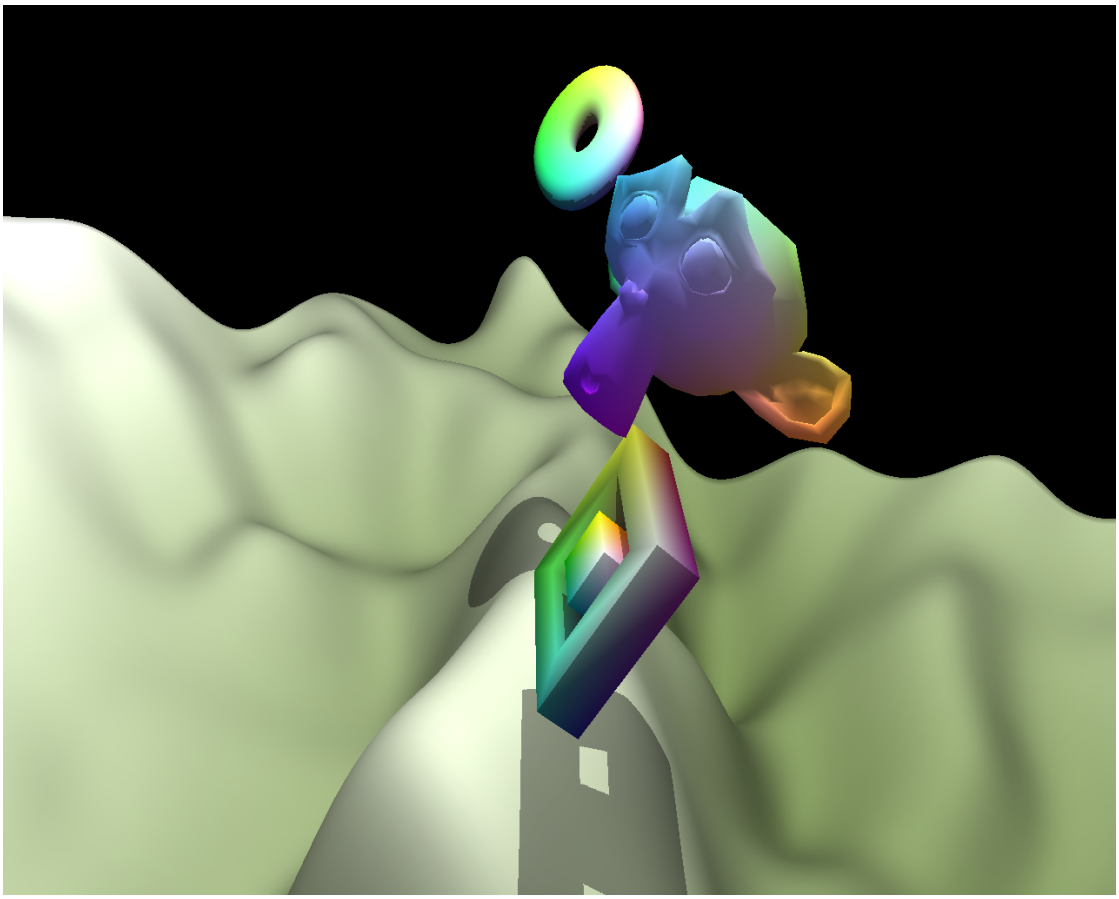
# Prosjektrapport
# TDT4230 Visualisering



*Simon Jonassen*

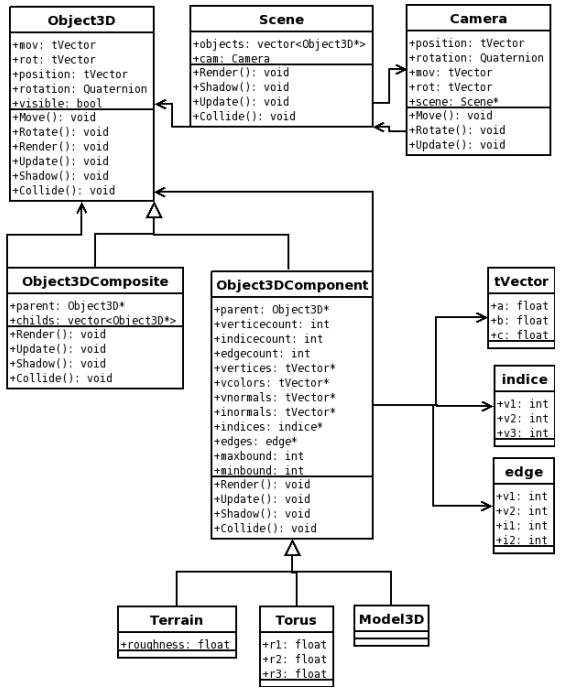
Norges teknisk-naturvitenskapelige universitetet
Institutt for datateknikk og informasjonsvitenskap

# Innhold

# 1 Oppsett og initialisering

Generelt oppsett skjer ved å bruke OpenGL Utility Toolkit. Lys og fogsettings er veldig standarde, verdiene ble justert etter litt eksperimentering. Videre brukes det en rekke tilbakekallsfunksjoner til fremvisning, oppdatering, mus- og tastaturhendelser. Alle generelle innstillinger settes opp i `Visualisering.cppi`.

