Voxelbasert 3D visualisering i OpenGL

Bjørn Egil Jenssen

Universitetet i Oslo

27/04-03

Forord

Denne rapporten oppsummerer mitt arbeid med Cand. Scient graden i informatikk ved Universitetet i Oslo, studieretning matematisk modellering.

Arbeidet startet i januar 2001 og ble fullført i april 2003. Hoveddelen av programmeringen og skrivingen er utført ved Universitetet i Oslo.

Hovedoppgaven tar for seg volumrendering med OpenGL i forbindelse med de forskjellige anvendelsesområdene som 3D-modellering og visualisering blir brukt. Oppgaven fokuserer spesielt på voxelrendering, dens fordeler og ulemper, samt implementasjon og testing av resultater.

Generelt egner voxelrendering seg til all visualisering der man har et 3D skalar- eller vektorfelt. Oppgaven drøfter også hvordan man lager en hardwareakselert voxelrenderer i OpenGL.

Jeg vil i tillegg se på muligheter, fordeler og ulemper med denne teknikken, samt drøfte hvilke krav som må stilles til hardware for å løse oppgavene effektivt. Avslutningsvis vil jeg sammenligne min OpenGL renderer med andre offentlig tilgjengelige voxelrendrer.

Jeg vil gjerne takke min veileder Hans Petter Langtangen for all støtte, hjelp og veiledning med arbeidet. Jeg vil gjerne også takke Trond Gaarder og Ståle Waage Pedersen for deres hjelp.

Oslo april 2003

Bjørn Egil Jenssen

Innholdsfortegnelse

1. Innledning....6
1.1 Innledning....6
1.2 Problemet....7
2. Visualiseringsteori....9
2.1 Koordinater og transformasjoner....9
2.2 Translasjon....10
2.3 Rotasjon....11
2.4 Skalering....12
2.5 Skjær....13
2.6 Projeksjoner....14
2.7 Lys og refleksjonsmodeller....16
3. OpenGL....20
3.1 Introduksjon....20
3.2 Oppsett og bruk av OpenGL....21
3.3 OpenGL vs DirectX (Direct3D)....26
3.4 Qt....27
3.5 Renderings teknikker i OpenGL....27
3.6 Fargetabell....30
3.7 Konklusjon....30
4. 3D visualisering av numeriske beregninger med Voxelrenderer....32
4.1 Volumrenderer....32
4.2 Voxelrender Algoritmen....33
4.3 Shading i volum visualisering....36
4.4 Visualisering av vektorfelt....37
4.5 Kutteplan....39
4.6 Fargetabell....41
4.7 Animasjon....42
4.8 Volum visualisering - begrensninger....42
4.9 Implementasjon....43
4.10 Praktiske anvendelser av voxelrendering....44
4.11 Konklusjon....45
5. VTK vs OpenGL....46
5.1 VTK....46
5.2 VTK vs OpenGL....46
5.3 Konklusjon....50
6 Simulering og datasett....51
6.1 Simulering av 3D bølgeligning....51
6.2 Datasett....51
6.2 HDF....51
7. Konklusjon....53
Referanser....55

1. Innledning

1.1 Innledning

I denne hovedfagsoppgaven har jeg sett på bruken av voxelrendering av numeriske beregninger. Målet med oppgaven har vært å utvikle en voxelrenderer i OpenGL fra bunnen av, til et funksjonelt program som kan visualisere forskjellige 3D datasett generert av f.eks. Diffpack. Datasettene er skalarfelt som visualiseres ved å assosiere gjennomsiktighet og farge til en voxel (3D piksel), til hver av skalarverdiene i feltet.

Det finnes flere forskjellige måter å visualisere voxler på, jeg har hovedsakelig brukt en metode som benytter teksturer mappet til 2D-plan som settes sammen til et 3D voxelsett. Denne metoden gir støtte for hardware akselerasjon. En beskrivelse av voxelrendereren jeg har laget dekkes i kapittel 4.

Voxelrendering er et bra alternativ til en av de vanligste måten å visualisere 3D felt på, nemlig ved hjelp av isoflater. Denne teknikken går ut på å bryte feltet ned til enkle primitiver som triangler, for så å bygge opp en lukket flate rundt en gitt skalarverdi. Denne metoden gir også støtte for hardware, men har flere problemer som ofte oppstår, og som jeg skal vise senere, ofte ikke representerer dataene like godt som voxelrendering kan. Alternative visualiseringsmetoder tar jeg for meg i kapittel 5.

Jeg har brukt OpenGL som verktøy i utviklingen av voxelmotoren. OpenGL er et lavnivå hardwareakselerert grafikkbibliotek og blir dekket i kapittel 3. Voxelrendereren er programert i C++.

Voxelrendreren har blitt testet med forskjellige data. Noe av dataene er simulert med Diffpack, og noe er datafiler skaffet fra Internett. Dataene og simuleringene blir dekket i kapittel 6.

For å kunne oppnå et effektivt visualiseringssystem som kan håndtere store datasett i sanntid, er man avhengig av datasystemer med muligheten til å ha en stor dataflyt. CPU'ene i moderne datamaskiner har blitt meget kraftige, men ikke raske nok til å kunne gi nok interaktivitet alene i disse tilfellene. Derimot er ny 3D hardware ypperlig for slik visualisering. I løpet av de siste årene har bruken og utviklingen av dedikert 3D hardware eksplodert, akselerert av den sterkt voksende spillindustrien. Disse skjermkortene har et relativt lite antall av mulige operasjoner, men som er dedikert til 3D visualisering. Operasjoner som blir håndtert av 3D hardware er blant annet polygon data, fargetabeller, transformasjon og lyssetting samt tekstur data. Slik hardware ønsker jeg å utnytte i denne oppgaven.

For å få maksimal utnyttelse av de forskjellige 3D hardware kortene som finnes, kan man benytte seg av forskjellige grafikkbibliotek og programvare. For PC'er finnes det to lavnivå grafikkbibliotek man kan benytte seg av. OpenGL, først utviklet av Silicon Graphics, som jeg benytter meg av i denne hovedoppgaven, og DirectX utviklet av Microsoft som kun kan benyttes under Microsoft Windows og til spillkonsolen X-Box. OpenGL er et kraftfullt software bibliotek som utnytter moderne grafikk hardware. Gjennom OpenGL kan det meste av grafikkortenes funksjonalitet brukes. OpenGL er derimot et lavnivå bibliotek og krever mye kunnskap på et grunnleggende nivå for å kunne brukes til å visualisere kompliserte objekter og scener. Flere høynivå grafikk programvare slik som Open Inventor og VTK derimot er basert på OpenGL, og gir en enklere tilgang til verktøyene man trenger slik at man kan lettere og raskere visualisere det man ønsker, mot et viss tap av ytelse.

Voxelrendering har blitt en stadig mer populær visualiseringsmetode. En av årsaken til dette er at det har vært stor utvikling av teknisk utstyr som kan generere volumdata. Dette har ledet til økt etterspørsel for mer effektive og kraftfulle verktøy for å visualisere og tolke de observerte dataene. Det har spesielt vært en revolusjon innen medisin, der MRI (Magnetic Resonance Imagers), CT (Computer Tomographs) og PET (Positron Emission Tomograph) spiller en viktig rolle som et diagnoseverktøy for mange forskjellige sykdommer. Astrofysiske, meteorologiske og geofysiske målinger, samt datasimuleringer som benytter finite element modeller av fluider, materialpåkjenninger osv. genererer også 3D volumdatasett.

1.2 Problemet

Målet med vitenskapelig visualisering er å gjøre om numeriske data til en visuell form som gir muligheten til å oppdage relevant og verdifull informasjon fra de gitte data. Når data blir konvertert til visuell form har menneskets sanser, spesielt synet og hjernen en meget kraftfull egenskap til å kunne identifisere komplekse objekter og samtidig komprimere og trekke ut viktig informasjon visuelt. Når dette blir gjort på rett måte kan viktig informasjon oppdages nesten øyeblikkelig.

Høy grad av interaktivitet og rask respons er meget viktig for at visualiseringen skal oppnå best mulig effekt. Det vil si å kunne raskt flytte og roterere på objektene i scenen. Dette er viktig for at man skal kunne se de romlige aspektene ved visualiseringen. Hvis en scene er for komplisert og bevegelsene blir for trege, vil vårt visuelle minne, som er tidsmessig veldig kort, glemme hva som ble vist på forrige bilde. I en slik situasjon kan vi miste verdifull informasjon. Derfor er det viktig at en voxelrenderer tar i bruk moderne 3D hardware, slik at ytelsen blir høy, og stor grad av interaktivitet blir mulig.

I tillegg til viktigheten av å kunne bevege og rotere objektene i scenen i sanntid, er det også nyttig å kunne bevege kutteplan rundt med samme hastighet. Kutteplanet skjærer vekk alt som er på den ene siden av planet, og lar resten være igjen. Dette planet bør man kunne flytte og rotere rundt en vilkårlig akse i sanntid. Andre ting som også vil være nyttig å kunne forandre på hurtig, er blant annet lyskilder og farger. Innen voxelgrafikk er det viktig å kunne forandre på farger og gjennomsiktighet med øyeblikkelig virkning.

Riktig bruk av farger kan forbedre den visuelle informasjonen i en stor grad. Det er derfor viktig å ha en god fargetabell som kan benyttes til å endre farger og gjennomsiktighet i datasettet. Når man opererer med farger, er det viktig å tenke på hvordan det menneskelige øyet oppfatter fargene. Øynene har visuelle reseptorer som er følsomme for rødt, grønt og blått lys, og alle farger kan genereres ved å blande disse fargene.

Et grafisk brukergrensesnitt er også viktig for interaksjon med et datatasett. Man må ha muligheten til å zoome, rotere, samt manipulere datasettet med slidere, knapper, input bokser, tastatur og musbevegelser. For å løse dette problemet har jeg tatt i bruk Qt og The OpenGL Utility Toolkit (GLUT) som er verktøy for håndtering av brukergrensesnitt.

I denne oppgaven ser jeg på metoder og strategier for å visualisere og interaktere med volumer på en effektiv måte ved å benytte meg av hardware som er vanlig i dagens maskiner. Jeg ser på fordelene og begrensingene en slik hardware avhengig fremgangsmåte tilbyr, samt hva som kreves for å utnytte hardware på et lavt nivå.

2. Visualiseringsteori

Senere i oppgaven vil voxelrenderings teknikken bli forklart, men først vil jeg se på hva som ligger i grunnen for all 3D visualisering, og hva som ligger bak kommandoene i OpenGL. I dette kapittelet ser jeg på hva som ligger bak kommandoene som roterer og beveger på objekter, samt hvordan lys og shading fungerer. Dette er ikke nødvendig å forstå for å bruke OpenGL eller annen 3D visualisering verktøy, men det er nyttig å ha en oversikt over dette dersom man ønsker å ha maksimal kontroll i en 3D scene. Dersom man ønsker å manipulere en 3D scene på et lavere nivå ved å gå direkte inn å manipulere de forskjellige matrisene som OpenGL benytter seg av, må man kunne dette. Dette blir blant annet gjort i voxelrendereren når jeg laget rotasjon av scenen via muskontroll.

2.1 Koordinater og transformasjoner

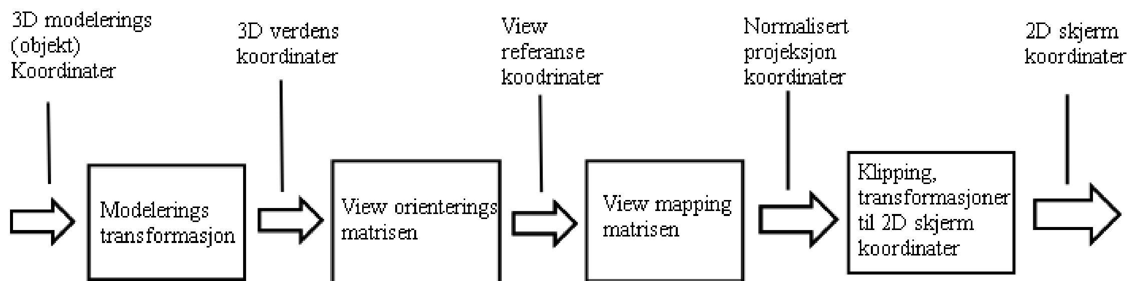
For å få en 3D scene til å vises på en 2D skjerm er man nødt til å forholde seg til forskjellige koordinatsystemer og transformasjoner, samt projeksjoner for å overføre 3D objekter til 2D slik at de kan visualiseres på skjermen. Typiske koordinatsystemer man bruker er:

- Objektkoordinater
- Verdenskoordinater
- View koordinater
- Normaliserte projeksjonskoordinater
- 2D skjermkoordinater

Mellom disse koordinatsystemene bruker man følgende transformasjoner:

- Modelleringstransformasjon
- View orienterings matrise
- View maping matrise
- Klipping til 2D skjermkoordinater

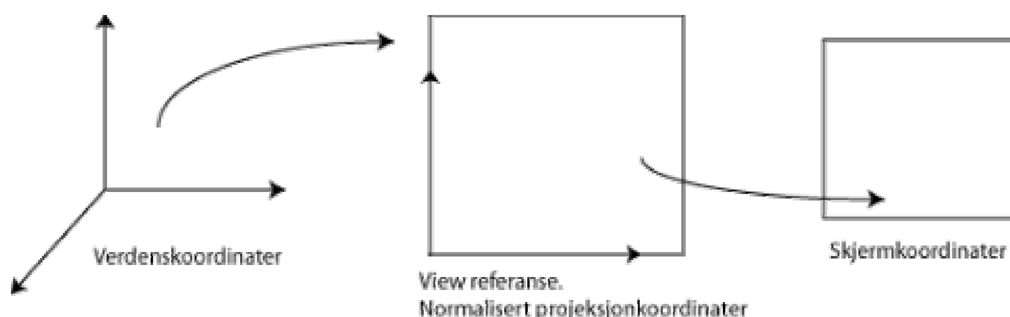
Koordinatene og transformasjonene man trenger for å overføre et modellert lokalt 3D objekt til 2D og som resulterer i bildet på skjermen er vist i figur 1.



Figur 1: Koordinater og transformasjoner som trengs for å overføre en 3D scene til et 2D skjermbilde

Alle objekter i en scene blir definert og bygget opp i sitt eget 3D modellerings koordinatsystem. Objektene plasseres så i 3D verdens koordinatsystemet ved å gjøre modellerings transformasjoner på de, dvs translasjon, rotasjon og skalering.

I tillegg til at de geometriske posisjonene må defineres, må det samme gjøres med lys og kameraposisjonene. Hvordan en scene blir seendes ut er avhengig av disse transformasjonene. Deretter defineres view orientasjons matrisen, eller kameraet som vi bruker til å se på scenen med. Videre må projeksjons koordinatene som skal brukes bestemmes, enten om det skal være en ortogonal projeksjon eller perspektiv projeksjon. En projeksjons- og klippetransformasjon blir til slutt gjort for å generere et 2D bilde av scenenen som igjen kan mappes til skjermen. Kort oppsummert i figur 2.

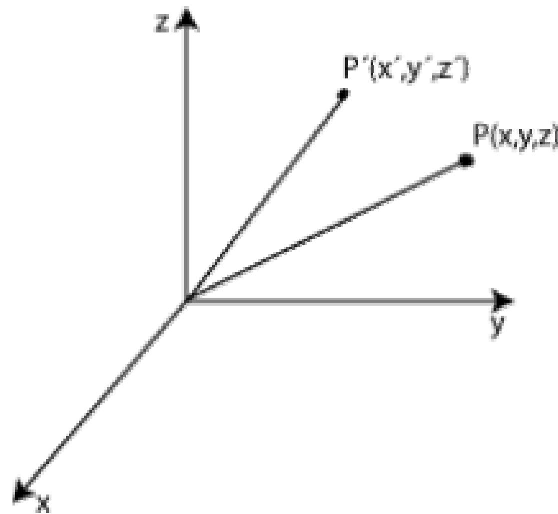


Figur 2: Verdenskoordinater blir omgjort til skjermkoordinater

2.2 Translasjon

Translasjon er operasjonen som utføres når et objekt beveger seg fra ett punkt til ett annet. Matriserepresentasjonene for translasjon er som følger:

$P' = T + P$
$P' = S * P$
$P' = R * P$



Figur 3: Translasjon av et punkt

Hvor T er translasjonsvektoren, S er skaleringsmatrisen og R er rotasjonsmatrisen. Deres 2D matriserepresentasjon er som følger:

$$T = \begin{pmatrix} d_x \\ d_y \end{pmatrix}, \quad S = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}, \quad R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

For å oppnå en enkel matriseoperasjon for disse tre transformasjonene, blir en 3 komponent vektorformulering brukt med en 3x3 matrisenotasjon for 2D operasjoner. Ved å bruke denne formuleringen kan translasjon, skalering og rotasjon uttrykkes slik:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \left[\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \right] \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Antallet aritmetiske operasjoner for denne 3x3 matrisen er høyere på en vanlig CPU sammenlignet med 2x2 matrisen pluss vektoraddisjons operasjonen. Fordelen ved å bruke 3x3 matriser ser man dog når man benytter seg av spesiell grafikk hardware for å utføre operasjonen. I slik hardware blir de aritmetiske operasjonene utført i parallell. Hver parallelle operasjon kan utføre multiplikasjons og adderings operasjoner brukt i 3x3 matriser.

