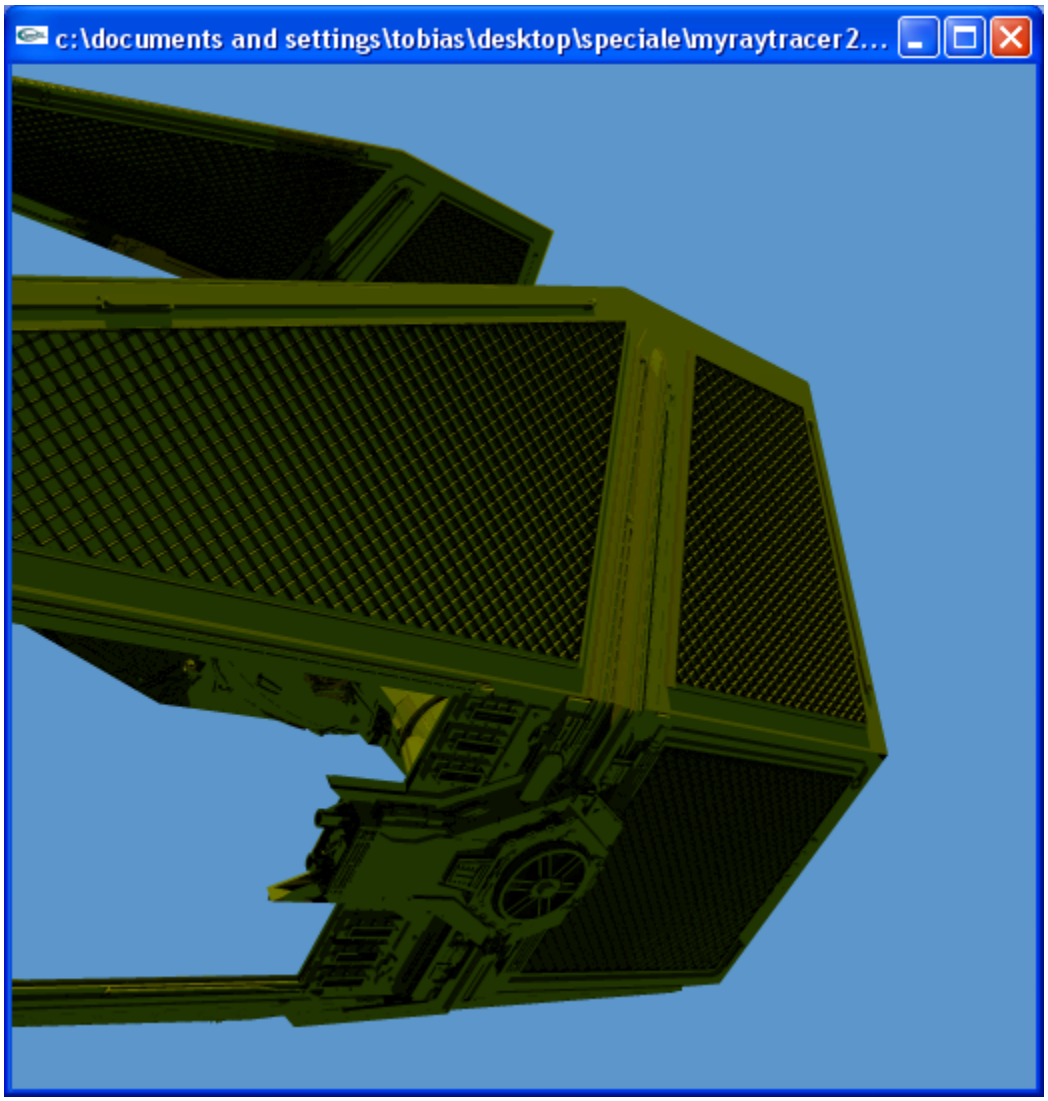
Figur 61: TieFighter bagfra.