# 2 Scenegraf




Figur 1: Klassediagram for scenegraf

Scenegraf (se Fig. 1) er definert og implementert i `Object3D.h` og `Object3D.cpp`. Klassehierarkiet som ble implementert består av tre nivå: først er det en **Object3D** klasse som har attributter som hastighetsvektorer for bevegelse og rotasjon, posisjonsvektor, rotasjonskvaternion, en boolsk variabel som bestemmer om objektet skal kaste skygge eller ikke. Objektets rotasjon og translasjon defineres i det globale koordinatsystemet. Det er også en rekke funksjoner som brukes til å rotere, translere, oppdatere (i forbindelse med animasjon), punktkollidere og rendre selve objektet og tilhørende skyggebånd. Hver av disse funksjonene blir forklart senere.

**Object3DComposite** og **Object3DComponent** brukes til å realisere et ordnet objekthierarki i scenen. Begge klasser har en peker til et forelder-Object3D (eller en nullpeker dersom det er ingen) og overkjører metoder Update, Render, Shadow og Collide. Ob-

ject3DComposite har en liste (en std vektor) av Object3D-instanser; alle oppdaterings-, kollisjons-, og renderingskall vil delegeres til kompositttobjektets barn (det utføres en rekke koordinattransformasjoner først, bla. kameraposisjontransformasjoner og oversetting av globale koordinater til kompositttobjektets lokale koordinater).

Object3DComponent er mer kompleks; den er ment til å beskrive 3d-objekter som består av primitiver som hjørner, polygoner/indices, kanter. Har også en rekke lister som inneholder polygonnormaler, hjørnenormaler, hjørnefarger og variabler som oppgir størrelsene til disse (edgecount, indicecount, verticecount). I tillegg til det overkjøres metodene for visning og kollisjonsdeteksjon spesifikk til intern objektstruktur. Disse metodene er veldig generelle og blir forklart senere, men de kan også overkjøres av klasser som arver fra Object3DComponent.

Object3DComponent arves av tre klasser til: **Torus**, **Terrain** og **Model3D**. De første brukes til å illustrere bruk av periodiske og ikkeperiodiske B-Splines. Den siste klassen brukes til å realisere 3D-modeller fra en tekstformat slik det blir forklart senere.

Til slutt har vi en kameraklasse og en sceneklasse. Scenen består enkelt av en liste med objekter og en kamerainstans. Kameraklassen beskriver posisjonering og rotasjon til visningsvindu, også initialiserer punktkollisjon med objektene i scenen ved oppdatering.

# 3 Polygonstruktur

Polygonstruktur som ble brukt har en del svakheter siden f.eks. hjørnene er bare posisjonsvektorene, mens indiceobjektene er bare tripler av hjørneindekser (indeks til hvor i tabellen hjørnene er lagret) og lagres separat fra tilsvarende polygonnormal. Allikevel er valget tatt for å oppnå høyere utelse og fleksibilitet i kode. Primær idé bak dette er å implementere noe som kalles VertexArrays. Ved bruk av denne teknikken kan alle polygoner rendres ved å sende pekere til arrays hvor hjørnene, normalene og fargene er lagret. Og så kalle glDrawElement med GL_TRIANGLES og pekeren til trekanttabellen. VertexArrays er både mye raskere og mye enklere enn å tegne en og en triangel. Oversetting mellom polygonprimitiver og vertexarrays skjer ved å kaste en liste av objekter til en liste av flytall. Dette medfører imidlertidig at klassene ikke kan ha noen andre attributter enn de som skal brukes under rendringen.

En av mulige utvidelser er å realisere *Vertex Buffered Objects*. En videregående bruksområde for VertexArrays hvor data kan lastes direkte til skjermkortbuffer. Ytelsen vil øke dramatisk, spesielt for statiske objekter med mange polygoner. Kompleksiteten vil økes tilsvarende.

Alle hjørner/punkter er bare en **tVector**'er av tre koordinater. Alle polygoner består av tre hjørneindekser. Alle kanter består av to hjørneindekser og to polygonindekser. Normal- og fargelistene inneholder vektorer indeksert som tilsvarende hjørnene og polygonene. Flatenormaler regnes på forhånd og brukes senere under kollisjonsdeteksjon og skyggebåndkonstruksjon. For tegning brukes det hjørnenormaler og hjørnefarger, det gir glattere/finere effekt pga bedre muligheter for interpolasjon.

# 4 3D Modeller

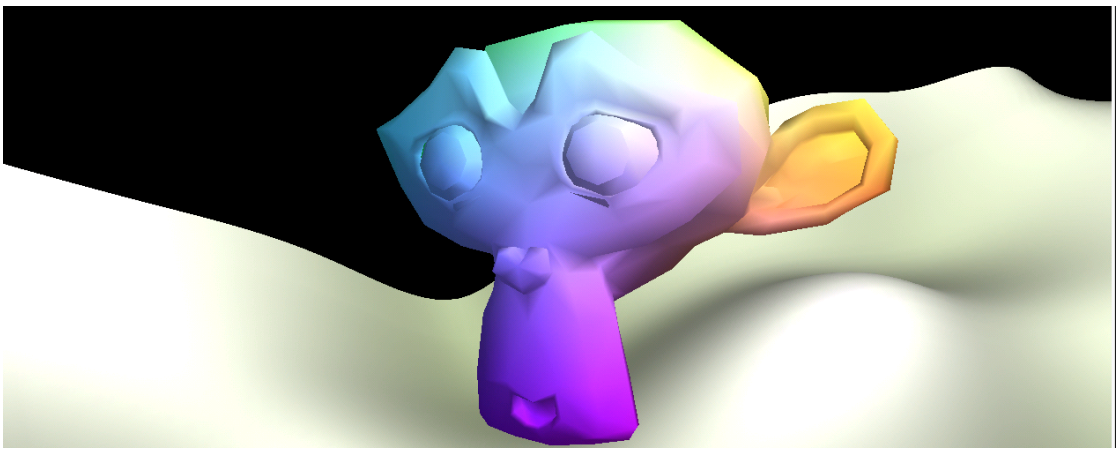
Det ble implementert støtte for innlasting av 3D modeller i meget forenklet versjon av *Wavefront OBJ File Format* [dI06] som støttes av blant annet **Blender**. Eneste to instruksjoner som parses er `v float float float` som beskriver et hjørne med tre koordinater og `f int int int ...` som beskriver en flate med et vilkårlig antall hjørner (tre eller flere).