2.3 Rotasjon

For å rotere et objekt rundt et punkt P=(x1, y1) i 2D, kan transformasjonene bli utført ved først å flytte objektet til origo ved å trekke fra (-x1, -y1), for deretter å gjøre rotasjonen, og så flytte objektet tilbake ved å addere (x1, y1). Dette kan gjøres ved en enkel matriseoperasjon.

$$\begin{pmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{pmatrix}$$

I 2D ser den generelle rotasjonsmatrisen slik ut:

$$M = \begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ 0 & 0 & 1 \end{pmatrix}$$

Dette kan utvides til 3D ved å benytte seg av 4x4 matriser. Rotasjon rundt X, Y og Z aksen kan da bli gjort med følgende matriser.

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

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

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

Alle tre matrisene gir positive rotasjoner for positive θ i forhold til høyrehånds regelen for x, y og z aksene.

Translasjoner og rotasjoner er eksempler på solide transformasjoner, dvs. transformasjoner som ikke forandrer på størrelsen eller formen til objektet.

2.4 Skalering

Skalering kan tilordnes alle de 3 aksene uavhengig av hverandre, og er et eksempel på transformasjoner som endrer på størrelsen og formen til objekter.

Hvis vi sender punktet (x, y, z) til det nye punktet:

$$x' = \beta_x x, \quad y' = \beta_y y, \quad z' = \beta_z z$$

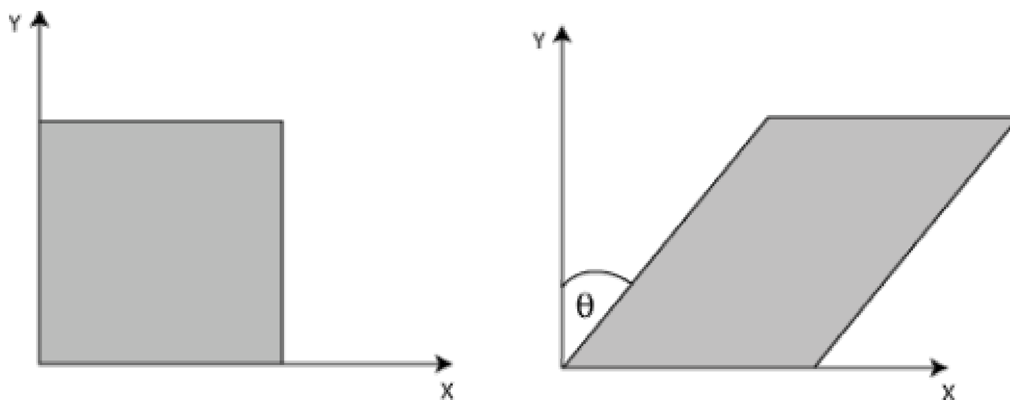
Den tilhørende skaleringsmatrisen blir da:

$$S = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dersom alle skaleringsfaktorene er like kalles skaleringen unifrom, objektet beholder sin form, men forandrer sin størrelse. Hvis ikke er skaleringen ikke-uniform og objektet blir deformert.

2.5 Skjær

Skjær er en deformasjon av et objekt langs en akse.



Figur 4: Skjær

Et skjær langs x aksen er definert slik:

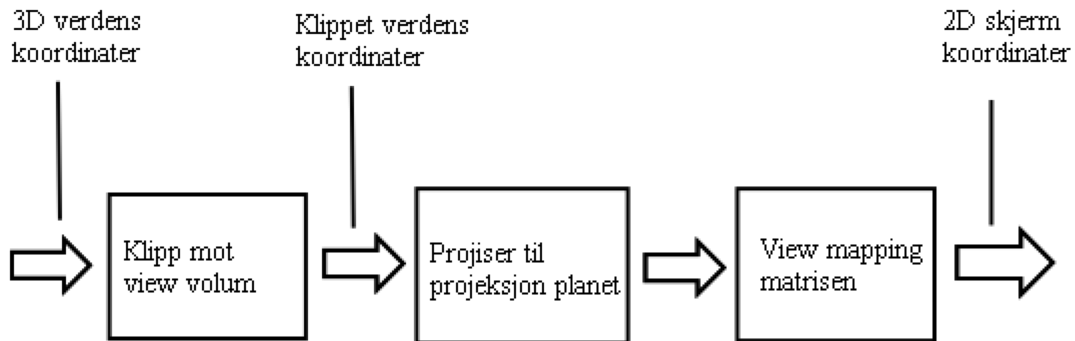
$$x' = x + (\cot\theta)y, \quad y' = y, \quad z' = z$$

Den tilhørende skjær matrisen blir da:

$$H = \begin{pmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

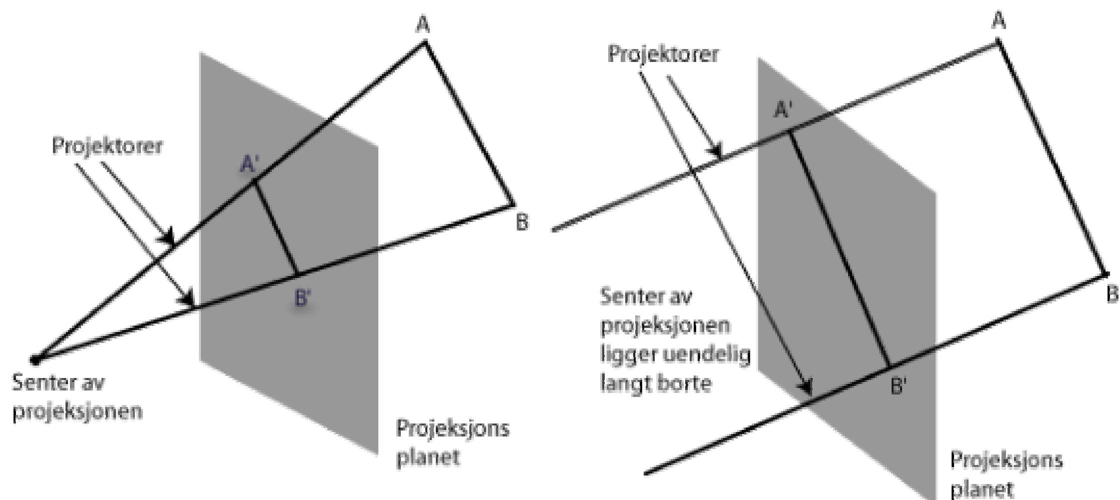
2.6 Projeksjoner

I 3D visualisering må det defineres et volum av scenen som skal bli visualisert, kalt view volumet. Objekter som befinner seg utenfor dette volumet vil ikke bli visualisert. Objekter i 3D rommet blir klippet mot 3D view volumet for så å bli projisert. Innholdet av projeksjonen av view volumet på projeksjonsplanet blir mappet til viewporten for å kunne vises på skjermen.



Figur 5: Projeksjoner

Generelt vil projeksjonstransformeringer omforme punkt fra et N dimensjonalt rom, til en dimensjon mindre enn N. Projeksjonen av et 3D objekt er definert av rette projeksjonslinjer som kommer fra midten av projeksjonen, passerer gjennom hvert punkt på objektet og skjærer projeksjonsplanet som vist på figur 6.

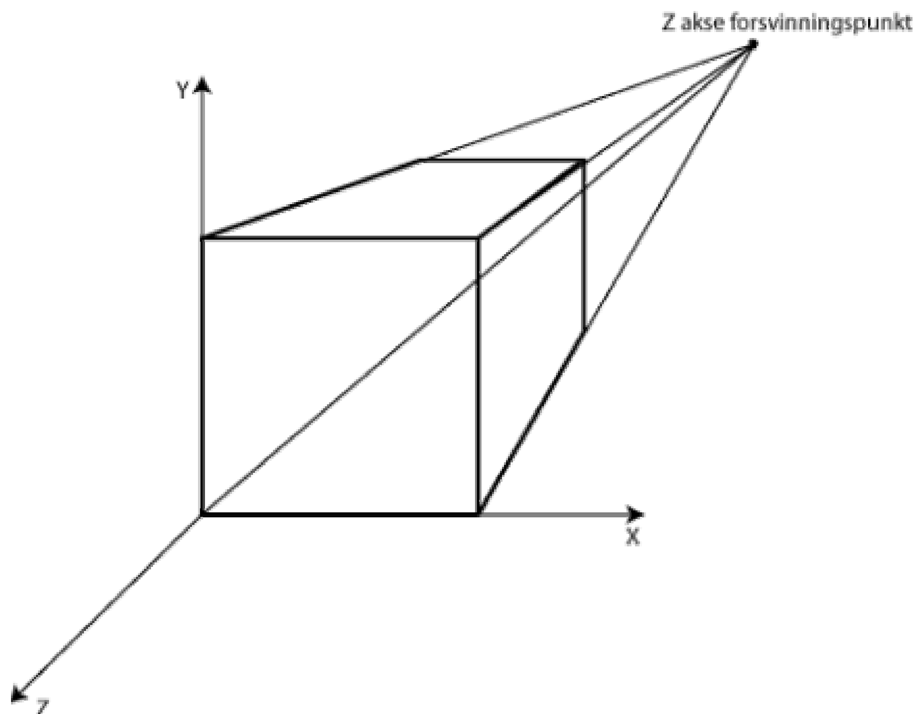


Figur 6: Perspektiv og parallell projeksjon

Til venstre vises en perspektiv projeksjon av linjen AB til A'B', og til høyre er en parallell projeksjon vist. Projeksjonslinjene blir parallelle i figuren til høyre, dvs. AA' og BB' er parallelle.

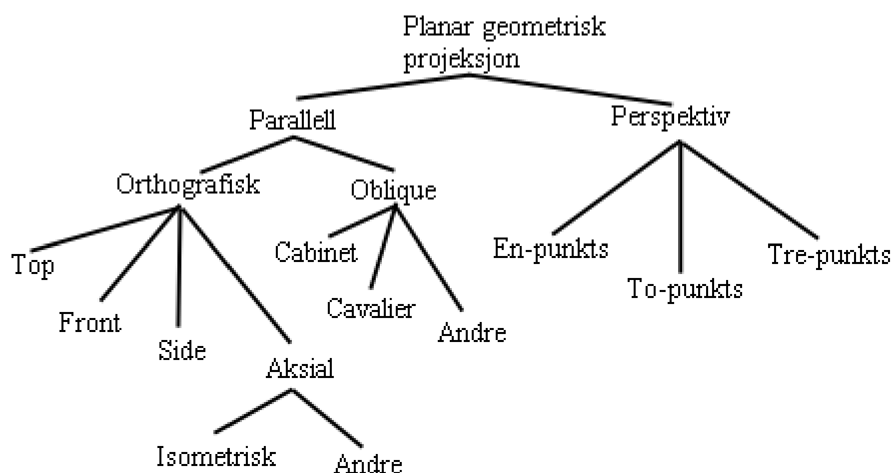
Ved perspektivprojeksjon vil alle linjer som ikke er parallelle konvergere i et forsvinningspunkt som vist i figur 7. I 3D vil parallelle linjer kun møtes i en avstand uendelig langt borte.

Perspektivprojeksjoner blir kategorisert ved antall forsvinningspunkter. Det kan være ett, to eller tre forsvinningspunkter i 3D. Vanligvis blir ett eller to forsvinningspunkt brukt. Bruk av tre forsvinningspunkter blir regnet for å gi lite ekstra realisme i forhold til to punkts perspektiv.



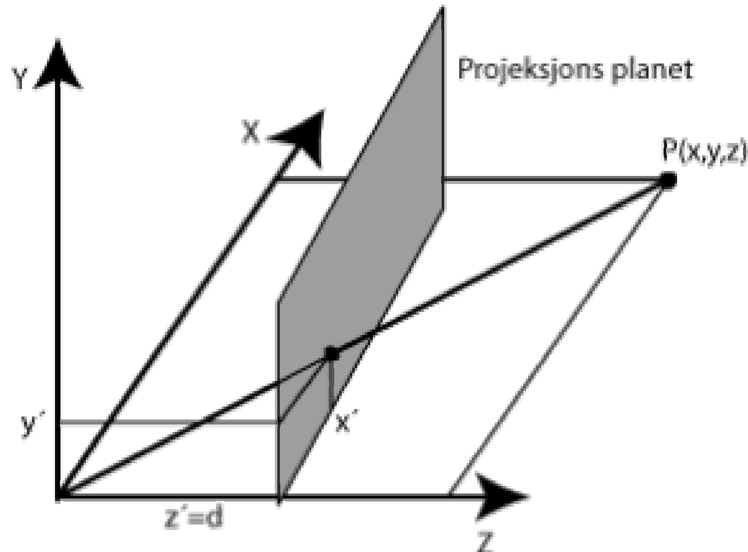
Figur 7: Ett forsvinningspunkt

Det finnes en hel del forskjellige typer geometriske projeksjoner. De viktigste kan klassifiseres som vist i figur 8.



Figur 8: Geometriske projeksjoner

Matematikken bak planar geometrisk projeksjon er som følger: For enkelthets skyld sier vi at projeksjonsplanet er vinkelrett på z-aksen og krysser z-aksen i punktet z=d. Projeksjonen kan defineres av en 4x4 matrise. Gitt et punkt P(x,y,z) blir projisert til punktet P'(x', y', z') ved z'=d, som vist i figur 9.



Figur 9: Planar geometrisk projeksjon

Fra figuren ser vi at x'/d=x/z og y'/d=y/z som kan skrives slik: x'=(x*d)/z og y'=(y*d)/z. Transformasjonen kan gjøres ved å bruke denne matrisen.

$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} = M_{per} \cdot P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Som gir (X, Y, Z, W) = (x, y, z, z/d) transpondert. Deler vi på W får vi projeksjonkoordinatene i 3D rom: (X/W, Y/W, Z/W) = (x',y',z') = ((x*d)/z, (y*d)/z, d)

Alternativt kan man flytte projeksjonsplanet til origo og senteret til projeksjonen til z=-d, da får vi følgende resultat: x' = x/((z/d +1), y' = y/((z/d) +1). I dette tilfellet finner vi perspektivmatrisen. Ved å la $d \to \infty$ får vi parallell projeksjonen.

2.7 Lys og refleksjonsmodeller

For å oppnå en mer realistisk visualisering kan det ofte hjelpe å benytte seg av forskjellige lys- og refleksjonsmodeller. En god

refleksjonsmodell kan gi tydelige inntrykk av overflater laget av plastikk, metall, glass osv.

Basisen for refleksjoner er lys. Innen datavisualisering har vi to typer lys som ofte blir brukt. Det ene er en punktkilde, plassert uendelig langt borte. Her er lysintensiteten konstant, og lysstrålene parallelle. Den andre er en lokal lyskilde. Her varierer intensiteten typisk med $1/d^2$, hvor d er avstanden mellom lyskilden og objektet.

Det er tre refleksjonsmodeller som er de vanligste å bruke innen datagrafikk. Det er ambient refleksjon, diffus refleksjon og spekulær refleksjon. Alle tre blir ofte brukt i den samme scenen for å gi et realistisk visuelt inntrykk. Ambient refleksjon modellerer bakgrunnslyset. Diffus refleksjon vil spre lyset isotropt i alle retninger og virker matt. Spekulær refleksjon blir brukt for å rendre skinnende objekter, som f.eks. metall.

Ambient refleksjon: Mengden lys L_a er den samme på alle punkter på overflatene. Noe av lyset blir absorbert, noe blir reflektert. Mengden av reflektert lys er gitt av refleksjonskoeffesienten $R_a = k_a$ der $0 \leq k_a \leq 1$. Lys intensiteten blir altså $I_a = k_a L_a$

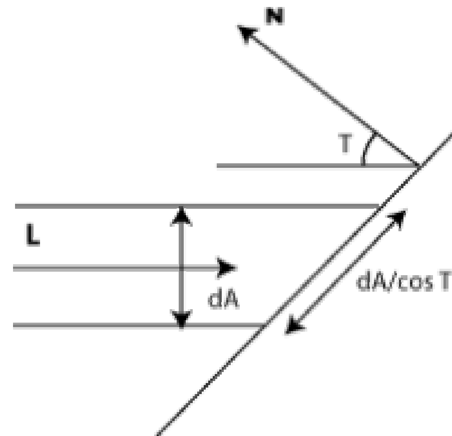
Diffus refleksjon: Diffuse overflater sprer lys i alle retninger like mye, mengden som reflekteres følger Lamberts lov: den diffuse refleksjonen er proporsjonal til $\cos\theta$, der θ er vinkelen mellom normalen N og retningen til lyset l. Denne formen for refleksjon er altså uavhengig av hvilken retning kamera har i forhold til objektet, men er avhengig av normalene til objektet og retningen til lyset.