TieIntercepter

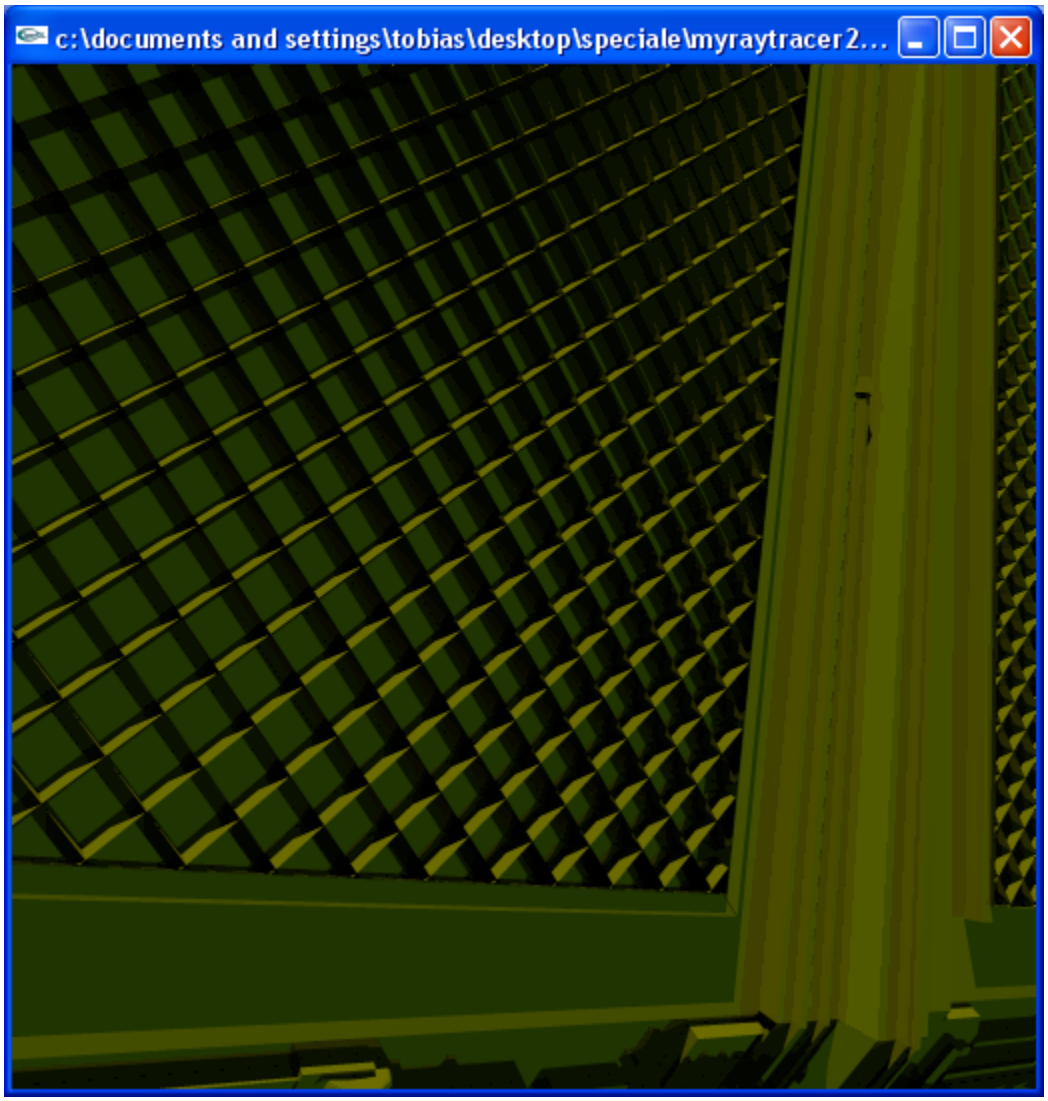
kD-træ bygget ud fra filen TieIntercepter.obj. Scenen består af 175.930 trekanter. Denne figur er ganske detaljeret, hvilket illustreres på de tre følgende *screendumps* hvor der zoomes ind på den ene vinge.



Figur 62: TieIntercepter fra siden.



Figur 63: Zoom in på TieIntercepter.



Figur 64: Zoomed helt ind på vingen af TieIntercepter.