Parsingen er veldig enkel: først leses alle hjørner og lagres inn i en stdlib *vector*. Deretter leses alle flater, første tre tall brukes til å danne første triangel, hver etterkommende tall kombineres med to forrige og danner et triangel til; flateinstruksjon med $n$ heltall danner $n-2$ trinagler. Viktig å huske at verdiene som leses inn er 1-indeksert med data som lagres er 0-indeksert. Samtidig beregnes flate og hjørnenormalene (de siste skal normaliseres senere). Videre brukes algoritme 1 til å finne alle kanter i modellen. Til slutt kopieres data fra listene inn i nyallokerte arrays.

**Data**: A list of indices of size $n$
**Result**: A list of edges

**for** $i1$=*0* **to** $n$ **do**
  **for** $i2$=$i1+1$ **to** $n$ **do**
    **foreach** $v1$ *in* $i1$ **do**
      **foreach** $v2$ *in* $i2$ **do**
        **if** $v1$ *equals the next vertex to* $v2$ *and* $v2$ *equals the next vertex to* $v1$ **then**
          add ($v1$,$v2$,$i1$,$i2$) to edges;

**Algorithm 1**: Enkel kantsøking



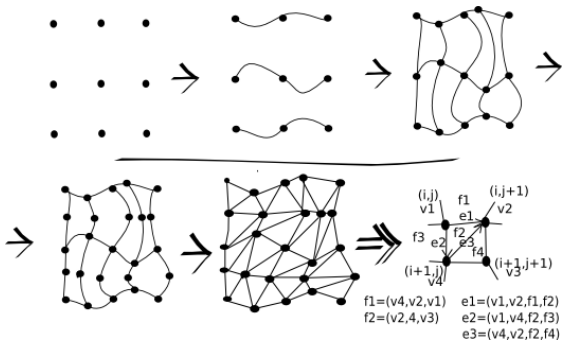
Figur 2: Blender Monkey, 507 hjørner, 968 triangler, 1431 kanter

Blant mulige forbedringer kan nevnes utvidelse av instruksjoner som støttes. Tekstur og materialstøtte. Også hadde det hadde vært veldig nyttig å implementere en eller en annen algoritme som kan gjøre lavpolygonmodeller om til modeller med en høyere flateoppløsning (kan tenke en spline lignende teknikk for flater og naboflater, slik at en flate kan på en måte bøyes og deles opp i flere flater).

# 5 Splines

$$\vec{p}_i(u) = \frac{1}{6} \begin{vmatrix} u^3 & u^2 & u & 1 \end{vmatrix} \begin{vmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{vmatrix} \begin{vmatrix} \vec{p}_{i-1} \\ \vec{p}_i \\ \vec{p}_{i+1} \\ \vec{p}_{i+2} \end{vmatrix} \tag{1}$$

Det ble implementert to typer uniforme B-splines: periodiske og ikke-periodiske (se `Bspline.h`). Forskjellen mellom disse ligger prinsipielt i at for periodiske vil kontrollpunkter *wrapes* rundt (dvs. tar kontrollpunkter fra andre enden hvis den går utenfor kanten) ,dette ble implementert ved hjelp av enkel moduloaritmetikk, men den andre type vil bare duplisere endepunktet i en slik situasjon. Metoden som det ble implementert utnytter det at spline-flater kan tenkes som separable. Det som skjer er at kontrollpunktene brukes først til å generere en mengde med midlertidige kontrollpunkter etter (1); det brukes en skaleringsfaktor sx som bestemmer hvor detaljert samplingen av kurven blir, sx sampler per segment. Deretter brukes de nye kontrollpunktene til å generere punkter i en annen retning på samme måte som i første tilfelle. Resultatgrid er $(l_x s_x) \times (l_y s_x)$, sx parameter velges i konstruktoren til objektene som illustrerer teknikken. Figur 3 illustrerer overgangene.