Refleksjonskoefesienten blir her $R_d = k_d(l \cdot n)$ siden $\cos\theta = l \cdot n$ der $0 \leq k_d \leq 1$. Hvis vi også vil ha med at lyset svekkes når det reiser en avstand d kan vi benytte oss av et kvadratisk ledd,

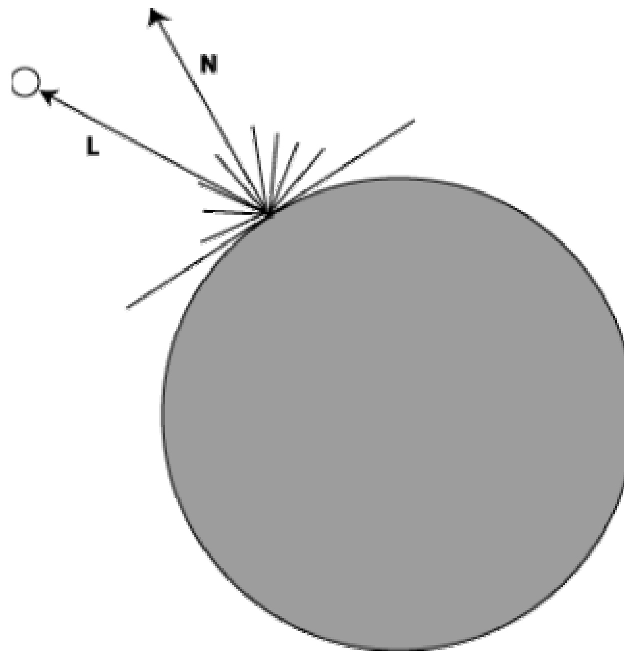
$$R_d = \frac{k_d}{a + bd + cd^2}(l \cdot n)$$

Lysintensiteten blir da,

$$I_d = \frac{k_d}{a + bd + cd^2}(l \cdot n)L_d$$



Figur 10: Refleksjon av lys. Hvor L er retningen til lyset og N er normalvektoren på flaten. T er vinkelen mellom L og N.



Figur 11: Diffus refleksjon. Hvor L er retningen til lyset og N er normalvektoren på flaten.

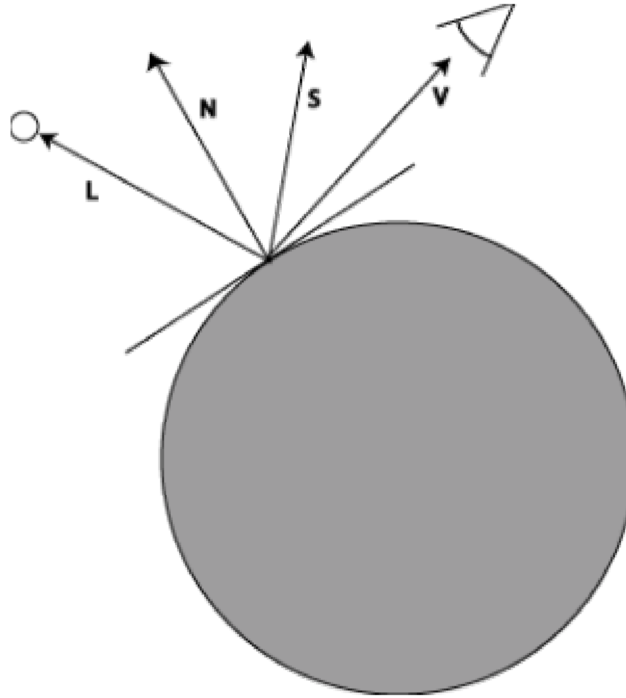
Spekulær refleksjon: Denne refleksjonsmodellen er avhengig av retningen til kamera samt vektoren til den reflekterte lysstrålen. Denne formen for refleksjon gir highlights. Mengden av spekulært lys som observeres er avhengig av vinkelen ϕ mellom refleksjons retningen *r* og retningen *v* til observereren.

I dette tilfelle blir refleksjonskoeffisienten $R_s = k_s \cos^{\alpha} \phi$, der $0 \leq k_s \leq 1$, og α er shininess koeffisienten. Når α øker vil det reflekterte lyset konsentreres i et mindre område, sentrert rundt *r*. Verdier for α mellom 100 og 500 korresponderer med metalliske overflater, mindre verdier vil gi en bredere highlight. Dersom *v* og *r* har lengde 1 blir $\cos\phi = r \cdot v$. Vi får da:

$$R_s = k_s (r \cdot v)^{\alpha}$$

Vi kan her også gange inn en lys forminskning som ligner på den for diffus refleksjon, og får en lysintensitet som ser slik ut:

$$I_s = \frac{k_s}{a + bd + cd^2} (r \cdot v)^{\alpha} L_s$$



Figur 12: Spekulær refleksjon. Her er L retningen til lyset, N er normalvektoren på flaten, S er vektoren til den reflekterte lysstrålen og V er retningen kameraet peker.

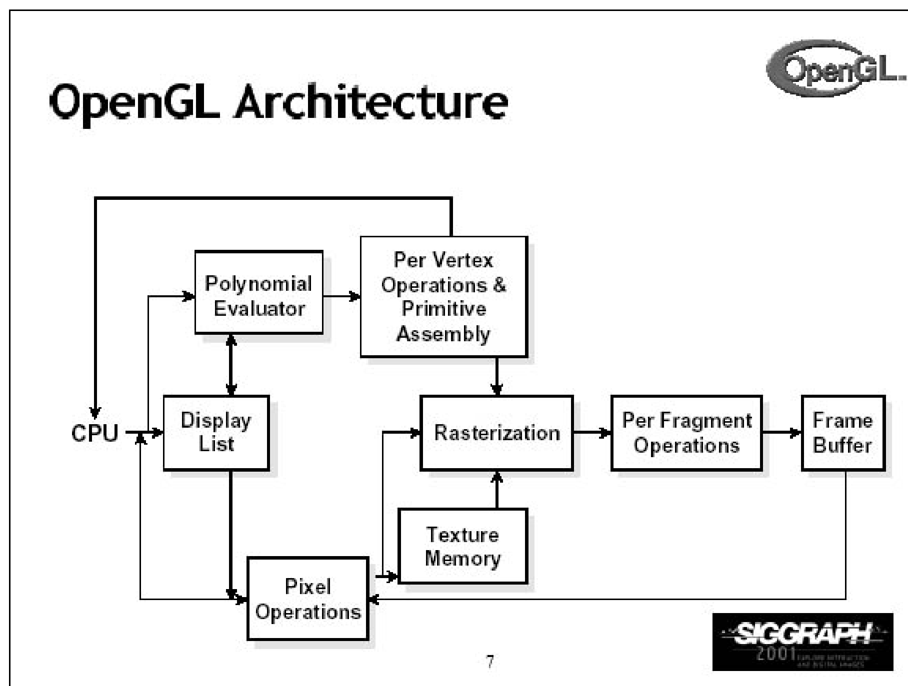
Ved å bruke moderne grafikkhardware inkludert teksturering, kan man oppnå svært realistiske renderinger. I neste kapittel skal vi se nærmere på hvordan dette fungerer i OpenGL.

3. OpenGL

Jeg har helt og holdent brukt OpenGL som verktøy for å utvikle voxelrenderen. OpenGL er et relativt lavnivå visualiseringsbibliotek, i forhold til f.eks. The Visualization Toolkit (VTK). Den største utfordringen man møter når man benytter OpenGL er at alt som skal visualiseres må bygges opp manuelt av enkle primitiver, punkter, linjer eller polygoner. Det finnes med andre ord ingen verktøy som kan benyttes til vitenskapelig visualisering, dette må utvikles selv. I dette kapittelet beskrives litt teori om OpenGL, og jeg går inn på noen detaljer om hvordan man benytter seg av rutinene som finnes.

3.1 Introduksjon

OpenGL er et lavnivå programvaregrensesnitt ment for å være portabelt og gi tilgang til hardware på lavest mulig nivå. Dette grensesnittet inneholder over 100 forskjellige kommandoer man kan bruke for å spesifisere de objekter og operasjoner som er nødvendig for å lage interaktive 3D applikasjoner.



Figur 13: OpenGL arkitekturen

OpenGL ble opprinnelig designet for å brukes i programmerings-språkene C og C++, men nå støttes OpenGL i mange flere språk, blant annet Java, Tcl, Python, Ada og FORTRAN.

Et av målene til designerne av OpenGL var å kun inkludere de egenskapene i API'en som ville bli akselerert av hardware [17]. Funksjoner som vindusoperasjoner og databehandlings verktøy ble delegert til høyere nivås verktøy. Man må derfor jobbe gjennom det vindussystemet som finnes i det operativsystemet du bruker. For å gjøre vindus- og input-operasjoner enklere, kan man bruke biblioteket The OpenGL Utility Toolkit (GLUT), som inneholder operasjoner for å sette opp vinduer og bruke forskjellige input enheter, uavhengig av den plattformen man kjører på. Dette gjør at OpenGL koden kan brukes på mange forskjellige plattformer uten å måtte forandre på koden.

OpenGL inneholder heller ikke kommandoer for å lage komplekse 3D objekter som biler, fly, deler, molekyler o.l. Man må istedenfor bygge opp komplekse objekter fra enkle geometrier, som punkter, linjer og polygoner. Det finnes også et bibliotek som gir tilgang til mer komplekse objekter. The OpenGL Utility Library (GLU) inneholder flere modeleringsegenskaper, blant annet quadric surfaces som man kan bruke til å generere objekter som kuler, sylindre, skiver, osv. GLU inneholder også mer avanserte operasjoner som NURBS kurver.

OpenGL er mye brukt innen mange forskjellige grener av dataindustrien. Spill industrien er kanskje der OpenGL er mest brukt, ved siden av det mer og mer dominerende Microsoft DirectX, men likevel, OpenGL kan måle seg i funksjonalitet og hastighet med DirectX fremdeles. OpenGL brukes også i flere visualiseringsprogrammer som f.eks. VTK og Open Inventor. Det er kombinasjonen mellom plattformuavhengighet, hardwarestøtte og den relativt enkle API'en som gir OpenGL sin styrke og popularitet.

3.2 Oppsett og bruk av OpenGL

Å sette opp OpenGL er relativt enkelt dersom man bruker GLUT. Dette biblioteket inneholder funksjoner for å blant annet sette opp vinduer. I main funksjonen under settes et vindu opp med omtrent 10 linjer kode. Denne koden er plattformuavhengig og fungerer på de fleste operativsystemer. Her settes også keyboard og mus opp.

```
#include <gl/gl.h>     // The GL Header File
#include <gl/glut.h>  // The GL Utility Toolkit (Glut) Header

void main ( int argc, char** argv )
{
  glutInit                  ( &argc, argv );
  glutInitDisplayMode ( GLUT_RGB | GLUT_DOUBLE );   // Display Mode
  glutInitWindowSize  ( 512, 512 );                          // Window size
  glutCreateWindow   ( " OpenGL Framework" );            // Window Title
  InitGL ();
  glutDisplayFunc     ( display );                              // Display function
  glutReshapeFunc    ( reshape );                              // Window reshape function
  glutKeyboardFunc  ( keyboard );                            // Keyboard functions
  glutSpecialFunc     ( arrow_keys );                         // Special keys function
  glutMainLoop        ( );                                          // Initialize The Main Loop
}
```

Vi må også definere en funksjon som brukes når vinduet forandrer størrelse, dette gjøres i funksjonen reshape(). Her bestemmes også projeksjonen som skal brukes, i dette tilfelle er det en perspektivprojeksjon.

```
gluPerspective ( FOV, aspect_ratio, near, far );
```

Hvor FOV definerer Field of View, eller åpningsvinkelen til kamera. Aspect Ratio er forholdet mellom høyden og bredden på vinduet. Near og Far definerer et volum hvor objekter som ligger mellom de to verdiene blir rendret.

```
void reshape ( int w, int h )                // Create The Reshape Function
{
  glViewport     ( 0, 0, w, h );
  glMatrixMode   ( GL_PROJECTION );  // Select The Projection Matrix
  glLoadIdentity ( );                        // Reset The Projection Matrix
  if ( h==0 )                                //Calculate The Aspect Ratio Of The Window
    gluPerspective ( 80, ( float ) w, 1.0, 5000.0 );
  else
    gluPerspective ( 80, ( float ) w / ( float ) h, 1.0, 5000.0 );
  glMatrixMode   ( GL_MODELVIEW );    // Select The Model View Matrix
  glLoadIdentity ( );                         // Reset The Model View Matrix
}
```

Når vinduet er satt opp med GLUT, må vi ofte initialisere noen av OpenGL funksjonene, dette gjøres i en InitGL funksjon.

```
void InitGL ( GLvoid )
{
   glShadeModel(GL_SMOOTH);                          // Enable Smooth Shading
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);             // Black Background
   glClearDepth(1.0f);                               // Depth Buffer Setup
   glEnable(GL_DEPTH_TEST);                          // Enables Depth Testing
   glDepthFunc(GL_LEQUAL);                           // Depth Testing To Do
}
```

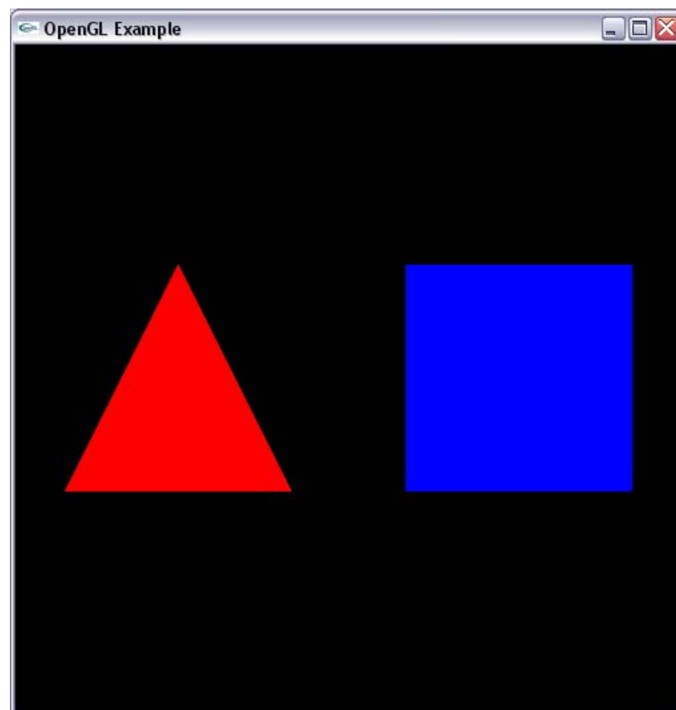
Nå er vi klare for å render objekter til skjermen. Display() funksjonen blir kjørt i en løkke og ser i sin enkleste form slik ut. Skjermen blir resatt, objekter tegnes til skjermbufferet, og skjermbufferet vises på skjermen.

```
void display ( void )   // Create The Display Function
{
    //ClearScreen And Depth Buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    RenderScene();
    glutSwapBuffers ( );
}
```

Voxelrender algoritmen baserer seg på å manipulere og teksturere kvadrater. Under er et eksempel på hvordan et kvadrat kan rendres i OpenGL.

```
void RenderQuad()  // Renders a quad
{
        glColor3f(1.0, 1.0, 1.0);                   // White color
        glTranslatef(0.0,0.0,-3.0);                 // Move Right 3 Units
        glBegin(GL_QUADS);                          // Draw A Quad
                glVertex3f(-1.0, 1.0, 0.0);         // Top Left
                glVertex3f( 1.0, 1.0, 0.0);         // Top Right
                glVertex3f( 1.0,-1.0, 0.0);         // Bottom Right
                glVertex3f(-1.0,-1.0, 0.0);         // Bottom Left
        glEnd();
}
```



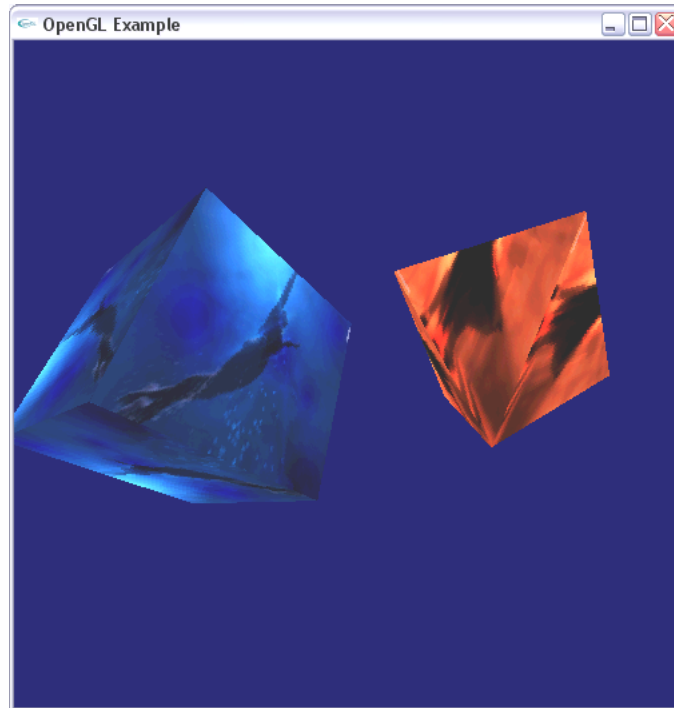
Figur 14: Trekant og kvadrat visualisert i OpenGL