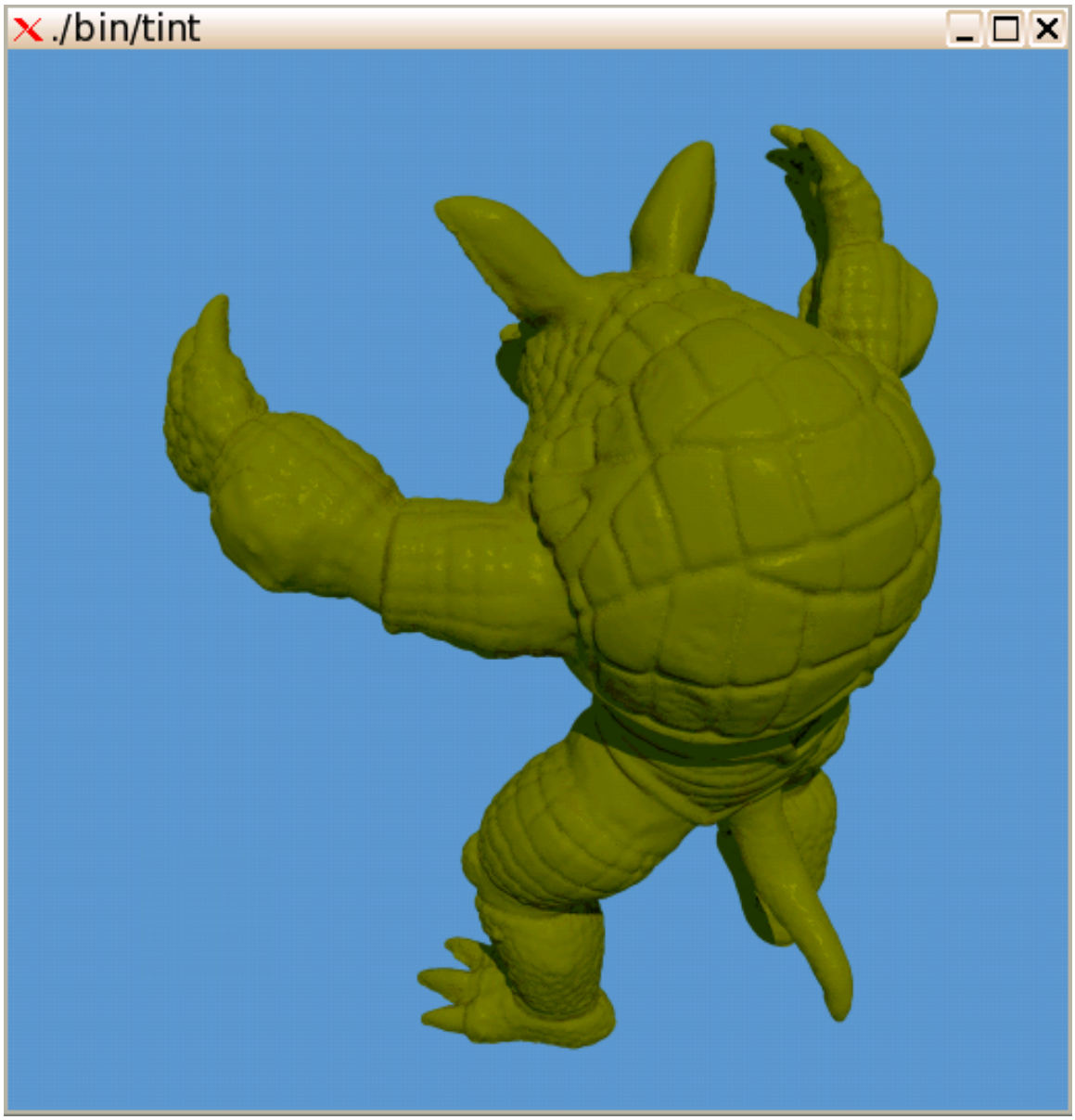
Bolter

kD-træ bygget ud fra filen bolter.obj og bolter.mtl (overflade specifikationer). Figuren består af 4.136 trekanter.