Figur 3: Splineprosessering

Begge splinemetoder generer også hjørne- og flatenormaler, liste over trekanter. Flatefunksjon for periodiske splines beregner også kantene ved å bruke enkel moduloregning til å finne indeks til tilsvarende flater; hovedtanken er at *(pos+offs+len)%len* gir et tall mellom 0 og len uansett hva pos og offs er. Figur 3 illustrerer også dette prinsippet. Derimot blir dette mye vanskeligere for ikke periodiske flater, en mulig løs-

ning her er å finne alle kanter som har to flater og så finne de gjenstående kantene. Indeksaritmetikk blir mye mer komplisert i dette tilfelle (nok å si en gang at det er $3 * (lx * sx - 1) * (lx * sx - 1) + lx * sx + ly * sx - 1$ kanter totalt!). I sted for dette kan brukes algoritme 2 som er en spesialisert versjon av algoritme 1, hovedtanken er å avslutte innerste løkke med en gang en flate som deler kanten blir funnet, derimot hvis det ikke finnes en slik flate settes indeks for den andre flate til -1.

```
Data: A list of indices of size n
Result: A list of edges
for i1=0 to n do
    foreach v1 in i1 do
        found ← False;
        for i2=i1 + 1 to n do
            foreach v2 in i2 do
                if v1 equals the next vertex to v2 and v2 equals the next vertex to v1
                then
                    add (v1,v2,i1,i2) to edges;
                    found ← True;
                    proceed on next v1;
        if not found then
            add (v1,v2,i1,-1) to edges
```

**Algorithm 2**: Utvidet kantsøk

Siden kantgenerering for ikkeperiodiske splines er veldig ueffektiv ble tilsvarende kode ekskludert fra splinegenering og står bare som en kommentar i konstruktoren til terreng i `Object3D.cpp`. Det er mulig å fjerne kommentarmarkering og bruke koden. Allikevel vil dette gjøre initialisering veldig tidskrevende (siden det skal genereres $O(lx^2 ly^2 sx^4)$ kanter). Slik det er nå vil bare edgecount til terrengen settes til 0 og ingen kanter blir generert, det går bra siden disse ikke brukes uansett.

Koden mangler en tredje type, semiperiodiske splines ,en som kan brukes til å generere konvekse volumobjekter. Der spline wraper bare to maskekanter, mens for de to andre trekkes bare til et gjennomsnittspunkt.

**Torus** ble generert fra et grid med 4x4 kontrollpunkter og det kan oppgis en vilkårlig skaleringsfaktor. sx 10 eller 20 virker veldig bra, men skyggen på selve objektet blir allikevel litt hakkete. sx 1 vil bare lage en veldig minimalistisk torus ;)

**Terreng** kan genereres for et vilkårlig antall kontrollpunkter og en vilkårlig skaleringsfaktor. Kontrollpunktverdiene genereres runtime etter (2).

$$
\begin{aligned}
p(0,0) &= rand(-r,r) \\
p(i+1,0) &= p(i,0) + rand(-r,r) \quad i > 0 \\
p(0,j+1) &= p(0,j) + rand(-r,r) \quad j > 0 \\
p(i+1,j+1) &= (p(i+1,j) + p(i,j+1))/2 + rand(-r,r) \quad i,j > 0
\end{aligned} \tag{2}
$$

En mulig forbedring her er å bruke Brownian Modell [HB04] i stedet og gange ut tabellverdiene med en Gaussisk maske for å glatte ut landskapet ut til kantene.

# 6 Kvaternioner og rotasjonsmatriser

Til å implementere rotasjon ble det brukt kvaternioner (se `Quaternion.h`). Generelt kan det sies at en kvaternion kan representeres enten som et hyperkompleks tall som består av et reel og tre komplekse komponenter eller som et rotasjonsvinkel-rotasjonsvektor par.

$$
q = (\theta, \vec{v}) = (\cos(\theta/2), \sin(\theta/2)\frac{v.x}{|\vec{v}|}, \sin(\theta/2)\frac{v.y}{|\vec{v}|}), \sin(\theta/2)\frac{v.z}{|\vec{v}|}) \tag{3}
$$

$$
q^{-1} = (q_0, -q_1, -q_2, -q_3) \tag{4}
$$

$$
normalized(q) = \frac{q}{|q|} \tag{5}
$$

$$
\begin{aligned}
(q_0^a, q_1^a, q_2^a, q_3^a) * (q_0^b, q_1^b, q_2^b, q_3^b) = normalized(&q_0^a q_0^b - q_1^a q_1^b - q_2^a q_2^b - q_3^a q_3^b, \\
&q_0^a q_1^b + q_1^a q_1^b + q_2^a q_3^b - q_3^a q_2^b, q_0^a q_2^b + q_2^a q_1^b + q_3^a q_1^b - q_1^a q_3^b, q_0^a q_3^b + q_3^a q_1^b + q_1^a q_2^b - q_2^a q_1^b)
\end{aligned} \tag{6}
$$

Rotasjonen tilsvarer kvaternionmultiplikasjon (6) og den skjer i tre akser separat fra hverandre (7-9). Det er viktig å skille mellom intern (10) og ekstern (11) rotasjon; forskellen ligger nemlig i at rekkefølgen til operandene under multiplikasjon blir i motsatt rekkefølge. Kamerarotasjon er en intern rotasjon, mens objektrotasjon er ekstern.

$$
q_{roll}(\theta) = (\cos(\theta/2), 0, 0, \sin(\theta/2)) \tag{7}
$$

$$
q_{pitch}(\theta) = (0, \cos(\theta/2), \sin(\theta/2), 0, 0) \tag{8}
$$

$$
q_{yaw}(\theta) = (\cos(\theta/2), 0, \sin(\theta/2), 0) \tag{9}
$$

$$
q_{r,p,y}^{ext} = q_{roll}(r) q_{pitch}(p) q_{yaw}(y) q_{original} \tag{10}
$$

$$
q_{r,p,y}^{int} = q_{original} q_{yaw}(y) q_{pitch}(p) q_{roll}(r) \tag{11}
$$