Det finnes ingen støtte for å lese inn teksturer i OpenGL. Det finnes derimot støtte for dette i GLU og i flere andre bibliotek, alt ettersom hvilke filformat man ønsker å bruke. Når en tekstur er lastet inn, må man skru på teksturmapping samt binde teksturen til det som skal tegnes. I denne sammenheng må man tilordne teksturkoordinater til hver node i objektet.

```
void RenderTexturedQuad()  // Renders a textured quad
{
        glEnable ( GL_TEXTURE_2D );
        glBindTexture ( GL_TEXTURE_2D, texture_id );

        glBegin(GL_QUADS);                                          // Draw A Quad
                glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f); // Top Left
```

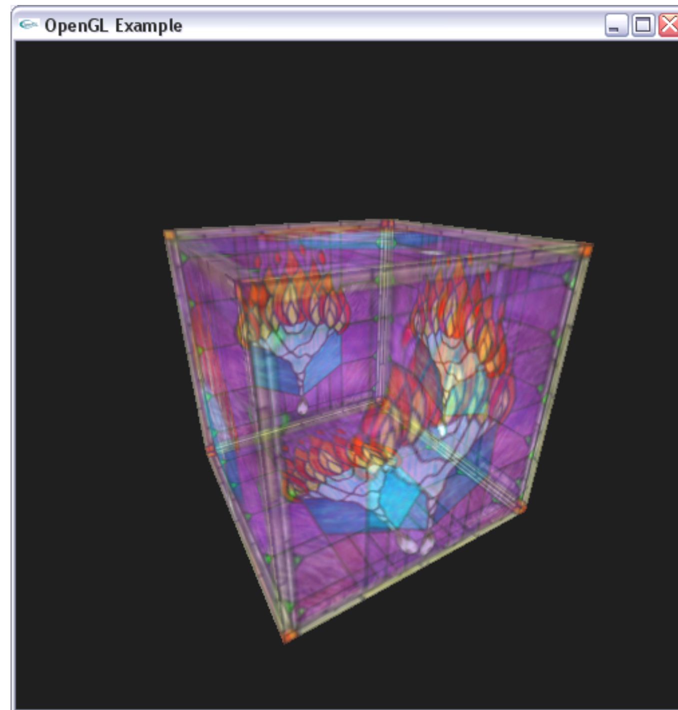
```
		glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f); // Top Right
		glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f); // Bottom Right
		glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f); // Bottom Left
	glEnd();
}
```



Figur 15: Teksturert kube og pyramide i OpenGL

I voxelrendererern er man avhengig av gjennomsiktighet av voxelplanene. OpenGL har støtte for dette, og kalles blending. Blending er meget enkelt å sette opp og bruke i OpenGL. Først velger man hvilke blendingsfunksjons man vil bruke, så skrur man på blending, og som oftest skrur man av dybde testing. Når dybdetesting er på tegnes ikke skjulte flater opp på skjermen, derfor må man skru denne av når man bruker blending, siden vi da vil se også de skjulte flatene gjennom de gjennomsiktige objektene.

```
void RenderBlendedQuad()  // Renders a transparent
{
	// Blending Function For Translucency Based On Source Alpha Value
	glBlendFunc(GL_SRC_ALPHA,GL_ONE);
	glEnable(GL_BLEND);				// Turn Blending On
	glDisable(GL_DEPTH_TEST);			// Turn Depth Testing Off
}
```



Figur 16: En teksturert kube med blending

Det er mange forskjellige måter å bevege kamera på. En av de enkleste måtene er å knytte musa til rotasjonen av objektet. Man kan benytte GLUT for å få tak i (x,y) posisjonen av muspila og bruke disse verdiene som grunnlag for rotasjonen av kameraet.

```
glLoadIdentity ( );
glTranslatef ( 0.0, 0.0, z );
glRotatef(mouse_x, 0.0, 1.0, 0.0);
glRotatef(mouse_y, 1.0, 0.0, 0.0);
```

Når man har ett objekt som skal tegnes opp på skjermen mange ganger kan man i OpenGL putte dette objektet i en Display List. Denne listen blir så kalt hver gang objektet skal vises. Denne metoden øker hastigheten til programmet, fordi objektet ikke må reallokeres i minne og struktureres på nytt hver gang objektet tegnes til skjermen.

```
GLvoid BuildLists()                         // Build display list
{
    object=glGenLists(1);                   // Generates list for one object
    glNewList(object, GL_COMPILE);          // Compiles the list
        drawObject();                       // Draw the object to the list
    glEndList();                            // End list
}
glCallList(box);                            // Draws the display list to screen
```

OpenGL har også støtte for klippeplan. Klippeplanet fjerner alt som ligger på den ene siden av det definerte planet, og tegner resten.

```
double peq[4] = {0.0, 0.0, -1.0, -depth};
glClipPlane(GL_CLIP_PLANE0, peq);
glEnable(GL_CLIP_PLANE0);
```

3.3 OpenGL vs DirectX (Direct3D)

DirectX, eller mer spesifikt Direct3D, som er 3D delen av DirectX er det mest brukte grafikkbiblioteket i spillsammenheng. Sammenlignet med OpenGL er ikke DirectX raskere, men har på Windows maskiner bedre driverstøtte enn OpenGL fra noen av de forskjellige 3D kortene som finnes. DirectX er ikke støttet i noe annet operativsystem enn Microsoft Windows, mens OpenGL har en bredere støtte i de vanligste operativsystemene. Den største forskjellen ligger i hvordan disse to bibliotekene brukes. OpenGL sitt grensesnitt er basert på prosedyrer, man utfører operasjoner ved å kalle GL funksjoner. Kodebiten under viser hva som må gjøres i OpenGL for å tegne opp en trekant.

```
glBegin (GL_TRIANGLES);
        glVertex (0,0,0);
        glVertex (1,1,0);
        glVertex (2,0,0);
glEnd ();
```

Direct3D sitt grensesnitt er basert på buffere. Man bygger en struktur som inneholder data og kommandoer, og utfører hele strukturen med ett enkelt kall. Dette kan virke som en fordel for Direct3D, siden man blir kvitt en del prosedyrekall. Problemet er at dette fort blir meget tungvint å kode. Koden under viser tilsvarende hvordan man lager en trekant i Direct3D.

```
v = &buffer.vertexes[0];
v->x = 0; v->y = 0; v->z = 0;
v++;
v->x = 1; v->y = 1; v->z = 0;
v++;
v->x = 2; v->y = 0; v->z = 0;
c = &buffer.commands;
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);
```

Man vil i Direct3D aldri kjøre ett buffer med bare et enkelt triangel i, man ville i såfall få meget dårlig ytelse. Man bygger derimot opp en stor samling av kommandoer, slik at man kan flytte mye arbeid til Direct3D med et enkelt prosedyrekall. Et problem som oppstår i denne sammenheng er hvordan vi skal definere ”stor samling av kommandoer”, dette vil variere avhengig av hva slags grafikkhardware man bruker. Istedenfor å la dette være opp til driverene, må her programmereren selv avgjøre hva som er best.

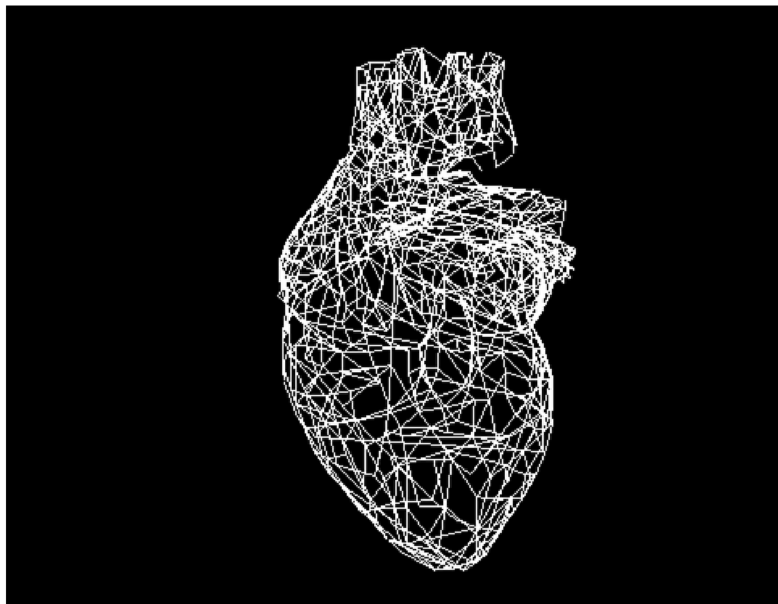
3.4 Qt

Qt er et multiplattform C++ bibliotek utviklet av Trolltech [25] som kan brukes under Windows, Linux og Mac OS X. Biblioteket har blant annet vertkøy for å lage grafisk brukergrensesnitt med menyer, knapper, slidere og OpenGL renderings vindu, samt har støtte for input enheter som mus og tastatur.

I voxelrendereren jeg lagde benyttet jeg meg av Qt istedenfor GLUT. GLUT egner seg veldig dårlig til større applikasjoner som krever brukergrensesnitt og input fra bruker. GLUT inneholder bare de enkleste funksjonene for å sette opp vindu, og input operasjoner. Qt derimot inneholder det aller meste man trenger for å lage et godt brukergrensesnitt, blant annet knapper, slidere og inputfelt.

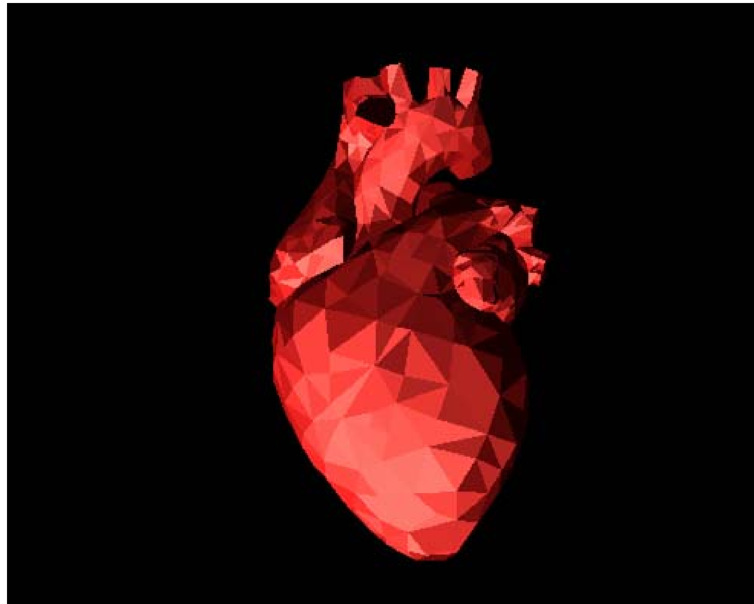
3.5 Renderings teknikker i OpenGL

OpenGL har mange teknikker å benytte seg av for å oppnå forskjellige effekter og representasjoner av en 3D scene. Den enkleste er Wireframe teknikken. Med Wireframe vil alle objekter kun bestå av linjer, der hver linje representerer en kant av et polygon. Hele scenen vil da bli gjennomsiktig, siden du kan se rett gjennom linjerepresentasjonen Wireframe gir, og alle linjene er like tydelige uansett hvor langt unna de er.



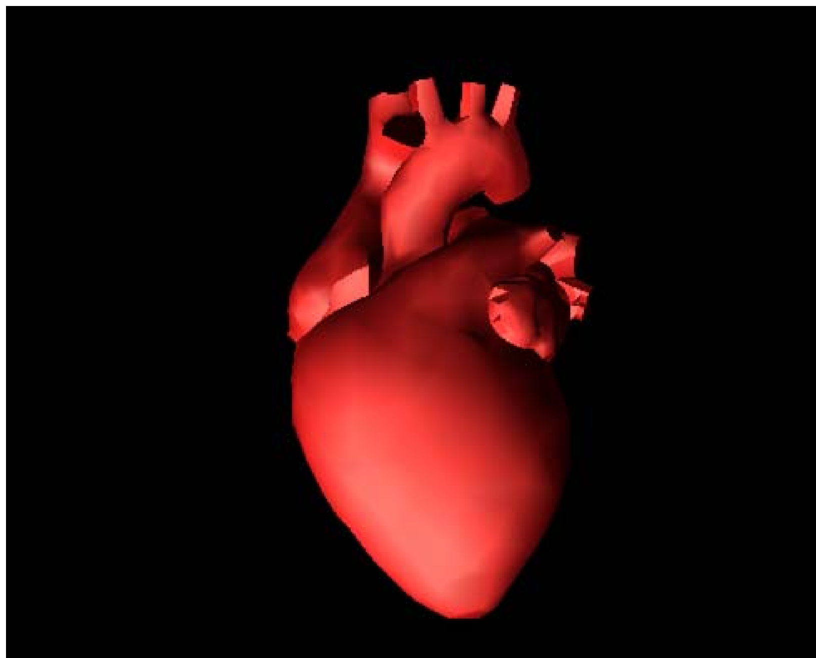
Figur 17: Wireframe av et hjerte

Videre kan man få objekter til å se solide ut, ved hjelp av Flat Shading. Hvert polygonen til et objekt vil da få den samme fargen, og hele objektet vil se kantete ut.



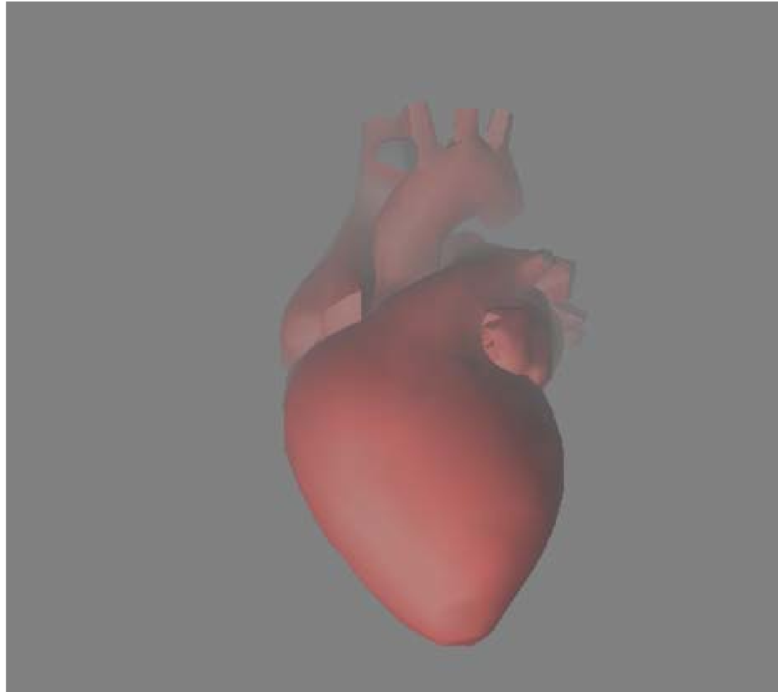
Figur 18: Flat Shading av et hjerte

For å få objekter til å se mer realistiske ut, kan man benytte seg av Smooth Shading. I OpenGL bruker man Gouraud Shading for å oppnå denne effekten. I tillegg kan man legge til lys, slik at man får opplyste områder av et objekt, samt skyggelagte og mørkere områder, som bidrar til dybdefølelsen av objektet.