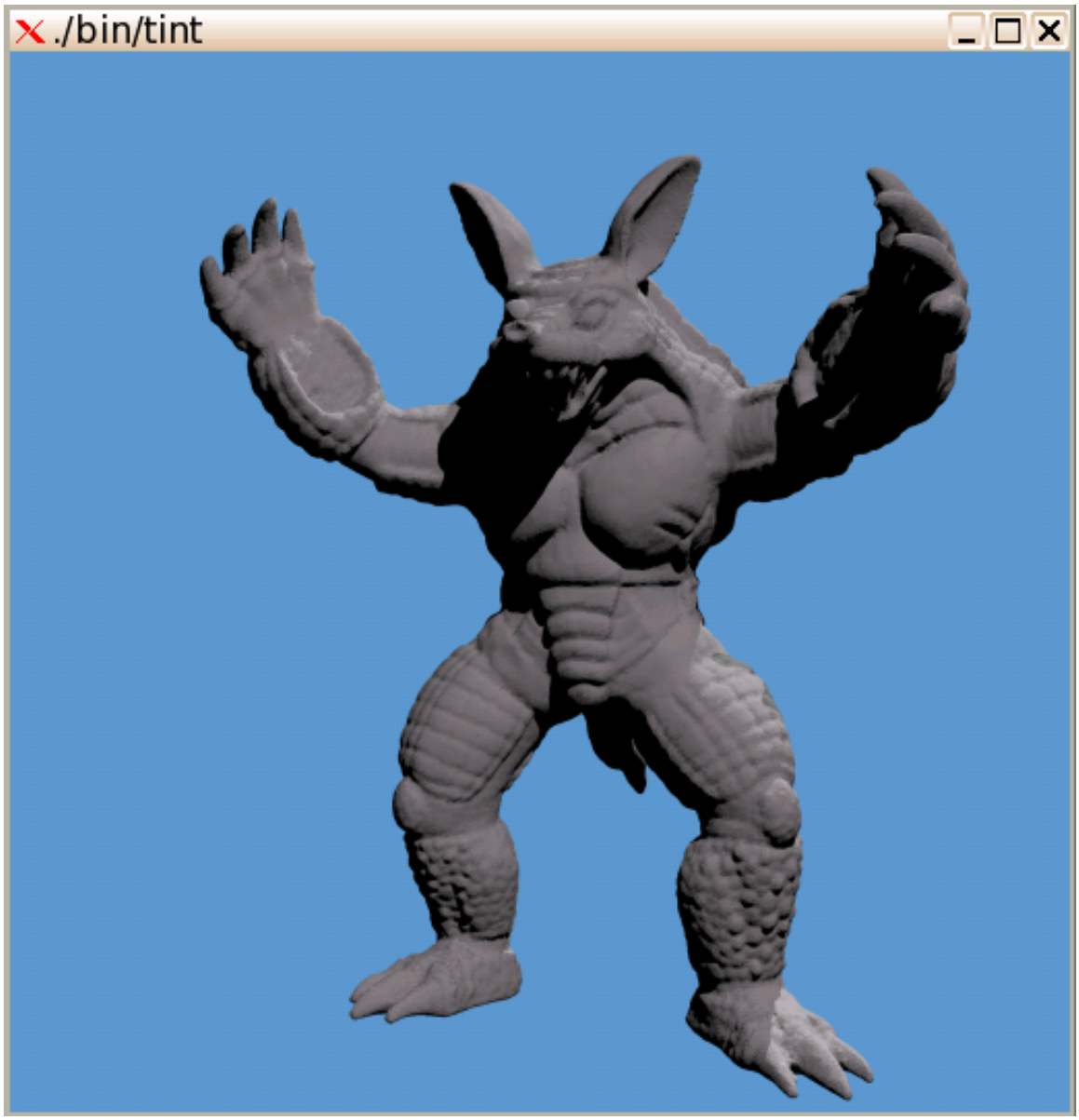
Figur 65: Bolter.

Armadillo_orig

kD-træ bygget ud fra armadillo_orig.obj. Figuren består af 345.944 trekanter, hvilket gør den til den mest komplekse figur i denne billedserie. Figuren ses fra forskellige vinkler og med forskellige overflader og lyskilder.



Figur 66: Armadillo, gul, en lyskilde.