En veldig viktig detalj i implementeringen rotasjonsmatrisen (12) som genereres av **rmat_gl(rmat)** er *row major* (etter OpenGL spesifikasjon) og ikke *colum major* slik man kan forvente. Siden rotasjonsmatrisene er ortogonale (det er lett å observere at $R^T = R^{-1}$) vil **rmat_gl** gi den inverse matrise til en *column major* rotasjonsmatrise. Dette er en

veldig nyttig observasjon som brukes til å unngå eksplisitt kvaternioninvertering under skyggerendring, selv om dette kan medføre en misforståelse av kode til en viss grad.

$$\mathbb{R}_{(q_0,q_1,q_2,q_3)} = \begin{vmatrix} 1-2(q_2^2+q_3^2) & 2(q_1q_2-q_0q_3 & 2(q_0q_2+q1_q3 & 0 \\ 2(q_1q_2+q_0q_3) & 1-2(q_1^2+q_3^2) & 2(q_2q_3-q_0q_1) & 0 \\ 2(q_1q_3-q_0q_2) & 2(q_0q_1+q_2q_3) & 1-2(q_1^2+q_2^2) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{12}$$

Det ble også implementert et enkelt vektorbibliotek, `Vector.h`, som tar seg av enkle operasjoner på tredimensjonale vektorer, blant annet addisjon, subtraksjon, skalarmultiplikasjon og -divisjon, kryss- og dotprodukter. Headerfilen inneholder også en enkel mxnxk-matrisemultiplikasjon og matriseinvertering; Siden det ble brukt bare 3x3 matriser er inversen definert på eksplisitt form (13). Ellers så kunne det implementeres metoden basert på Gaussisk eliminasjon en kjenner fra TMA4115 Matematikk 3 [EP88].

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}^{-1} = \frac{1}{a(ei-fh)-b(di-fg)+c(dg+eg)} \begin{vmatrix} ei-fh & ch-bi & bf-ce \\ fg-di & ai-cg & cd-af \\ dh-eg & bg-ah & ae-bd \end{vmatrix} \tag{13}$$

# 7 Animasjon og styring

Til å animere objekter brukes det en rotasjons og en translasjonsvektor. Ved oppdatering vil rotasjonskvaternion og translasjonsvektor oppdateres med en verdi spesifisert av disse. Rotasjon og translasjon av kamera er avhengig av brukerinnput som ble nevnt tidligere.

Rotasjonskvaternionen brukes til å hente ut 3 vektorer gitt etter hovedakser [1] til objektet (14). Hver av disse aktorene ganges med akselerasjon i dette tidspunktet (en vast verdi) og brukes til å akkumulere er hastighetsvektor definert. For å få litt bedre effekt er rotasjonen og bevegelsen til kamera dempet - når en knapp er trykket vil mov vektoren akkumuleres som det ble nevnt, ellers avtar den med 5% for hver prosesseringssykel. Det samme gjelder rotasjon. Som resultat blir alle bevegelsene glattere og dette virker litt mer *realistisk*.

$$\vec{forward} = \mathbb{R}_q(0,0,\vec{-1}) \quad \vec{upward} = \mathbb{R}_q(0,\vec{-1},0) \quad \vec{left} = \mathbb{R}_q(\vec{-1},0,0) \tag{14}$$

Kollisjonstesting og kollisjonsrespons skjer ved oppdatering, selve prosedyren blir beskrevet i detalj senere. De fleste objektene i scenen har tilfeldig valgt rotasjon ved starten for å demonstrere animasjon.

## 7.1 Kamerastyring

Musbevegelsene brukes til å styre pitch og roll til kamera. Tastaturhendelsen slår på og av enkle flags i en bitvektor som bestemmer kameraoppførsel. For å kunne håndtere flere samtidige tastetrykk brukes det enkel bitvektorlogikk. Følgende mapping ble brukt:

[1] *principal axes*; fram/tilbake, opp/ned og mot venstre/høyre

**w**- flytte kamera fram
**s**- flytte kamera bak
**a**- flytte kamera til venstre
**d**- flytte kamera til høyre
**z**- flytte kamera opp
**x**- flytte kamera ned
**q**- yaw-rotere ccw
**e**- yaw-rotere cw
**f**- slå på/av tåke
**k**- avslutte (kill!)

# 8 Kollisjonsdeteksjon

Kollisjonsdeteksjon skjer ved oppdatering av kameraposisjon. For å detektere kollisjoner brukes kameraets posisjon i forrige tilstand og posisjon som ble økt med hastighetsvektor. Disse to er definert i globale koordinater og sendes til en kollisjonsrutine i scenen som da vil iterere over sine objekter og endre sluttposisjonen etter hvert. For hvert objekt i scenen vil start- og sluttkoordinater oversettes til lokale koordinater (15) ved første omgang, deretter vil disse enten sendes videre til objektets barn hvis det er et kompositt objekt eller brukes til selve kollisjonsberegningen hvis det er et komponentobjekt.