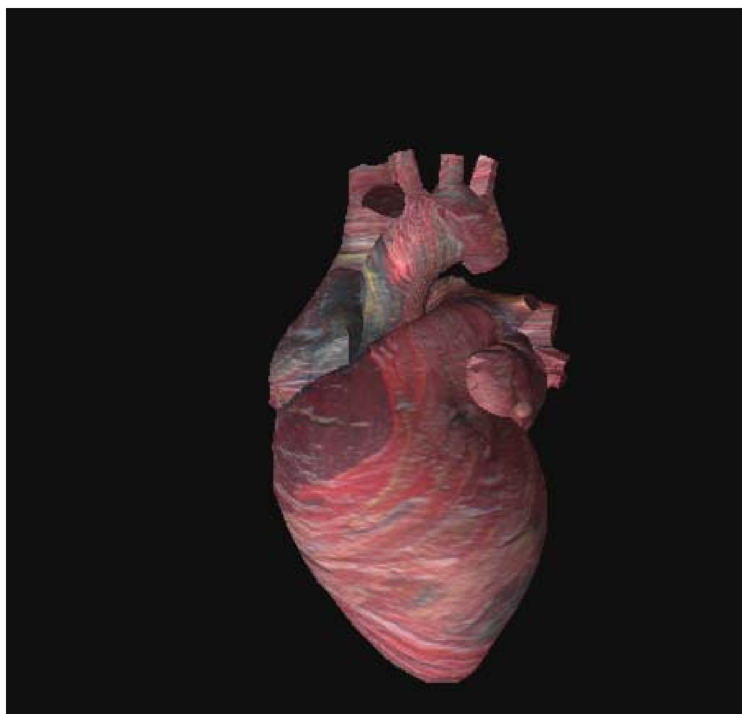
Figur 19: Smooth Shading av et hjerte

Man kan legge inn dybdeeffekter som gjør slik at objektene lenger vekk blir dimmet ut. OpenGL oppnår en slik effekt ved å bruke atmosfæriske effekter som tåke. Effekten gjør at det ser ut som om objektene befinner seg i en tåke.



Figur 20: Fog effekt

Teksturering kan gi objektene et meget realistisk utseende. Med teksturering kan man legge 2D bilder på et 3D objekt, f.eks. ved å legge på et bilde av et gulv på et enkelt tynt kvadrat, og man har på den måten laget et realistisk gulv. Dette kan man gjøre i svært god detalj, og sammen med en detaljert 3D modell og god lyssetting, kan man lage objekter som det ikke er mulig å skille fra et virkelig objekt.



Figur 21: Teksturering av hjertet

På grunn av den begrensede oppløsningen man ofte har på en dataskjerm kan objektene se hakkete ut, spesielt ved skarpe kanter i scenen, alt etter hvor lav oppløsning man bruker. For å forminske denne effekten kan OpenGL benytte seg av antialiasing. Dette er en teknikk som glatter ut kantene i en scene ved å approksimere nye farger på pikslene ved å ta gjennomsnittet av nærliggende piksler.

Skygger kan videre bidra med å øke realismen i en scene. Dessverre har ikke OpenGL noen kommandoer for å lage dette. Det finnes derimot flere forskjellige andre teknikker man kan benytte seg av for å lage skygger i OpenGL, men disse må man lage selv.

Andre effekter man kan benytte seg av, er motion-blur, som gir inntrykk av at noe er i bevegelse, og Depth-of-Field, som gir den samme effekten som oppstår når man f.eks. har et kamera ute av fokus.

3.6 Fargetabell

Det er ønskelig å benytte en fargetabell i voxelrenderer sammenheng. OpenGL har støtte for dette, og kalles Color-Index mode. En fargetabell fungerer på mange måter på samme måte som en maler som på forhånd har gjort klar fargepaletten. Man plukker ut den fargen man vil bruke og tilfører den til piksler på skjermen. Man er da begrenset til antall farger samtidig på skjermen av størrelsen på fargetabellen. Størrelsen til fargetabellen er avhengig av hvor mye hardware som er dedikert til den. Størrelsen på en fargetabell er alltid av en potens av 2, og typiske størrelser ligger mellom 256 og 2096. I motsetning til RGBA modus der hver piksels farge er uavhengig av hverandre, vil i Color-Index modus alle piksler med samme farge ha den samme verdien. Derfor, hvis man forandrer på en farge i fargetabellen vil alle pikslene med den fargen forandre seg til den nye fargen i fargetabellen. Dette er spesielt nyttig i sammenheng med voxelrendering. Dersom man i RGBA modus ønsker å skifte farge eller gjennomsiktighet av visse verdier av datasettet, må man kalkulere hele datasettet på nytt, og tilordne de nye fargene. I Color-Index modus kan man forandre direkte på paletten knyttet til de dataverdiene man ønsker å forandre og forandringen skjer instantant, uten at ekstra kalkulasjoner er nødvendig.

3.7 Konklusjon

Jeg har jobbet svært mye med OpenGL i forbindelse med hovedoppgaven, og opparbeidet mye erfaringer med API'en. Det som slår meg som det mest positive med OpenGL er at det er enklere å benytte enn f.eks. DirectX, og man får samtidig utnyttet dagens moderne 3D hardware på en svært effektiv måte. Noen av ulempene ved å bruke OpenGL er blant annet at for å laste inn teksturer kreves at man må lage sin egen innleser, eller bruke et bibliotek som håndterer

dette. Et annet problem er at ikke alle 3D kort har like god støtte for OpenGL som DirectX. Det er heller ikke trivielt med OpenGL å lage ting som isoflater og voxelrendering, siden det ikke finnes noe funksjonalitet for dette i OpenGL. Man må derfor lage dette selv. Det finnes dog alternativer, nemlig pakker som VTK, der du har bibliotek med ferdig lagde funksjoner som f.eks. isoflater, hvor man på noen veldig få linjer med kode kan definere og tegne opp en isoflate. Open Inventor bringer dette videre ytterligere ett steg, her har man et grafisk brukergrensesnitt som gjør det enklere å lage vitenskapelige visualiseringer. Bakdelene med disse programmene er at de er betydelig tregere ytelsesmessig, i motsetning til å kode programmet i ren lavnivå OpenGL.

Jeg klarte uten problemer, og med akseptabel framerate, å visualisere datasett opp til 512x512x512 på et nytt GeForce4 3D kort med 128 MB teksturminne. Når datasettet blir større enn dette er det ikke lenger plass i teksturminnet til skjermkortet og dataen blir da lagret i minnet på hovedkortet, noe som fører til veldig dårlig ytelse for disse datasettene. Så lenge dataen får plass i teksturminnet derimot er hastighetene gode.

OpenGL egner seg meget bra for utvikling av 3D applikasjoner, og som grunnlag for andre applikasjoner. Med kombinasjonen GLUT eller Qt og OpenGL kan man sette sammen små applikasjoner med svært få linjer kode, som i tillegg er plattformuavhengig og hardwareakselerert.

4. 3D visualisering av numeriske beregninger med Voxelrenderer

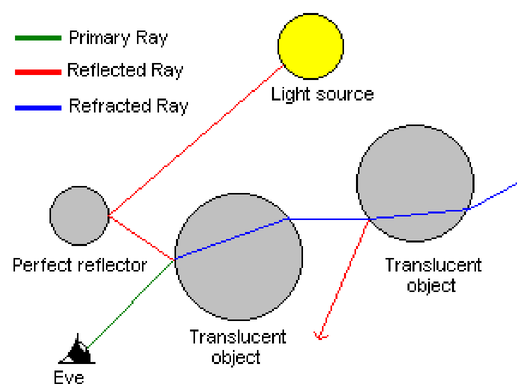
I dette kapittelet forklares teorien som ligger bak en voxelrenderer, hvordan OpenGL voxelrendereren ble implementert, og hva den kan brukes til. Samt forskjellige teknikker som ofte benyttes innen voxelrendering.

4.1 Volumrenderer

Volumrendering av vitenskapelige data er typisk visualisering av volumetriske skalarer, vektorer eller tensor data. Slike datasett hentes gjerne fra simuleringer, eksperimenter eller annet teknisk utstyr som genererer volumdata.

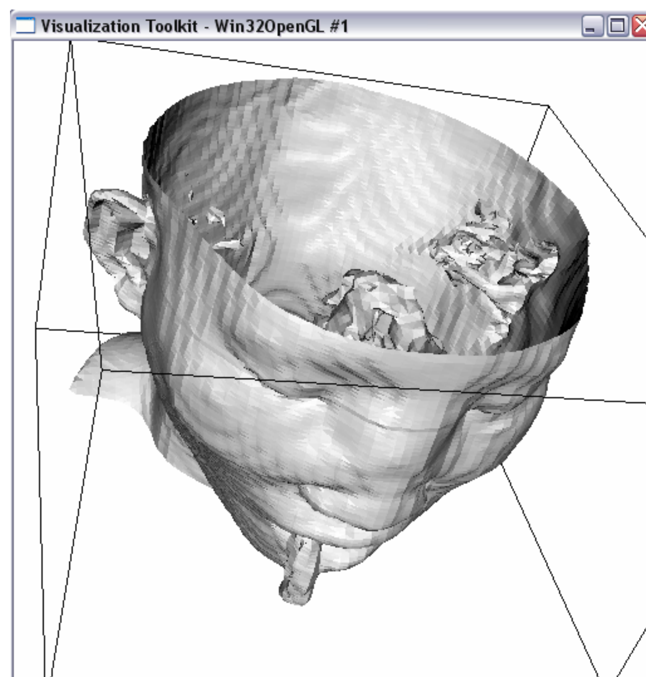
Volumrendering gjør det mulig å visualisere et 3D datasett, f.eks. et skalarfelt, i et enkelt bilde. Å prosessere slike bilder blir fort veldig tregt, spesielt hvis man ikke har støtte fra hardware, og i dette tilfellet, teksturhardware. Det finnes mange forskjellige volumrenderings algoritmer, men de to vanligste måtene å gjennomføre volumrendering på er enten ved å benytte seg av en teksturbasert volumrenderings teknikk, eller en raytracing metode. Flere volumrenderings teknikker bygger videre på dette og kan for eksempel kombinere både raytracing og teksturmetoden, Ray-Slice-Sweep [20] er en slik algoritme.

Raytracing metoden gir resultat ved å kalkulere fargen til hver enkelt lysstråle som treffer hvert piksel på skjermen. Raytracing gir et godt resultat, men er desverre tregt, og det finnes liten støtte for raytracing i hardware. Det finnes spesialhardware for å rendre raytracing på forskningsnivå, men dette finnes kun på få maskiner [21]. Det finnes også spesialhardware for volumrendering, for eksempel Cube chipsettet som er diskutert i artikkelen til Kreeger og Kaufman [18]. Med teksturmetoden derimot kan man benytte vanlig 3D hardware som kan kjøpes til en rimelig pris, og som gir en veldig rask rendering.



Figur 22: Raytracing

Den vanligste måten å visualisere 3D skalarfelt på, er å bruke isoflater. Isoflater representerer skalarfeltet med polygoner og lager en lukket flate av feltet ved en gitt verdi i skalarfeltet. Derfor må man velge en verdi man vil generere en isoflate ut i fra, og man vil av den grunn ikke kunne visualisere hele feltet med ett bilde. Dersom man vil se på skalarfeltet ved en annen verdi, må man generere hele isoflaten på nytt med den nye verdien. Ved å gjøre isoflatene gjennomsiktig kan man rendre flere isoflater samtidig, men det skal ikke mange isoflater til før det blir så mange at man ikke får noen ny informasjon ved å legge til en ny isoflate. I praksis vil derfor denne teknikken bare kunne vise en liten del av det totale datasettet. Volumrendering derimot kan bruke gjennomsiktighet til en mye større grad enn isoflater og kan vise en mye større del av det totale datasettet.



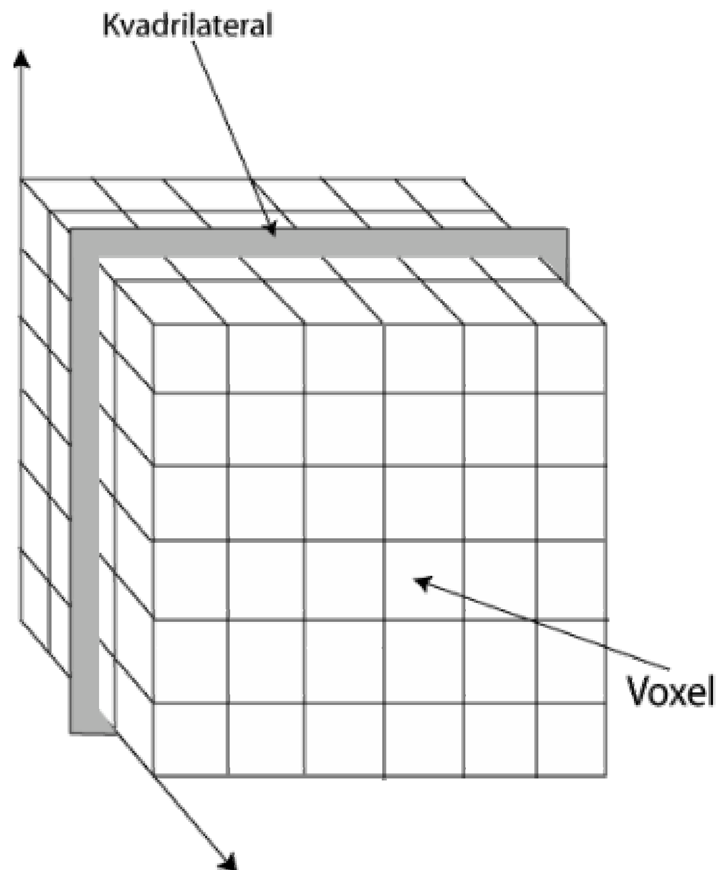
Figur 23: VTK Isoflate ved en gitt skalarverdi

4.2 Voxelrender Algoritmen

Vi starter med et gitt voxelsett med dimensjoner X, Y, Z, og en fargetabell som inneholder r, g, b fargeverdier for hvert av verdiene assosiert med voxelsettet. Et koordinatsystem blir definert, der (0, 0, 0) er i midten av scenen, og renderingen tegner rundt dette punktet. Før man begynner renderingen må man finne det koordinatplanet som har retning nærmest skjermnormalen. Etter at dette koordinatplanet er funnet, blir voxelsettet rendret i 3D ved å bruke kvadrilateraler som vist i figuren under. Kvadrilateralene genereres der de møter voxelsettet, og danner et plan vinkelrett på skjermnormalen. Kvadrilateralene rendres bakenfra og forover (mot skjermen). Når kvadrilateralene rendres slik vil hver voxel dekke til de voxlene som alerede har blitt

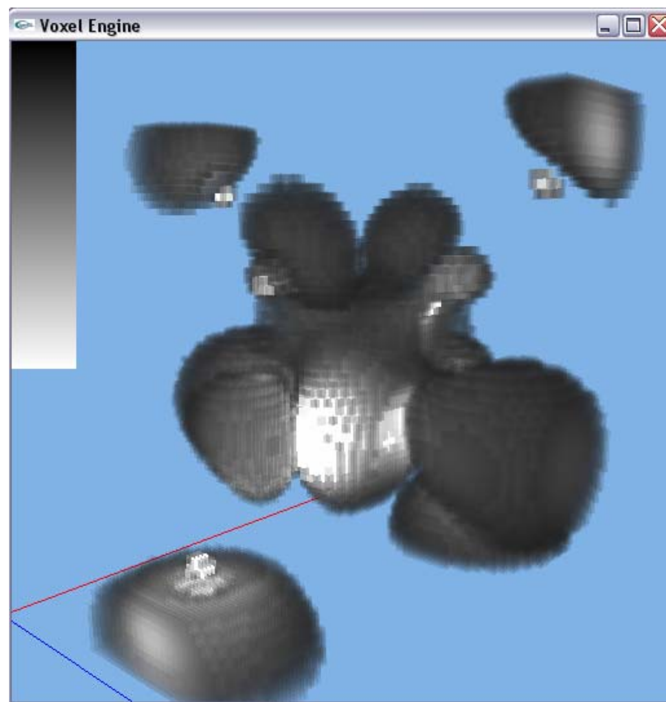
tegnet i forhold til dens farge og gjennomsiktighet. Mindre gjennomsiktige voxler vil gi større bidrag til det ferdige resultatet enn de som er mer gjennomsiktige.

Antall kvadrilateraler må ikke nødvendigvis være det samme antallet som antall voxelplan i den spesielle retningen. Hvis antallet er det samme, havner kvadrilateralene i midten av hvert voxelplan. Da får man to valg når voxelen skal fargelegges. Kvadrilateralene kan få fargen til den nærmeste voxelen, eller man kan tri-lineært interpolere den fra voxlene rundt. Man kan også ha et større antall kvadrilateraler enn voxelplan, som gir en glattere rendering sammenlignet med et plan for hvert voxelplan. Dersom man har hardware som ikke støtter tri-lineær interpolasjon vil dette føre til et dårligere resultat enn bi-lineær interpolasjon når man ikke ser rett på et voxelplan. Men resultatet vil være det samme som tri-lineær interpolasjon dersom man ser rett på voxelplanet.