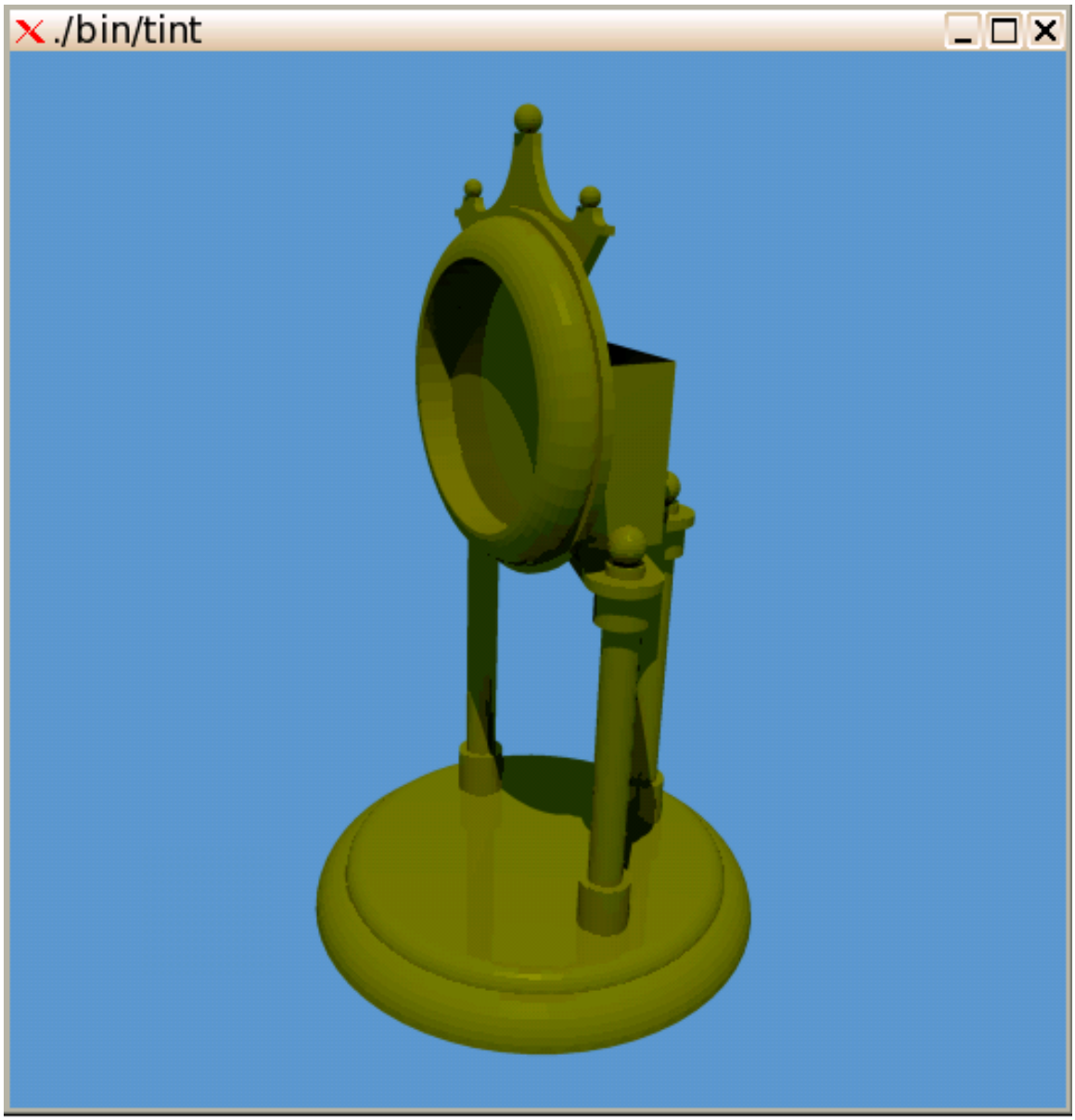


Figur 67: Armadillo, forfra, grå overflade og to lyskilder.

Figur 68: Armadillo, forfra, gul overflade og to farvede lyskilder.

Clock

kD-træ bygget ud fra filen si_clock.obj. Figuren består af 7.930 trekanter. Figuren er en del af en animeret ”dome clock” som jeg tegnede i ProEngineer og animerede i SoftImage i kurset ”Computer animation og modellering”.



Figur 69: En del af en “dome clock”.