$$\mathbb{P}_w = \mathbb{T}^{obj}_{w\leftarrow l}\mathbb{R}^{obj}_{w\leftarrow l}\mathbb{P}_l \quad \mathbb{P}_l = \mathbb{R}^{obj}_{l\leftarrow w}\mathbb{T}^{obj}_{l\leftarrow w}\mathbb{P}_w \tag{15}$$

For komponentobjekter vil punkt-til-plan avstand (16) sjekkes først. Hvis den har ulike fortegn for start- og sluttpunkt, vil disse ligge på ulike sider av flateplan. I så fall settes det opp en Moore's og Wilhelms' kollisjonsligning 17, siden linjen krysser planet så finnes det eksakt en løsning. Hvis $u$ og $v$ tilfredsstiller (18) ligger kollisjonspunktet innenfor triangelet.

$$d(p_{xyz}, f_{abcd}) = \vec{n}_f \cdot \vec{p}_p - \vec{n}_f \cdot \vec{p}_f \tag{16}$$

$$\vec{p}_s + (\vec{p}_f - \vec{p}_s)t = \vec{p}^{\,f}_0 + (\vec{p}^{\,f}_1 - \vec{p}^{\,f}_0)u + (\vec{p}^{\,f}_2 - \vec{p}^{\,f}_0)v \tag{17}$$

$$t = [0,1]; u \geq 0; v \geq 0; u + v \leq 1 \tag{18}$$

Ligningen (17) kan omskrives til vektorform (19) og deretter til matriseform (20).

$$\vec{p}_s - \vec{p}^{\,f}_0 = (\vec{p}_s - \vec{p}_f)(t, \vec{0}, 0) + (\vec{p}^{\,f}_1 - \vec{p}^{\,f}_0)(0, \vec{u}, 0) + (\vec{p}^{\,f}_2 - \vec{p}^{\,f}_0)(0, \vec{0}, v) \tag{19}$$

$$\vec{p}_s - \vec{p}^{\,f}_0 = \begin{pmatrix} \vec{p}_s - \vec{p}_f \\ \vec{p}^{\,f}_1 - \vec{p}^{\,f}_0 \\ \vec{p}^{\,f}_2 - \vec{p}^{\,f}_0 \end{pmatrix} (t, \vec{u}, v) \tag{20}$$

$$(t, \vec{u}, v) = \begin{pmatrix} \vec{p}_s - \vec{p}_f \\ \vec{p}^{\,f}_1 - \vec{p}^{\,f}_0 \\ \vec{p}^{\,f}_2 - \vec{p}^{\,f}_0 \end{pmatrix}^{-1} (\vec{p}_s - \vec{p}^{\,f}_0) \tag{21}$$

Løsningen (21) til matriseligningen (20) oppnås ved kan å gange begge sider med en inverse matrise (13) foran. Hvis $u$ og $v$ tilfredsstiller kondisjonene over er kollisjonspunktet gitt med $\vec{p}_s + t * (\vec{p}_f - \vec{p}_s)$.

Det ble implementert enkel kollisjonsrespons som går ut på å ta projeksjon av vektoren mellom kollisjonspunktet og sluttpunktet på flatenormal og trekke den dobbelte av den fra sluttpunktet. Dette tilsvarer et fullt elastisk støtt en kjenner fra 3FY.

Til slutt skal sluttpunktet funnet i lokale koordinater konverteres tilbake til globale og erstatte den tidligere verdien for sluttposisjonen. Dette gjelder både kompositt og komponentobjekter.

En svakhet med kollisjonsresponsen er at den kan resultere i kollisjonsfeil: dersom kamera kolliderer med flere objekter samtidig kan et av dem frastøtte kamera inn i et annet objekt (et som ble kollisjonstestet tidligere). Dersom det ikke hadde vært noen kollisjonsrespons så kunne problemet løses enkelt ved å returnere det allers første kollisjonspunktet. Derimot i løsningen som ble implementert vil kollisjon med flere objekter samtidig være veldig problematisk i en rekke av enkelte tilfeller, allikvel virker den som regel feilfritt.

I tillegg til det kan objektene i scenen kjøre inn i kameraet dersom kameraet står stille. Dette kunne fikses ved å separere tilstander og bruke det forrige for å konvertere startposisjonen og det neste for sluttposisjonen. Allikevel kan dette bli veldig unøyaktig. Av den grunn virker bare kollisjonstesting mellom kamera og objekter i scenen, men ikke omvendt.

# 9 Rendering

Framvisningen skjer i flere steg: først slås GL_VERTEX_ARRAY, GL_COLOR_ARRAY og GL_NORMAL_ARRAY på og alle objekter i scene tegnes. Så slås disse av og stencil buffer settes opp. Videre rendres alle skyggebånd til stencil buffer slik det blir forklart senere. Så brukes stencil buffer til å generere og tegne et semigjennomsiktig bilde foran kameraet. Til slutt skal alle innstillinger settes tilbake.

```
Rotate viewpoint with an inverse to cam. rotation ;
Translate viewpoint with an inverse to cam. position ;
Let there be light!;
foreach object i scenen do
    Translate viewpoint to objects position;
    Roterer viwepoint to objects rotation;
    Rendrer object;
    Undo rotation;
    Undo translatio;
Undo translation;
Undo rotation;
```

**Algorithm 3**: Visningstransformatjonser

Alle visningstransformasjoner ved rendringen av både objekter og skyggebånd skjer

etter algoritme 3. Det er viktig at lyset skal settes opp etter steg to. I så fall gir det en statisk lyskildepunkt, ellers vil lyset defineres relativt til kamera og ikke relativt til scenen. Translasjon er implementert ved å bruke **glTranslate3f()** og rotasjon implementert ved bruk av kvaternioner som genererer en rotasjonsmatrise som sendes deretter til OpenGL ved å kalle **glMultMatrixf(rmat)**. Rotasjon og translasjon kunne regnes om til en transformasjonsmatrise og eliminere glTranslate3f på den måte, men det vil antakeligvis gi lavere ytelse enn det er nå. I tillegg til det kunne transformasjonsmatrise beregnes ved oppdateringen av objekter og ikke under rendringen, dette ville resultere i høyere ytelse for scener med mange statiske objekter.