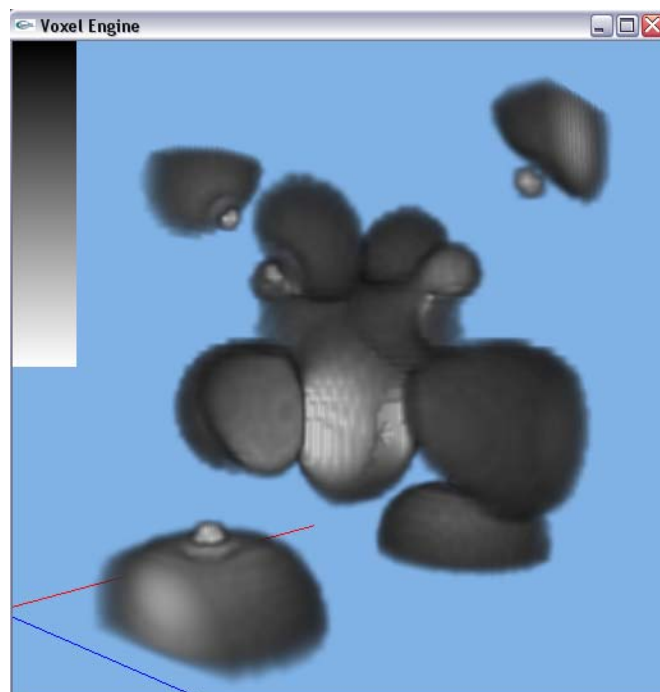


Figur 24: Voxel plan

Denne algoritmen har den fordelen at mesteparten av den er implementert i hardware systemer med støtte for teksturhardware. Bildene under er laget av OpenGL voxelmotoren jeg har kodet, og visualiserer et datasett med dimensjon 68x68x68 av et jernprotein. De to bildene illustrerer effekten av å benytte seg av teksturfiltrering for å interpolere datasettet slik at det ser glattere ut.



Figur 25: Voxelrendering, uten teksturfiltrering



Figur 26: Voxelrendering, med teksturfiltrering

4.3 Shading i volum visualisering

Når det gjelder datavisualisering, er det en del grunnleggende forskjeller sammenlignet med mer visuelle simuleringer eller virtuell virkelighet. Et mål innen vitenskapelig visualisering er å gi en representasjon av dataene ved hjelp av datagrafikk som kommuniserer konsekvensen av dataene så unikt som mulig. I en scenesimulering, kan det være et mål å bygge en virtuell scene som er så lik virkeligheten som mulig. I dette tilfellet vil man ofte benytte seg av eksterne lys, som ikke har noe med selve objektene å gjøre. Innen vitenskapelig visualisering vil vi at det genererte bildet skal representere så unikt som mulig datasettet vi har, slik at det er enkelt å se det som er viktig i dataen. Bildene trenger ikke å ha korrekte farger, men f.eks. ha farger som plukker ut kontraster på en god måte. Derfor trenger vi ikke å benytte oss av eksterne lys, men genererer fargene kun ut fra datasettet. Man kan deretter justere fargene og gjennomsiktigheten til hver verdi i datasettet for og blant annet kunne lokalisere og visualisere deler av datasettet.

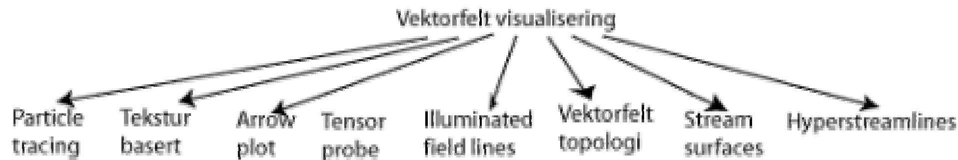
Dersom man ønsker å tilordne en lysmodell til volum visualisering, er det eneste alternativet man har, gjennom emisjon og absorpsjon av lys på voxelnivå [22]. Det utstrålte lyset skal uttrykke verdien til en spesiell voxel. Denne metoden er det ikke mulig å akselerere i vanlig 3D hardware, så jeg har derfor ikke implementert denne teknikken.

En annen metode for å skape shading i volum visualisering er å benytte seg av en effekt kalt randfordunkling. Dette er en effekt som man kan observere når man f.eks. ser på sola i et lite teleskop, nemlig at sola er lysest på midten og blir mørkere langs randen. Dette kommer av at temperaturen til sola varierer med dybden inne i atmosfæren, og fotosfæren lyser generelt sett sterkere jo varmere den er. Dersom du ser midt på sola vil du se lenger inn i atmosfæren enn langs randen, og det du ser på midten vil derfor være lysere. Dette kommer av at når du ser på randen må du se gjennom mer atmosfære for å nå den samme dybden som på midten av sola, pga. helningsvinkelen du får langs kanten. Denne effekten kan man også benytte seg av i volum visualisering for å oppnå en dybdeeffekt.

For å oppnå randfordunklings effekten manipulerer man fargetabellen på en tilsvarende måte, nemlig ved å tilordne mørkere verdier og minske gjennomsiktigheten langs kantene til et objekt. Objektet blir da seendes tredimensjonalt ut. Dette er vanskelig å få til uten en god HSVA fargetabell, det er mulig med en RGBA tabell, men mye vanskeligere. I OpenGL voxelrendereren jeg har laget har jeg ikke implementert en slik fargetabell, da dette er en meget stor jobb. Istedenfor har jeg laget en tilnærming av randfordunklingsteknikken der vi kan ved hjelp av en slider justere en verdi som forteller hvor i datasettet randfordunklingen skal skje.

4.4 Visualisering av vektorfelt

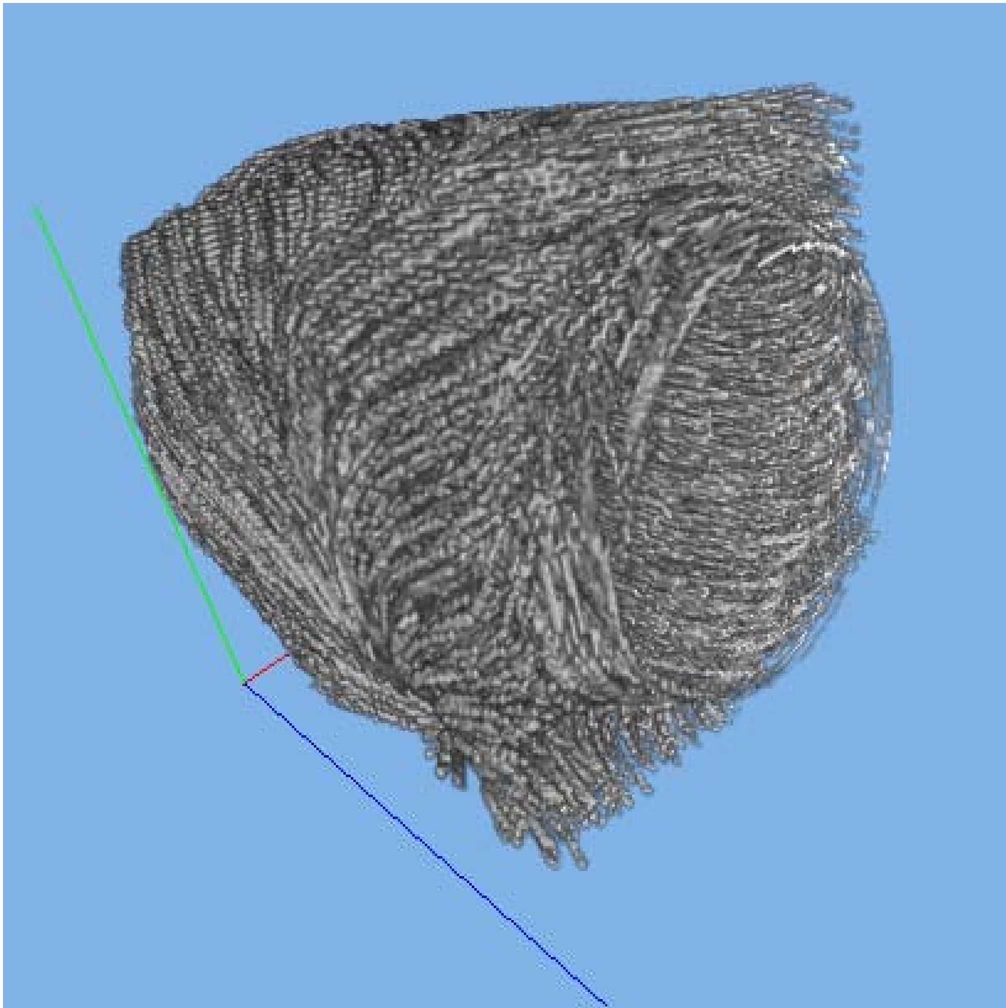
Visualisering av vektordata har alltid vært en av de store utfordringene innen vitenskapelig visualisering. Figur 27 viser hovedteknikkene for å visualisere vektorfelt. Bare noen av disse var kjent i begynnelsen av 90-tallet, når Hesseling og van Wijk [6] klassifiserte disse algoritmene. De fleste av disse algoritmene kan visualiseres enten med voxelrendering eller i kombinasjon med den. Jeg har sett på og implementert noen av disse.



Figur 27: Diagram av vektorfels visualiserings teknikker

Particle Tracing er en teknikk der en slipper partikler fri i et vektorfelt og lar de bevege seg påvirket av feltet. Denne teknikken er mye av det samme som skjer i eksperimentelle strømninger, der partikler, f.eks. røyk slippes ut i luften, eller farge i vann. Denne teknikken lar seg kombinere med voxelrendering, ved for eksempel å gjøre volumet gjennomsiktig slik at partiklene kan bevege seg gjennom datasettet.

Dette kan også gjøres på teksturer, ved hjelp av LIC (Line Integral Convolution). LIC gir en elegant måte å effektivt visualisere retningsbestemt informasjon [14]. Denne teknikken tar et vektorfelt som ligger i et kartesisk grid og en tekstur av den samme størrelsen, som for eksempel bare er hvit støy, og filtrerer denne teksturen langs vektorene som er tangent til feltet, dvs. langs streamlines. Denne teknikken gir et datasett som består av skalarer og kan lett visualiseres direkte i en voxelrenderer. Dette er gjort på figur 28. Her er det startet med et datasett med hvit støy, som så har blitt kjørt gjennom en LIC prosess slik at strukturen i vektorfeltet blir synlig, visualisert i voxelrendereren jeg kodet.



Figur 28: 128x128x128 datasett av random støy kjørt gjennom en LIC prossess slik at strukturen i vektorfeltet blir synlig. Visualisert i OpenGL voxelrenderer.

Arrow plot er den enkleste formen for vektorvisualisering. Ved å plukke ut punkter i vektorfeltet og tegne på lengden og retningen av vektorene i punktene får man en representasjon av det totale vektorfeltet. Man kan bygge videre på dette med såkalte Tensor Probes som kan gi mer informasjon enn kun magnetuden og retningen på feltet, som for eksempel kurving og vridning representert ved et sett med geometriske symboler.

Illuminated field lines [23] er en vektorfelts visualiseringsmetode som benytter seg av å skyggelegge et stort antall streamlines. Ved å ta i bruk ambient, diffuse og spekulære refleksjonsmodeller samt gjennomsiktighet, får man en bedre romlig følelse av dataen og man kan man oppnå finere detaljer slik at for eksempel små virverlstrømmer kan oppdages. Shading av linjer er ikke støttet i hardware, men ved å bruke en tekstureringsteknikk kan man få til en hardwarestøttet og rask metode å rendre skyggelagte streamlines på.

En annen alternativ tilnærming til vektorfelt visualisering benytter seg av å trekke ut topologiske trekk, for eksempel kritiske punkter. De

kritiske punktene i feltet kan identifiseres av en analyse av Jacobi matrisen til vektoren med hensyn på posisjonen. Saddelpunkt, tiltrekkende noder, frastøtende noder etc. blir plukket ut. Streamlines blir så tegnet fra hver av de utvalgte kritiske punktene. Resultatet blir ofte et meget enkelt og rent plot, som man kan observere og dedusere hele vektorfeltet ut i fra.

Stream surfaces er definert som et sett av tette nærliggende streamlines, som gir en intuitiv måte å observere 3D vektorfelt på. Hyperstreamlines er en generalisering av vektorfelt streamlines og kan brukes til å visualisere 3D andregrads tensor felt langs kontinuerlige veier.

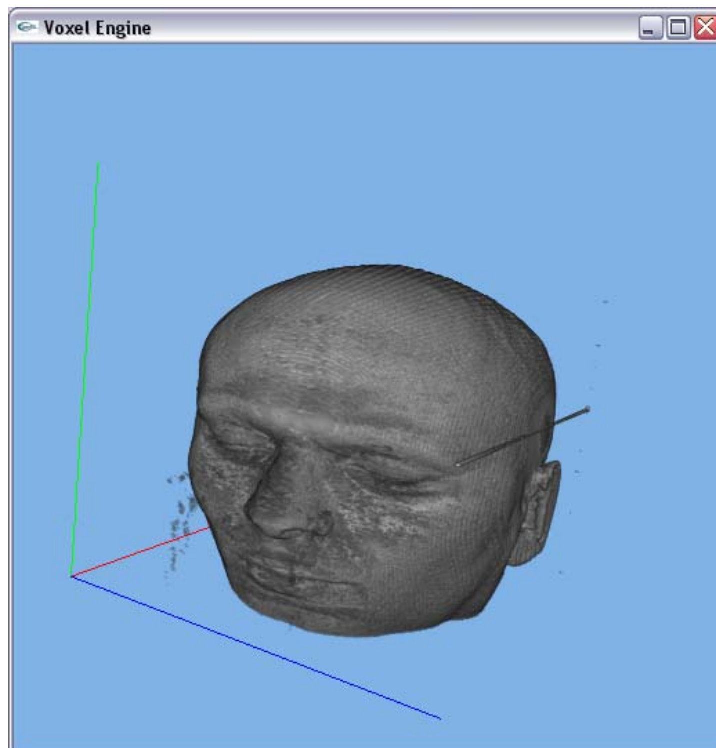
4.5 Kutteplan

Et kutteplan er et plan som skjærer gjennom datasettet og enten fjerner all data på den ene siden av kutteplanet, eller fjerner all data bortsett fra de dataene som kutteplanet skjærer gjennom. Dataen blir da avbildet på kutteplanet.

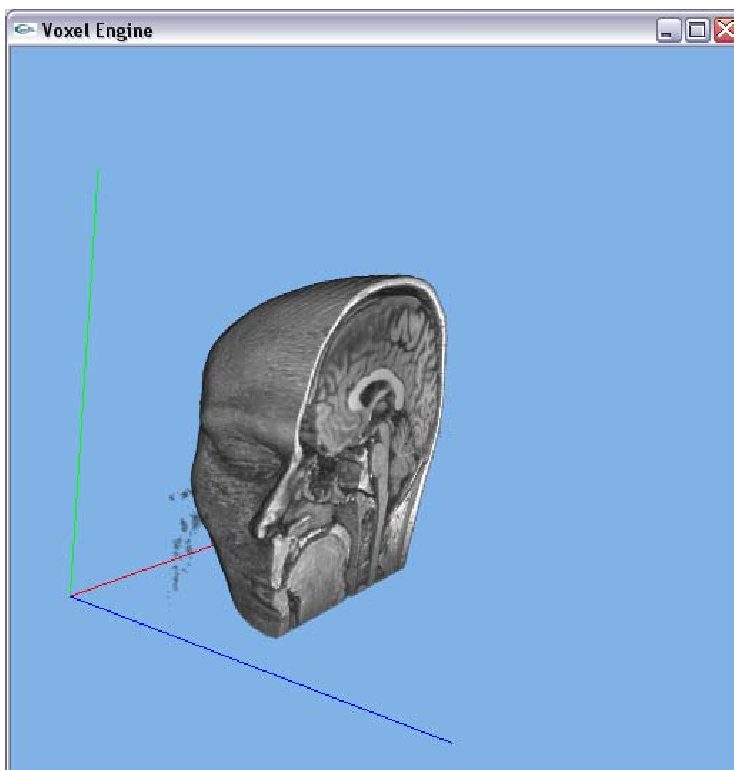
OpenGL har en egen funksjon for kutteplan, og er støttet i hardware på alle nyere 3D kort. Man definerer med denne funksjonen et kutteplan som det ikke er lov å tegne gjennom.

```
double peq[4] = {0.0, 0.0, -1.0, -depth};
        glClipPlane(GL_CLIP_PLANE0, peq);
        glEnable(GL_CLIP_PLANE0);
```

Denne kodebiten gir resultatet som vi ser i figurene 29 og 30. Bildene i figur 29 og 30 er generert av Voxelrendereren jeg har laget. Datasettet er en 256x256x256 MRI av en ukjent persons hode.