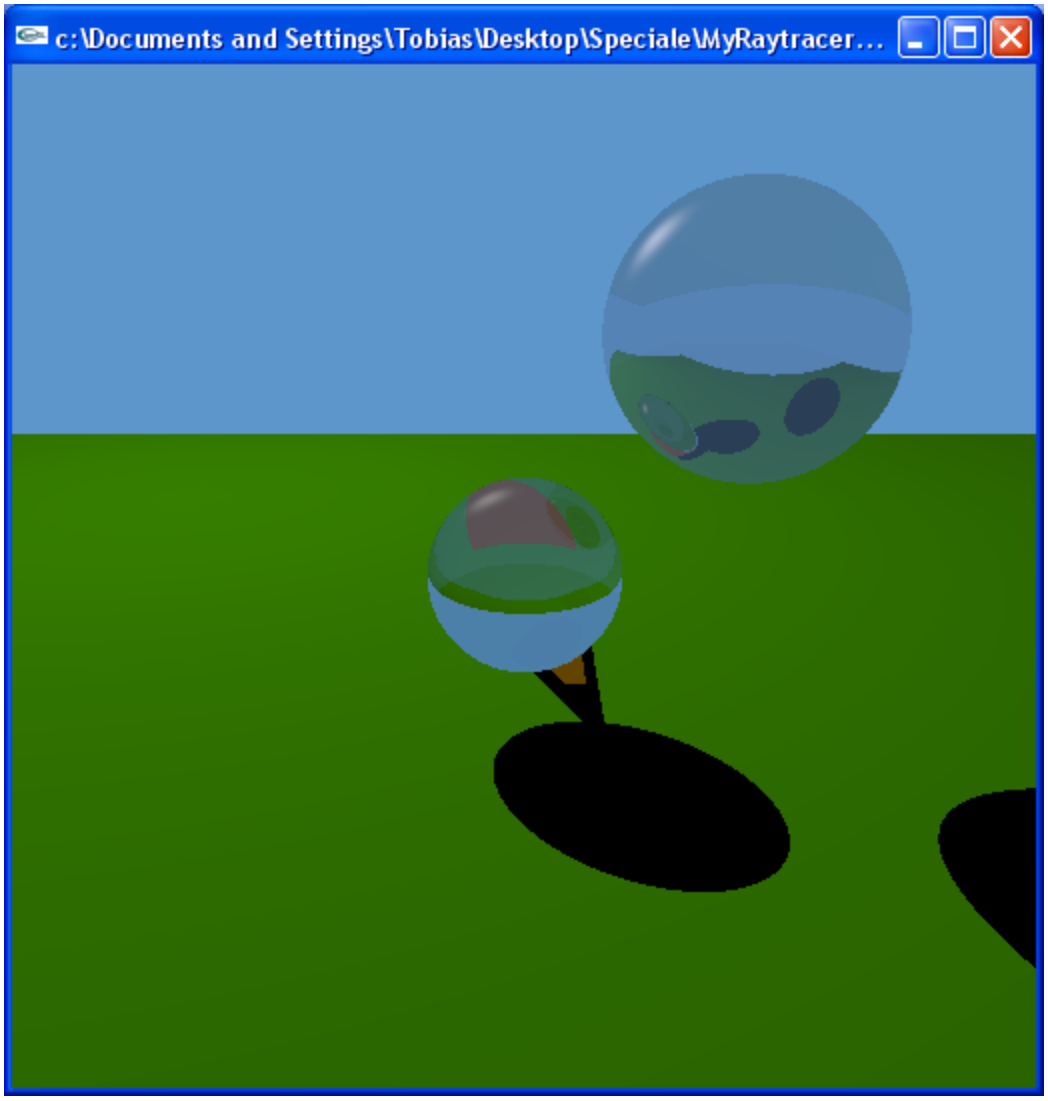
Kitchen

kD-træ bygget ud fra filen kitchen.obj og kitchen.mtl (overflade specifikationer). Figuren består af 181.755 trekanter. Lyskilden er placeret forkert i forhold til lampen i loftet.

Figur 70: Kitchen scenen. Med forkert placeret lyskilde.

Testscene

Manuelt opbygget testscene fra før understøttelse af kugler blev fjernet fra koden.



Figur 71: I midten en refraktiv kugle (en brun forvrænget trekant kan ses gennem kuglen) og oppe til venstre en reflektiv kugle.

Appendiks D

Kompilering

Projektet kan kompileres på Unix såvel som på Microsoft Windows. På CD'en kan der findes *make*filer til kompilering på Unix og projektfiler til Microsoft Visual Studio 2005.

Ved kompilering på Unix skal der benyttes *gmake*. I *headeren* for *Makefile* filen er det beskrevet hvilke argumenter der kan gives til *gmake*.

Eksekvering

Programmet kan eksekveres på Unix såvel som på Microsoft Windows. Dog benyttes en ikke dokumenteret og mindre effektiv metode til visuel rendering ved eksekvering på Microsoft Windows. Den primære løsning som bruger Pthreads til visuel rendering er således kun implementeret til Unix.

Programmet tager følgende argumenter:

a: aktiverer adaptive supersampling.
v: vælger visuel rendering.
s: bygger kD-træ ud fra specifikationer i *wavefront* fil angivet i det efterfølgende argument og gemmer dernæst kD-træet til fil.
l: loader kD-træ ud fra fil specificeret i det efterfølgende argument.

Eksempler:

Indlæs *wavefront* fil og render med visuel rendering og adaptiv supersampling:

./[program name] s [relative path to some *wavefront* file] v a

Indlæs kD-træ fil med visuel rendering men uden adaptive supersampling:

./[program name] l [relative path to some kD-tree file] v

Interaktion med scenen

Når programmet er startet kan der tilføjes og fjernes lyskilder, adaptiv supersampling og visual rendering kan slås til og fra, baggrundsfarven kan ændres, overfladerne kan ændres på objekterne i scenen og kameraet kan navigeres rundt. Hvordan dette gøres, beskrives i det følgende.

Navigation af kameraet:

Kameraet kan flyttes på følgende vis:

- Der kan *tiles* op, ned, til venstre og til høje ved tryk på henholdsvis *w*, *s*, *a* og *d*.
- Der kan zoomes ind og ud ved tryk på henholdsvis *z* og *x*.
- Der kan roteres op, ned til venstre og til højre rundt om centrum af scenen ved tryk på *8*, *5*, *4* og *6*.