All kode som ble beskrevet her befinner seg i `Visualisering.cpp` og ulike **Render()**-metoder i `Object3D.cpp`.

## 10 Skygger

Den enkleste renderingsmetode som kunne brukes er XOR. Alle skyggebånd rendres til stencilbuffer med både front og backfaces, bufferverdien inverteres ved depth-pass (dvs. om skyggebåndet ligger nærmere enn objektet bak). Til slutt rendres stencil buffer til et plan foran kameraet.

For å finne hvilke kanter som skal danne skyggebånd brukes det to dotprodukter mellom vektorer fra lyskilde til kantmidtpunktet og flatenormalene. Dersom begge dotprodukter har ulike fortegn er en av flatene vendt mot lyset mens den andre ikke er det, ergo kanten skal kaste skygge. I så fall skal det lages en projeksjon av kanten med lysvektoren ganget med et stort heltall. Dette danner et skyggebånd (en ulempe er at skyggebånd kan ha feil faceretning, blir forklart senere).

Det er imidlertidig tre store problemer med skyggelegginsalgoritmen som ble brukt:

1. Alle polygoner blir helsvarte, og skyggen på selve objektet blir veldig hakkete.

2. Når skygger fra forskjellige objekter eller fra det samme objektet legges over hverandre vil skyggeverdien inverteres.

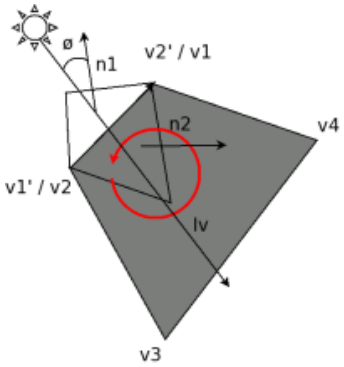
3. Skyggeverdier inverteres dersom kameraet befinner seg i skyggevolumet.

Alle tre problemer kan teoretisk sett fikses selv om det har en del vanskeligheter som ikke lar dette implementeres på en enkel måte.

Det første problemet kan fikses ved å f.eks. sammenkjede kanter og bruke bare annenhvert hjørne til tegning av skyggebånd.

Det andre problemet kan løses ved å bruke en teknikk som kalles '*z-pass method*' eller '*depth-pass method*'. Teknikken går ut på å nullstille stencilbuffer, rendre alle frontface'ne til skyggevolumer og øke verdiene i bufferen ved test-pass, så rendre alle backface'ne og minske bufferverdiene ved test-pass [Kwo06]. Resultatet skal kunne håndtere overlappende skygger siden sluttverdien blir positiv. Ulempe er at alle kantene må reverseres avhengig av relativ posisjonering av til lyskilden og flatenormalen som er vendt mot den. Denne regelen er veldig enkel egentlig: hvis face1 er vendet mot lyset (altså dotprodukt

mellom normalvektoren og lysvektoren er negativ) skal rekkefølgen på hjørnene byttes om! Figur 4 illustrerer prinsippet.




Figur 4: Kantretningredigering for skygger

Problemet med løsningen over er at den ikke klarer å håndtere skyggeinvertering når kameraet befinner seg inne i volumet. Dette kan fikses ved å bruke *depth-fail method* også kjent som *Carmack's Reverse*. Den er omtrent motsatt av den forrige: først rendres alle backfaces og verdien økes ved test-fail, så rendres frontfaces og verdien minskes ved z-fail. Resultatet stemmer uavhengig av hvor kameraet befinner seg. I tillegg til kravet om riktig kantretning kommer det også et krav om at skyggevolumet skal være lukket. Dette kan implementeres ved å spore opp kanter og finne sykler, så tegne et polygon som da skal lukke skyggevolumet [Kwo06].