Figur 29: Voxelrenderer uten kutte plan

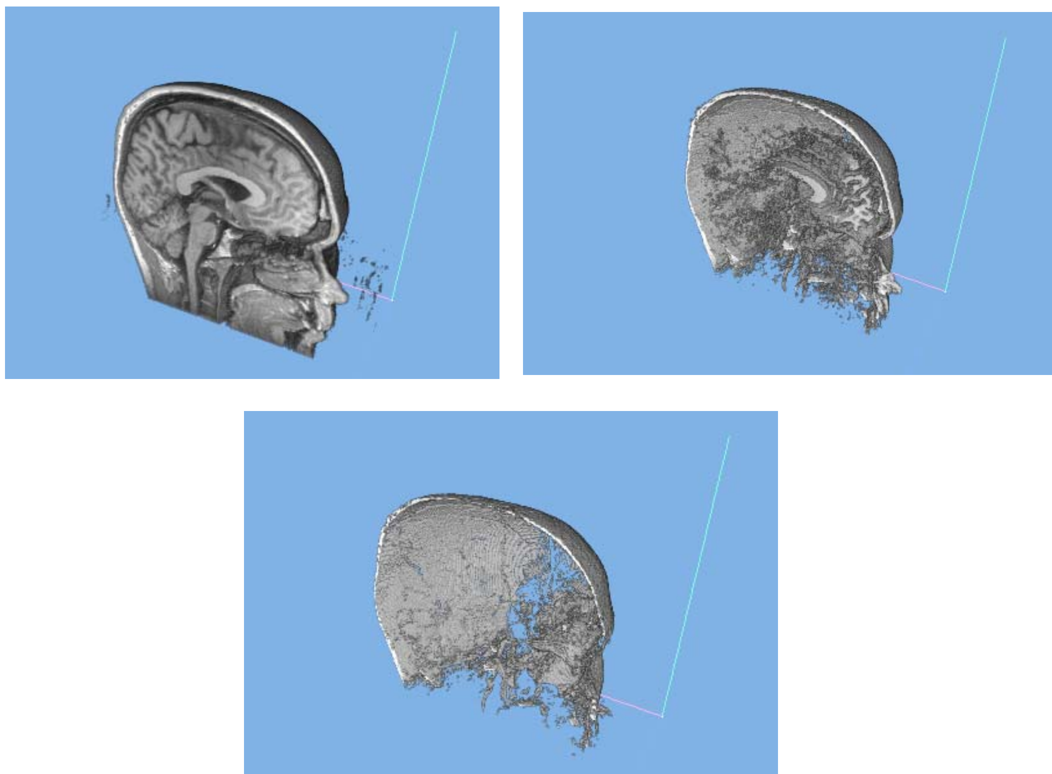


Figur 30: Voxelrenderer med kutte plan langs Y-aksen

4.6 Fargetabell

En god fargetabell inneholder muligheten til å endre på fargen og gjennomsiktigheten til hver verdi av datasettet. Den bør ha verktøy for å justere flere av verdiene samtidig. Den mest intuitive fargemodellen å bruke regnes for å være HSV modellen. Med denne modellen finner man fargen man ønsker ved å justere Hue, Saturation og Value. Hue verdien går gjennom alle fargene, Saturation verdien korresponderer med å legge til eller trekke fra hvitt til fargen og Value verdien korresponderer med å legge til eller trekke fra svart.

Jeg implementerte en meget enkel form for fargetabell i voxelrendereren. Denne fargetabellen tar to verdier som input, en nedre og øvre verdi som avgrenser hvilke verdier av datasettet som skal synes. I bildeserien under har jeg tatt tre bilder der jeg i bilde 1 ikke fjernet noen verdier, bilde 2 har jeg fjernet den laveste halvparten av verdiene, og i bildet 3 har jeg fjernet den laveste ¾ av verdiene. De lave verdiene er representert av mykere materie enn verdier med høye verdier. Derfor vil i bilde 3 være bare de hardeste delene av hodet som synes.



Figur 31: Tre bilder som illustrerer effekten av å fjerne deler av datasettet, ved å sette opasitet lik 0

4.7 Animasjon

Mennesket har en god egenskap til å trekke ut viktig informasjon og lagre detaljer fra bilder i sekvens over korte tidsskalaer. Dette er grunnen til at animasjon er et så verdifullt verktøy. Animasjon består av en rekke bilder i serie satt sammen i en sekvens. Dette kan bli brukt til å skape et inntrykk av bevegelse. Animasjon kan brukes til å fremheve tidsvarierende fysisk informasjon.

Innen datavisualisering blir animasjoner generert fra enkeltbilder rendrert separat. Ideelt sett vil man kanskje ønske å kunne generere sanntid animasjon mens simuleringen pågår, men under de fleste omstendigheter er ikke dette praktisk pga. de enorme kravene som ville settes for simulerings og renderings hardware'en. Store simuleringer av f.eks. turbulente strømninger, kan ta flere måneder å simulere på de største datamaskinene. De genererer flere terabyte med data som må lagres og post prosesseseres til en form som kan visualiseres. Når det gjelder vitenskapelig data vet man ofte ikke hva alle resultatene vil bli på forhånd, og det kan være vanskelig å vite hva man skal se etter senere. Det er derfor en god idé å lagre så mye av simulasjonen som praktisk er mulig, slik at man kan hente ut det man trenger senere. Det er også lurt å lagre dataen så nær hverandre i tid som man har muligheten til, for å få animasjonen til å bli så glatt som mulig.

Hvis vi ser bare noen få år tilbake, var veldig store datasett regnet for å være bare noen få gigabyte, og ting som nå kan kjøres i sanntid måtte genereres over flere timer eller dager. Dette gjør at når vi får nyere, raskere og større datamaskiner, kan vi simulere og visualisere flere nye ting, og flere viktige problemer kan nå og i fremtiden visualiseres i sanntid. De datasettene jeg har benyttet meg av er tilstrekkelig store til at det kan være nødvendig å rendre til fil. Jeg har derfor implementert rutiner for dette.

Jeg har implementert to former for lagring av animasjon i min voxelrenderer. Den ene formen er ment å bruke til å ta opp og lagre bilder i form av TGA filer endringer som blir gjort sanntid i programmet. Slik som rotasjon og bevegelse av kutteplan og endringer i fargetabellen. Den andre formen brukes når vi har animerte datasett, for eksempel en tidsserie generert av Diffpack. Da laster programmet inn datasettet for hvert tidssteg og lagrer hvert bilde på harddisken. Disse bildene settes sammen til en film senere, for eksempel lagret som MPEG. Den ferdige animasjonen kan lagres på hardisker, CD-ROM, DVD og videokassetter.

4.8 Volum visualisering - begrensninger

En av ulempene med å bruke denne formen for visualisering, er at man er begrenset til et uniformt kartesisk grid. Man kan utvide voxelmotoren

til å bli mer fleksibel ved å benytte seg av tetrahedriske teksturblokker i tillegg til voxler. Dette vil da gi støtte for et ikke-uniformt grid, men dette er en modell som krever kraftigere hardware for å visualisere tilsvarende store datasett i sanntid.

Dersom man har en lav oppløsning på dataen, eller man har zoomet veldig langt inn på dataen vil resultatet se veldig hakkete ut, eller smurt utover, alt ettersom om man bruker teksturinterpolasjon eller ikke. Ved å øke antall sample-planes får modellen et glattere utseende, og dersom datasettet er stort, kan man minske antall sample-planes for å minske minneforbruket og øke hastigheten på opptegningen.

4.9 Implementasjon

Jeg implementerte voxelmotoren i C++ og brukte OpenGL som visualiserings verktøy. Denne kombinasjonen er ypperlig for applikasjoner som krever høy ytelse. Vindu og input operasjoner ble gjort ved hjelp av GLUT i første omgang, og i senere versjoner i Qt for å få tilgang til flere verktøy, og for å bevare plattformuavhenighet på en enkel måte.

Jeg hadde også flere forskjellige datasett over flere filformater som skulle visualiseres. Matlab datafiler generert av Diffpack konverterte jeg til ukomprimert ASCII format, som beskrevet i kapittel 6.1. Jeg laget rutiner for å gjøre dette, og funksjoner som leste dataen inn. Jeg lagde også en HDF leser (mer om dette formatet i kapittel 6.2) for å lese inn datasett fra dette formatet. Jeg fikk også tak i datasett i binært RAW-format. Dvs. data lagret binært og ukomprimert, som jeg også lagde en leser for. Jeg lagde tilslutt kode for å generere egene enkle matematiske test-modeller, slik som kuler, sylindre, osv.

Det mest optimale formatet å bruke i denne forbindelse, er data lagret binært som en unsigned char, dvs. verdier mellom 0 og 255. Denne lagringsformen krever minimalt med plass og dersom man bruker luminocity modellen trenger man kun en verdi pr. punkt i datasettet. Dersom flere detaljer er ønsket kan man lagre dataene med unsigned char og lagre RGBA verdiene etter hverandre i datasettet. Etter at datasettet er lest inn fra fil genereres en 3D tekstur, der hver piksel i teksturen representerer en voxel.

Når datasettene blir større enn teksturminnet til skjermkortet, er det praktisk å rendre bildene til fil, slik at de kan sees i rask rekkefølge. Dette blir gjort ved å rendre hvert bilde og bruke OpenGL funksjonen glReadPixels() som leser renderingsvinduet og lagrer den i et array. Dette arrayet blir igjen lagret som en TGA fil på harddisken. Bildene kjører jeg deretter gjennom et program som generer AVI video filer.

For å kunne skille ut viktige detaljer i datasettet er det viktig å kunne justere fargen og gjennomsiktigheten til hver verdi i datasettet. Dette

gjøres ved å gi alle verdier over eller under en gitt verdi i datasettet en alpha verdi lik 0. Da vil alle verdier som ble satt til 0 usynlige og vi kan se på strukturer ved forskjellige verdier i datasettet. Videre kan man få frem flere detaljer i et datasett ved å justere fargene til verdiene i tillegg til å justere alpha verdiene.

Etterhvert kan man bruke mye tid på å lage en bedre fargetabell enn den jeg har laget. I den optimale fargetabellen bør man kunne justere R,G,B,A til alle verdiene i datasettet, eller ofte enda bedre H(ue), S(aturation), V(alue), A(alpha) verdiene, som er en mer intuitiv fargetabell enn RGBA tabellen å bruke. Det å lage en slik fargetabell i Qt er meget krevende, og var en for stor jobb å få ferdig i denne sammenheng.

4.10 Praktiske anvendelser av voxelrendering

Det praktiske område der voxelrendering nok blir mest brukt er innen medisin, med visualisering av MRI og CT scans til diagnostisering og som hjelp under operasjon. Innen vitenskapelig visualisering er også voxelrendering på sitt sterkeste, pga. den gode egenskapen til å rendre skalar, vektor og tensor data som ofte blir generert av simuleringer. Slike simuleringer kan være simuleringer av fluid dynamikk, kjernefysikk, elektromagnetisme for å nevne noen.

Volumrendering passer også ypperlig for å visualisere skyer, støv, røyk og lignende, noe som oftere og oftere blir benyttet innen spillindustrien, i for eksempel fly simulatorer. Volumrendering blir i disse tilfellene brukt sammen med polygonrendering.

Figur 32: Volumetriske skyer i spillet IL-2 Sturmovik laget av Madox Games

4.11 Konklusjon

Voxelrendering kan som vi ser gi meget gode visualiseringer av store datasett. I eksemplene der hodet blir visualisert er det lett å se hvorfor dette er et viktig verktøy for diagnostisering. I mange tilfeller kan også voxelrendering gi bedre resultater enn f.eks. bruk av isoflater.

En voxelrenderer kan bygges opp enten ved bruk av raytracing eller ved en projeksjonsmetode. Raytracing er meget tregt, og gir ikke nødvendigvis noe bedre resultat, som jeg ser mer på i kapittel 5. Projeksjonsmetoden derimot er meget rask, og man kan bruke 3D hardware for å gjøre den enda raskere.

Ved å benytte seg av randfordunkling kan man få frem en god visualisering av dybdefenomener uten å benytte seg av shading, noe som er meget tidsbesparende og som igjen gir økt interaksjon. Denne teknikken kan man også bruke for å slippe å benytte seg av volumetrisk opplysning som vil senke ytelsesevnen til applikasjonen. Randfordunklingsteknikken er en teknikk som bare består av å forandre på fargetabellen og har derfor ingen påvirkning på ytelsen til programmet.

5. VTK vs OpenGL

5.1 VTK

The Visualization Toolkit (VTK) er et open source, og gratis visualiseringsbibliotek for 3D datagrafikk, bildebehandling og visualisering. VTK finnes som et C++ bibliotek, og finnes også for Tcl/Tk, Java og Python. VTK er implentert på omtrent alle Unix baserte platformer, PC'er (Windows 95/98/NT/2000/XP) og Max OSX Jaguar og nyere versjoner [4].

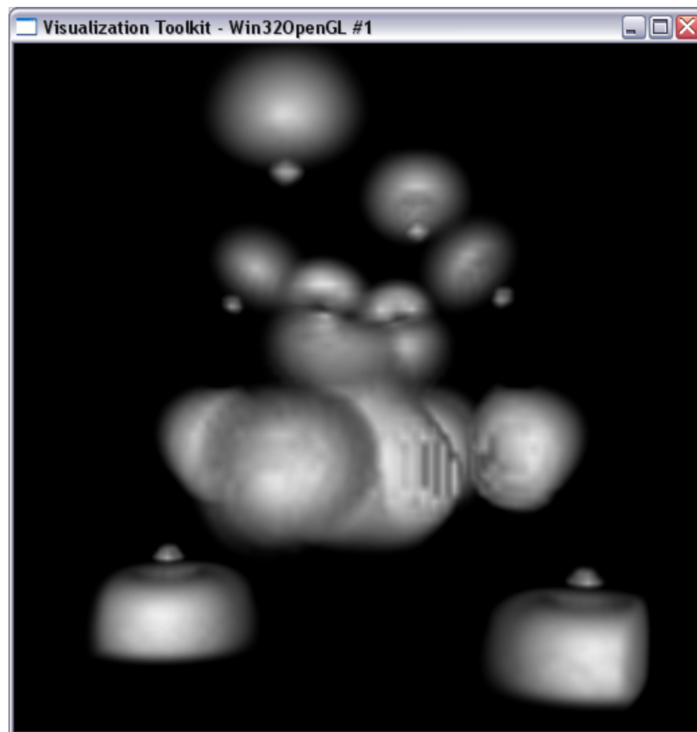
VTK baserer seg på et høyere nivå for å rendre grafikk enn f.eks. OpenGL. Dette gjør det enklere å lage nyttig grafikk og visualiserings applikasjoner. Med VTK kan applikasjonene skrives direkte i C++, Tcl, Java eller Python, som kan gjøre utviklingen av applikasjoner veldig rask.

Den største nytten av VTK ligger i at du ikke bare kan visualisere geometri. VTK støtter mange forskjellige visualiserings algoritmer, blant annet skalar, vektor, tensor, tekstur og volmetriske metoder. Samt avanserte modelleringsteknikker som implisitt modellering, polygon reduksjon, overflate utjevning, klipping, konturering og Delaunay triangulering. I tillegg finnes det bildebehandlings algoritmer slik at 2D og 3D grafikk kan kombineres.

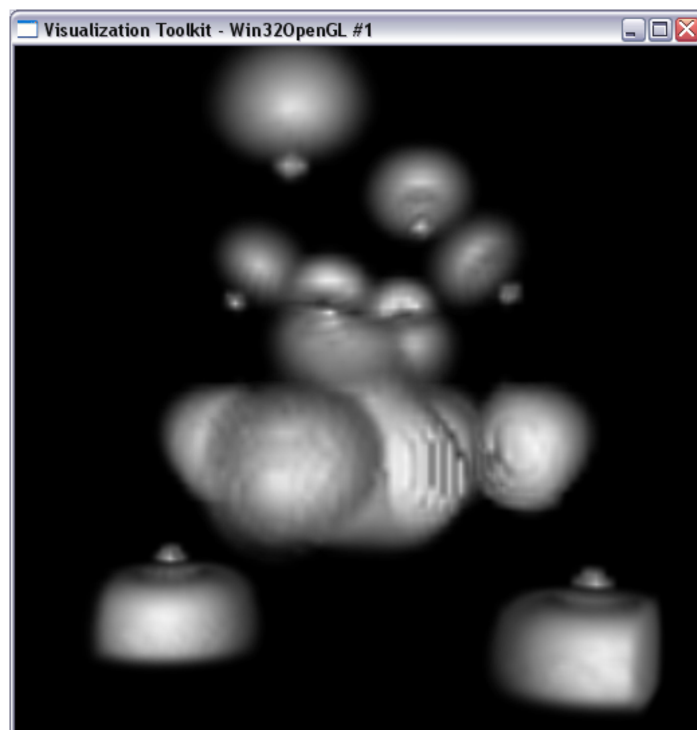
5.2 VTK vs OpenGL

Etter å ha laget en voxelrenderer i OpenGL har jeg også laget en voxelrenderer med VTK for å sammenligne kodelengde, hastigheter og resultater. Jeg benyttet meg av datasettet ironProt.vtk som er et datasett med 68x68x68 elementer, og som representerer et jernprotein. Datasettet ble visualisert i både OpenGL voxelrendereren og i VTK sin voxelrenderer i samme vindusstørrelse, 512x512.