Adaptiv supersampling

Ved tryk på *i* slås adaptiv supersampling til og fra.

Visuel rendering

Ved tryk på *o* slås visuel rendering til og fra.

Ny overflade

p skifter overfladen ud på samtlige trekanter i scenen. Der skiftes mellem et antal predefinerede overflader.

Indsæt lyskilde

l indsætter en ny lyskilde på en tilfældig position og med tilfældig farve i scenen.

Fjern lyskilde

k fjerner den sidst indsatte lyskilde.

Ny baggrundsfarve

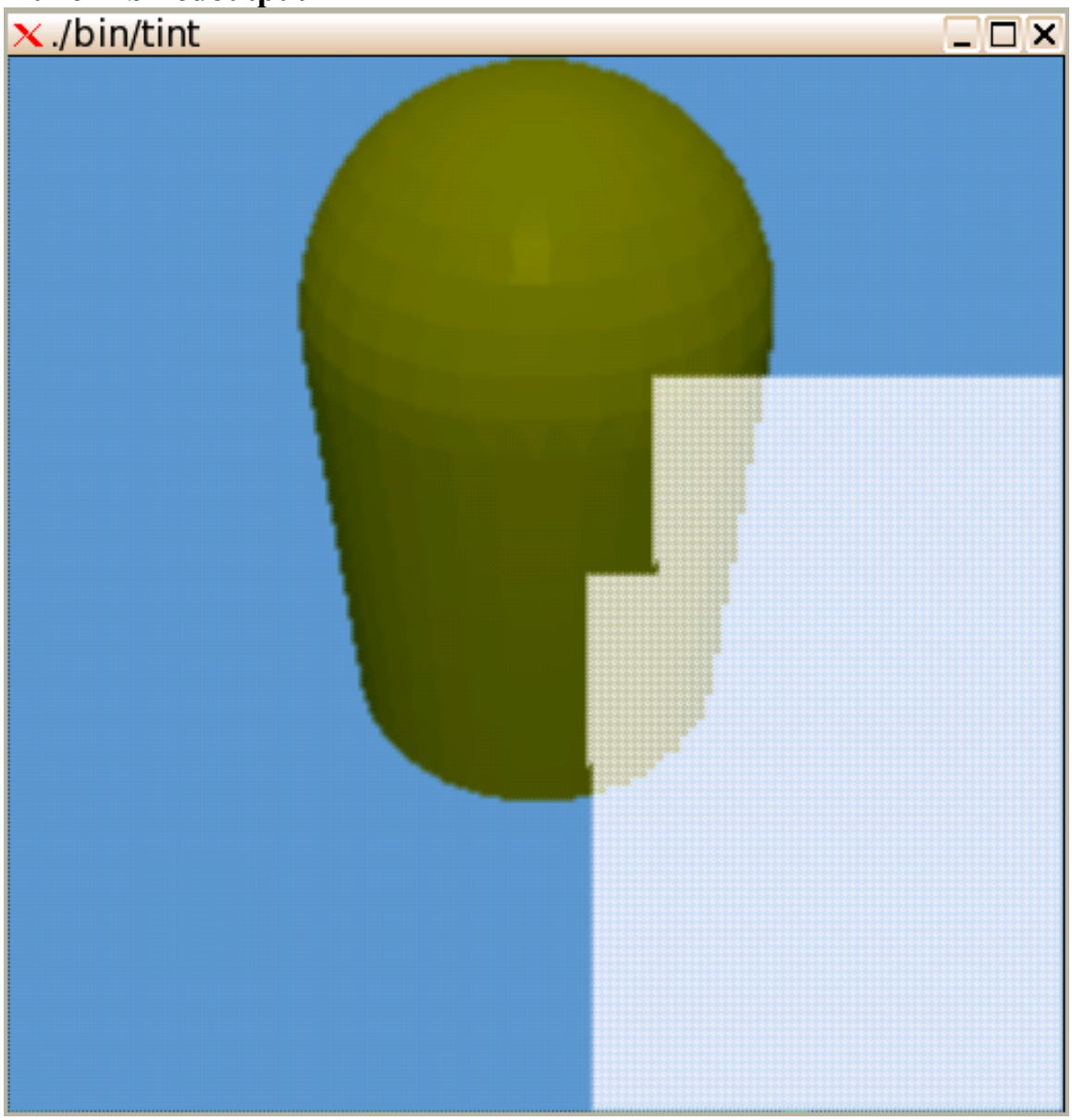
g ændrer baggrundsfarven til en tilfældig farve.