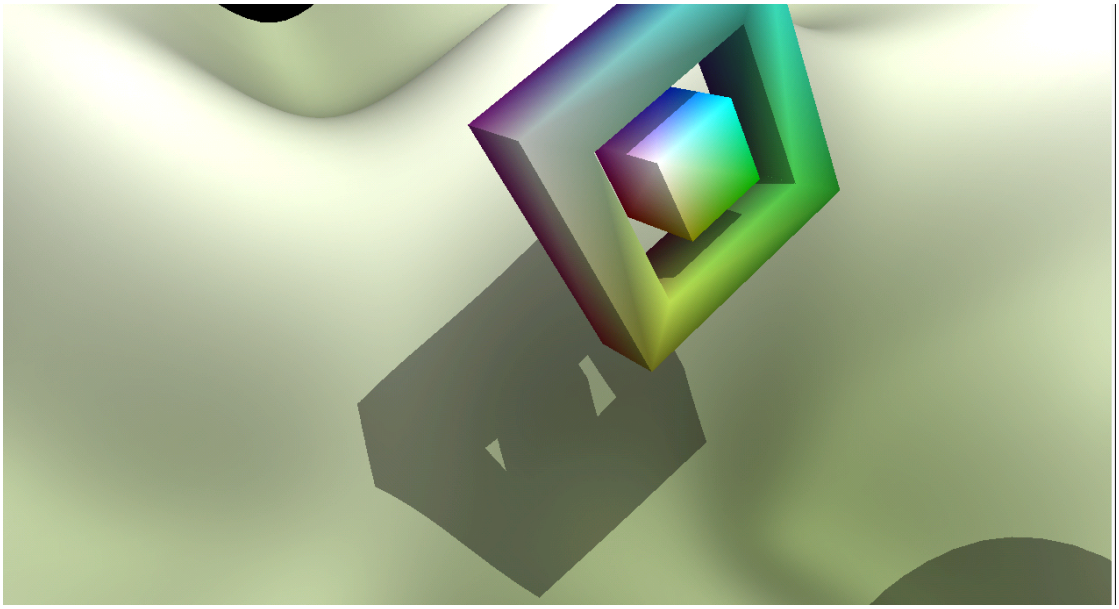
På grunn av manglende tid (dvs. i forbindelse med samtidig innlevering av Datamaskinprosjektet) ble det implementert **z-pass**-algoritmen med bruk av OpenGL kode indirekte tatt fra NeHE [CP06]. Skygger er hakkete og virker ikke når kamera kommer inn i skyggevolumet. Som framtidig arbeid kan tenkes implementering av Carmack's Reverse (dessverre er den patentert i tillegg [2]). En annen utvidelse er å implementere *soft shadows*, altså skygger med mer diffuse kanter. Dette kan gjøres enten ved å plassere flere lyskilder ved siden av hverandre og rendre skygger for hver av dem. Eller ved å bruke en av glattingsteknikkene vi kjenner fra Bildeteknikk på selveste bufferen. Siste metode er mer effektiv, mens den første vil gi et mer realistisk resultat.

Som det ble bemerket før blir matrisen som kommer ut av **rmat_gl** allerede invertert dersom den skal brukes til å transformere punktet manuelt. Av den grunn vil $\vec{p_l} = \mathbb{R}^{-1}\mathbb{T}^{-1}\vec{p_g} = \mathbb{R}_{GL}\mathbb{T}^{-1}\vec{p_g}$.

All kode som implementerer beregning av skyggebånd gitt i `Object3D.cpp`. Generell OpenGL kode for oppsett av skyggebuffer er indirekte tatt fra NeHe [CP06] og reprodusert i `Visualisering.cpp`.

Også viktig å legge merke til at den første rendringen skjer med lyset. Det hadde vært mer effektivt om rendringen skjedde bare med ambient lys først, så skyggetest, så rendre

[2] fy!



Figur 5: *z-pass shadowcasting*

diffus og spekular lys i de områdene som ikke har skygge i stencil buffer (dvs. verdien blir 0).

Alle objekter har en boolsk variabel *visible* som bestemmer om objektet kaster skyggen eller ikke. For komposittobjekter skal lyskildeposisjon oversettes til lokale koordinater og så vil skyggevolumer for alle objekter med *visible* satt til *true* rendres.

## 11 Etterord

Alle filer som ble nevnt ligger under `lizz` på den CD'en som følger med. Katalogen også har en makefile og prosjektet kan kompileres og kjøres under Linux ved å bruke **make**. For å kunne bruke dette under Windows følger det med en `vizz` katalog som inneholder Visual Studio prosjektfiler og ferdigkompilert program under `vizz/debug`.

Eksempeloppsett inneholder flere objekter: et torus med skaleringsfactor 10, en terreng 15×15 med skaleringsfactor 10 og amplitudefactor 300, en apefjesmodell (også kjent som *blender monkey*) eksportert fra Blender 2.41 og et komposittobjekt kombinert av en torus med skaleringsfaktor 1 og en 3D modell av en kube. Alle objekter bortsett fra apefjeset og terrenget kaster skygge. Apefjesen, koposittobjektet og torusen roterer med tilfeldigvalgt rotasjon og skyggen projeseres mot landskapet. Kollisjon kan skje mellom kamera og alle objekter i scenen. Fargene på alle objektene bortsett fra terrenget er randomgenerert, på terrengen brukes det interpolert høydefarging.

All kode er skrevet av Simon Jonassen og kan modifiseres og reproduseres under betingelsene av GPL [Inc91].

## Referanser

[CP06] Banu Octavian (Choko) and Brett Porter. Nehe shadowcasting tutorial. `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=27`, 2006.

[dI06] File Format dot Info. Wavefront obj file format summary. `http://www.fileformat.info/format/wavefrontobj/`, 2006.

[EP88] Edwards and Penney. *Elementary Linear Algebra*. Prentice-Hall, 1988.

[HB04] Hearn and Baker. *COMPUTER GRAPHICS with OpenGL (Third Edition)*. Pearson, 2004.

[Inc91] Free Software Foundation Inc. General public licence. `http://www.gnu.org/licenses/gpl.txt`, 1991.

[Kwo06] Hun Yen Kwoon. The theory of stencil shadow volumes. `http://www.gamedev.net/reference/articles/article1873.asp`, 2006.