Med VTK sin voxelrenderer har man to muligheter for valg av renderings teknikk. Den ene teknikken benytter seg av raytracing, den andre benytter seg av teksturer og fungerer på samme måte som min OpenGL voxelrenderer. Raytracing teknikken er ikke hardware-akselerert og er derfor ventet å være tregere enn alternativet. Jeg kjørte først en test for å sammenligne resultat og hastighet mellom raycasting metoden og teksturmetoden.



Figur 33: VTK volumrendering med Raycast metoden

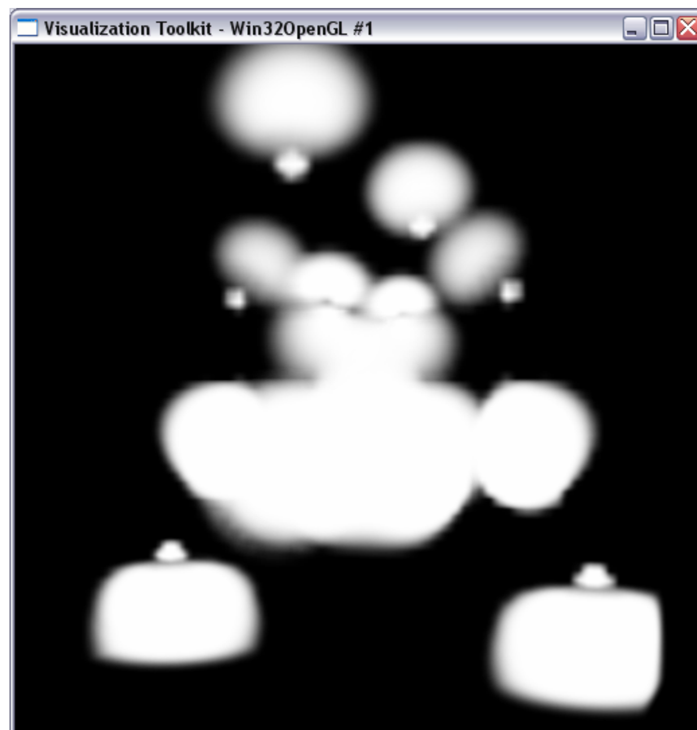


Figur 34: VTK volumrendering med teksturmetoden

Som vi ser i figur 33 og 34 er det nesten en ubetydelig forskjell i bildekvalitet mellom de to teknikkene. Raycast metoden ser litt glattere ut noen steder, men i dette eksempelet er forskjellen særdeles liten. Forskjell i hastighet er derimot større. Raycast metoden fikk en

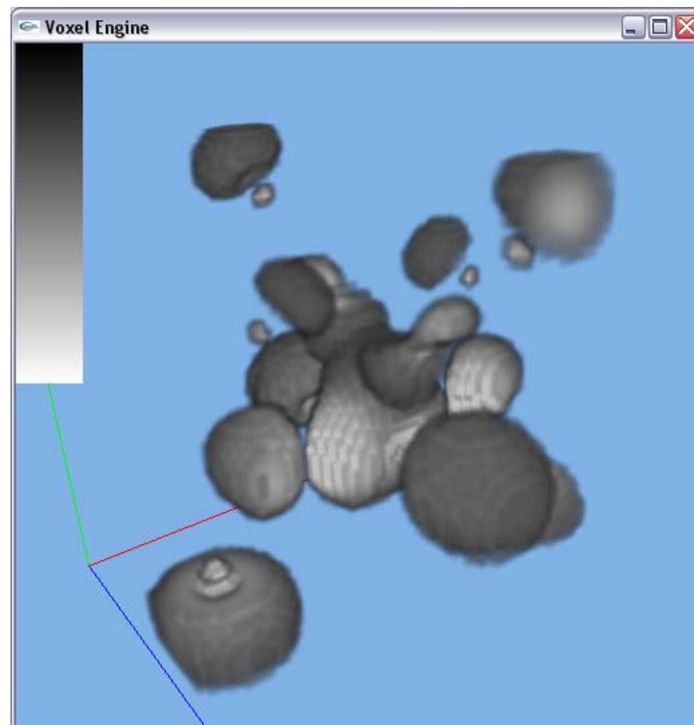
framerate på rundt ett bilde pr sekund, mot en hastighet på ca 15 bilder i sekundet med teksturmetoden.

I begge eksemplene over er shading skrudd på. Shading blir her brukt for å skape nyanser i datasettet for lettere å kunne skille ut detaljer. For å ytterligere forsøke å få VTK volumrendereren til å gå raskere skrudde jeg av shading. Jeg fikk da en betydelig hastighetsøkning, opp til ca. 60 bilder pr sekund, men resultatet ble mye dårligere, som vist i figur 35. Årsaken til dette er at ingen fargetabell er definert for datasettet, og VTK bruker bare gråtoner.



Figur 35: VTK volumrenderer uten shading

I min OpenGL voxelrenderer benytter jeg meg heller ikke av shading, men som vi ser i figur 36 er det fremdeles mye flere detaljer å se her. Hastigheten her er ca 70 bilder pr sekund.



Figur 36: OpenGL voxelrenderer

Det er to hovedfaktorer som begrenser hastigheten i en hardware akselerert voxelrenderer som baserer seg på teksturflater, fillrate og antall polygoner som skal rendres. I eksempelet over er det 68 teksturerte kvadrater på skjermen til en hver tid, eller 136 trekanter som skal rendres. Dette er et svært lite antall polygoner for moderne hardware som gjerne kan rendre flere hundre tusen polygoner pr. sekund. Fillrate derimot er den hastigheten skjermkortet klarer å rendre et visst antall piksler på pr. sekund. Disse 68 teksturflatene er gjennomsiktige og må derfor alle tegnes opp på skjermen for hvert frame og sjekke fargen mot de flatene som allerede er tegnet. Dersom alle teksturflatene dekker hele vinduet må med andre ord hele skjermbildet tegnes opp 68 ganger, og vi får 68 ganger så mange pikseloperasjoner. Dette blir en flaskehals i visualiseringen, siden CPU'en ikke har noe problemer med å mate flere polygoner, men skjermkortet klarer ikke å tegne de opp fort nok.

Dette er årsaken til at VTK voxelrendereren med 2D teksturmetoden og shading skrudd av er omtrent like rask som OpenGL voxelrendereren. Når polygonene først er sendt til skjermkortet er det skjermkortet som er flaskehalsen og ikke prosessoren. VTK er derimot mye tregere enn OpenGL dersom vi bruker et stort antall actors eller display lists som det heter i OpenGL. Dette er lett å teste ved å lage to applikasjoner for sammenligning.

En applikasjon ble skrevet i VTK og den andre i OpenGL. Begge applikasjonene viser n antall kuber, der hver kube er en 14 punkts triangle strip, og kan forandre måten dataen er strukturert på ved å variere antall actors i VTK eller display lists i OpenGL.

Når man bruker én actor eller én display list blir resultatet omtrent det samme:

Antall kuber	Framerate
1000	Framerate = 75 (begrenset av refreshraten)
8000	Framerate = 38
15625	Framerate = 25
27000	Framerate = 15

Når antallet actors ble økt for bare 1000 kuber ble resultatet forandret.

For 1000 kuber, men variabel antall actors.

Actors (1000 kuber)	Framerate
1	Framerate = 75 (begrenset av refreshraten)
100	Framerate = 75 (begrenset av refreshraten)
125	Framerate = 38
250	Framerate = 38
500	Framerate = 19
1000	Framerate = 9.4

I OpenGL var det ingen forskjell når man varierte antallet display lists, frameraten var her fremdeles 75 bilder pr sekund.

5.3 Konklusjon

Når det gjelder lengden og kompleksiteten på VTK koden mot OpenGL koden er det naturlig nok stor forskjell. Omtrent 100 linjer VTK kode er nok for å sette opp og kjøre visualiseringen, mot over 1000 linjer kode i OpenGL. VTK egner seg derfor meget bra dersom man vil lage visualiseringer på kort tid. Fordelen med OpenGL rendereren er at man har kontroll på et mye lavere nivå enn i VTK, og kan derfor optimalisere applikasjonen slik at den passer bedre til den jobben som skal utføres.

Hastigheten til VTK rendereren som er teksturbasert var heller ikke så mye tregere enn OpenGL rendereren, hovedsakelig pga. at det er fillraten som er begrensningen, og at det er et lavt antall polygoner som benyttes i en voxelrenderer. Man får størst ytelesesforskjell mellom OpenGL og VTK dersom man benytter seg av mange actorer eller display lister. Hvis dette er tilfelle i applikasjonen er det fordelaktig å velge OpenGL over VTK.

6 Simulering og datasett

6.1 Simulering av 3D bølgeligning

Simuleringene jeg har visualisert er blant annet generert av Diffpack. Her simuleres en bølgepuls i 3D, med refleksjon fra veggene, over en viss tidsperiode. Jeg brukte [Wave1] fra Diffpack CD'en til å generer et 3D datasett fra [test2_3D.i]. Formatet som [Wave1] genererte konverterte jeg til ASCII for å gjøre innlesningen av data så enkelt som mulig. Datasettene blir imidlertid svært ofte veldig store, så å lagre ukomprimert ASCII datafiler er ikke mulig i praksis dersom man har veldig store filer. I slike tilfeller bør man lagre dataen binært, og i tillegg komprimere datasettene.

Med kommandoen "simres2matlab -f SIMULATION -s -n u -A –a" konverterte jeg datasettet mitt til ASCII. Det blir da generert en fil pr. tidssteg, der hver fil inneholder alle verdiene til punktene i boksen pr. tidssteg.

Simuleringen visualiseres ved å kjøre scriptet simulate.py. Dette scriptet kjører voxelrendereren fra kommandolinjen og generer et TGA bilde pr. tidssteg, som etterpå kan settes sammen til en MPEG film.

6.2 Datasett

Jeg testet også voxelrenderen på en del statiske datasett. Disse datasettene er lagret på forskjellige formater blandt annet i HDF4 formatet. HDF4 datasettene har dimensjon 128x128x128 og inneholder LIC bilder laget av Anders Helgeland.

Jeg har også lastet ned datasett fra diverse sider på Internett, og er MRI og CT scans av forskjellige ting, bla. hoder, en motor, et tre og en bamse. Disse datasettene er lagret i RAW formatet.

Det neste datasettet jeg har brukt er et 68x68x68 VTK datasett av et jernprotein, dette datasettet har jeg visualisert i både VTK og OpenGL rendereren.

Tilslutt har jeg laget funksjoner for å generere datasett ut av diverse matematiske modeller, slik som kuler, sylindere og lignende.

6.2 HDF

Hierarchical Data Format (HDF) er et multi-objekt filformat for overføring av numeriske data mellom maskiner og er en vanlig standard innen vitenskapelig visualisering. HDF støtter seks forskjellige

datamodeller, og kan lagre bilder, arrayer, tabeller osv. HDF er også et selvbeskrivende format, som tillater en applikasjon å tolke strukturen og innholdet uten informasjon fra utsiden. HDF er også portabelt og kan deles over forskjellige plattformer.

For å lese HDF filer brukte jeg C++ biblioteket HDF4.1r5 som er et HDF4 bibliotek. Dette biblioteket inneholder alle verktøyene man trenger for å lese og skrive HDF filer.

7. Konklusjon

I denne hovedoppgaven har jeg studert hvordan en voxelrenderer fungerer i praksis og teori. Jeg har fra grunnen av laget en voxelrenderer ved å bruke OpenGL som verktøy for voxelmotoren, og Qt som verktøy for brukergrensesnittet. Jeg har sammenlignet mine resultater med andre eksisterende kjente volumrenderere, slik som den som ligger i VTK.

Voxelrendereren benytter seg av teksturmetoden. Denne metoden gir gode resultater som i tillegg både er rask og blir effektivt akselerert av 3D hardware. Dette bidrar til en høy grad av interaktivitet fra brukerens side, noe som er viktig innen vitenskapelig visualisering.

En god fargetabell er viktig for å kunne manipulere gjennomsiktighet og farge i volumet, for å isolere data som man ønsker å undersøke, og for å oppnå effekter slik som randfordunkling. En slik fargetabell er komplisert og tar tid å lage, derfor lagde jeg kun enkle versjoner av denne teknikken.

For å kunne rendre datasett over flere tidssteg kan det være nødvendig å rendre dette til fil. For hvert tidssteg må nye teksturer genereres og dette tar tid. Jeg knyttet derfor rendereren til Python, som gjør det mulig å scripte animasjoner, der informasjon til rendereren går via kommandolinjen.

Så lenge det som skal visualiseres har data fordelt på et uniformt kartesisk grid, er det relativt enkelt å visualisere med min voxelrenderer. Så sant datasettet ikke er for stort går det greit å visualisere det i sanntid, selv på billig 3D hardware som er vanlig i de fleste hjemmedatamaskiner. Når dataen er animert derimot, er det vanskelig å visualisere dette i sanntid, pga. de store datamengdene som må leses fra disk til minne. I disse tilfellene må det genereres nye teksturer for hvert tidssteg, noe som tar relativt lang tid. Her er det derfor best å rendre animasjonen til disk først, for så å spille den av i for eksempel MPEG format.

Jeg gjorde flere sammenligninger mellom VTKs voxelrenderer og min voxelrenderer og sammenlignet metoder, hastigheter, visuelle resultater og kode. Fra disse resultatene konkluderte jeg med at å lage en renderer fra grunnen av i OpenGL gir en raskere renderer enn hva VTK kunne gjøre. Raskere renderer gjør at det er lettere å interaktere med datasettet og man kan oppdage nye ting som man kanskje ikke ville oppdage hvis programmet gikk for tregt. I VTK derimot kan man visualisere et volum på ytterst få kodelinjer i forhold til antall linjer som måtte til i OpenGL rendereren. I tillegg har man i VTK tilgang til mange filtere og teknikker, slik som isoflater og Raytracing.

Videre utvikling av voxelmotoren kan ligge i å kode en bedre fargetabell, samt utnytte moderne hardwares nye programmerbare flyttalls pipelines, som kan benyttes meget bra til vitenskapelig visualisering. På denne måten kan enda mer av CPU kraften overføres til skjermkortet, og mer ytelse blir frigjort til for eksempel å kjøre simuleringer som genererer animasjon, og visualisere dette i sanntid.

Referanser

[1] WILL SCHROEDER, KEN MARTIN, BILL LORENSEN. The Visualization Toolkit.

[2] P.K. KUNDU. Fluid Mechanics.

[3] Silicon Graphics, Inc. OpenGL Red Book

[4] The Visualization Toolkit, VTK homepage
http://www.vtk.org/

[5] HANS PETTER LANGTANGEN. Computational Partial Differential equations.

[6] A. SANNA, B. MONTRUCCHIO, P. MONTITUSCHI. A survey on visualization of vector fields by texture-based methods.

[7] JOHN D. OWENS, BRUCEK KHAILANY, BRIAN TOWELS, WILLIAM J. DALLY. Comparing Reyes and OpenGL on a Stream Architecture

[8] PHILIPPE LACROUTE, MARK LEVOY. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Tranformation.

[9] PAOLO SABELLA. A Rendering Algorithm for Visualizing 3D Scalar Fields.

[10] ROBERT A. DREBIN, LOREN CARPENTER, PAT HANRAHAN. Volume Rendering.

[11] RALF BÖNNING, HEINCRICH MÜLLER. Interactive Sculpturing and Visualization of Unbounded Voxel Volumes.

[12] MARK J. KILGARD. Realizing OpenGL: Two Implementations of One Architecture.

[13] PRAVEEN BHANIRAMKA, Yves Demange. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets.

[14] VICTORIA INTERRANTE, Chester Grosch. Strategies for Effectively Visualizing 3D Flow with Volume LIC.

[15] JIAN HUANG, YAN LI, ROGER CRAWFIS, SHOA-CHIUNG LU, SHUH-YUAN LIOU. A Complete Distance Field Representation.

[16] SARGUR N. SRIHARI. Representation of Three-Dimensional Digital Images.

[17] GEORGE S. CARSON. Standard Pipeline The OpenGL Specification

[18] KEVIN KREEGER, ARIE KAUFMAN. Hybrid Volume and Polygon Rendering with Cube Hardware.

[19] STEFFEN PROHASKA, HANS-CHRISTIAN HEGE. Fast Visualization of Plane-Like Structures in Voxel Data.

[20] INGMAR BITTER, ARIE KAUFMAN. A Ray-Slice-Sweep Volume Rendering Engine.

[21] HANSPETER PFISTER, JAN HARDENBERGH, JIM KNITTEL, HUGH LAUER, LARRY SEILER. The VolumePro Real-Time Ray-Casting System.

[22] FRANK DACHILLE IX, KLAUS MUELLER, ARIE KAUFMAN. Volumetric Backprojection.

[23] MALTE ZÖCKLER, DETLEV STALLING, HANS-CHRISTIAN HEGE. Interactive Visualization of 3D-Vector Fields using Illuminated Stream Lines.

[24] RICHARD S. GALLAGHER, JOOP C. NAGTEGAAL. An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volumes.

[25] Trolltech Qt hompage
http://www.trolltech.com/