Appendiks E

Supersampling

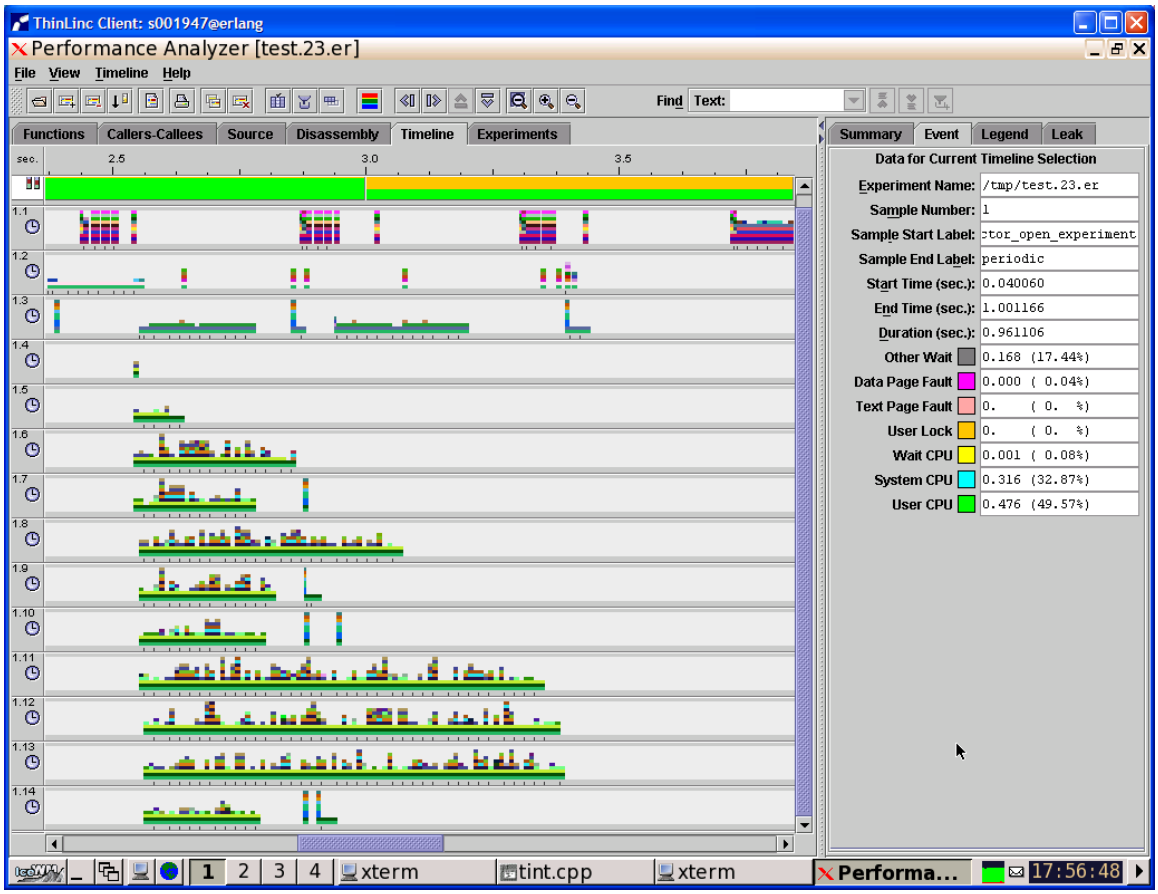
Desværre viste supersamplingen sig at være dårlig paralleliserbar. Jeg havde ikke forventet problemer på dette punkt, og tog det derfor for givet at det var en simpel sag at parallelisere supersamplingen ved brug af OpenMP. At det ikke gik som forventet kan måske skyldes rekursiviteten i supersamplingen.

Huller i billedoutput



Figur 72: Dome rendering. Viser mystisk fejl der nogen gange opstår.

Nogen gange sker det at firkantede områder i højre side af billedet ikke bliver færdigrenderet. Fejlen sker konstant på Erlang serveren men ganske sjældent på Bohr, selvom de to servere på papiret er identiske. Jeg har opdaget at Erlang distribuere arbejdet mellem trådene helt forkert (se Figur 73 i forhold til *timeline screendumps* i resultat afsnittet), hvilket sker fordi Erlang udregner nogle helt andre værdier ud fra *cost-bufferen* end Bohr. Jeg har ikke gransket problemet yderligere, men har i stedet holdt mig til at foretage tests på Bohr.



Figur 73: Viser den meget dårlige distribuering af arbejdet på Erlang serveren.