OH-BILDER

DATORGRAFIK HT 2005

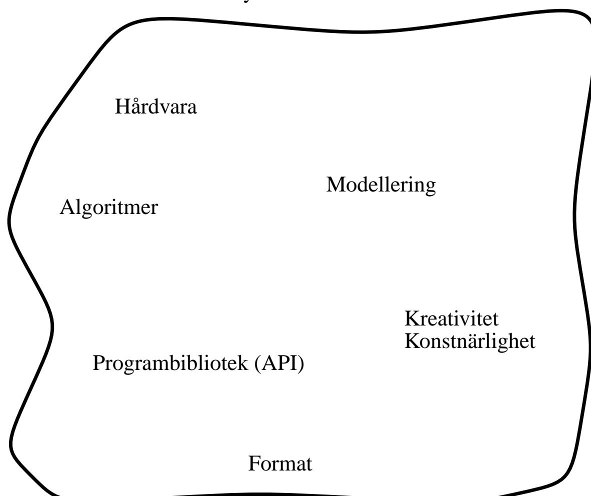
Bok ej nödvändig, men kan ge mer kött på benen.

Dessa kopior av OH-bilder tillsammans med ett antal småskrifter avses ge nödvändiga fakta. OH-bilder som enbart belyser teorin finns inte alltid med. Inte heller OH-bilder som innehåller kod från andra utdelade dokument.

Datorgrafik

Datorproduktion av synliga modeller av verkligheten eller virtuella miljöer. Visualisering (alltifrån affärsgrafik och reklamgrafik till s k vetenskaplig visualisering). Modeller för träning eller upplevelse. Ev animerade. Edutainment. Multimedia.

Grunden är matematik och fysik.



Centralproblemet: Avbilda en 3D-värld (ev tillverkad) på en platt skärm eller dyl på ett realistiskt sätt. Jfr fotografering.

Datorgrafik - Användningsområden

- CAD - Computer Aided Design
- Modernt användarsnitt
- Information (visualisering)
  - Statistik, affärsgrafik
  - Vetenskaplig visualisering
- Reklam
  - I press, TV, film
- Underhållning
  - TV, film
  - Spel
  - Skärmsläckare
- Utbildning
  - Färdighetsträning
  - Edutainment (=lek och lär)

Datorgrafik - Litteratur etc

Böcker (pris 2004 `www.amazon.co.uk` exkl frakt/moms,£1=13.5)

- **Hill**: Computer Graphics Using OpenGL, 2nd ed, Prentice Hall 2000
  Tar upp gammalt som nytt med många detaljer. C++. HC £37(f å).
- **Möller**/**Haines**: Real-Time Rendering, 2nd ed, A K Peters, 2002
  Titeln säger allt - fort skall det gå. Välskriven sammanfattning av modernt vetande med en trevlig webbsida. HC £38.95.
- **Angel**: Interactive Computer Graphics with OpenGL, 3rd ed, Addison-Wesley, 2003. PB 44.99. Datorgrafik och OpenGL i ett svep, men kändes tidigare mager. Den nya upplagan bättre.
- **Shreiner**/**Woo**/**Neider**/**Davis**: OpenGL Programming Guide, 4th ed, Addison-Wesley, 2003. PB £32.19.
- **Hearn**/**Baker**: Computer Graphics with OpenGL, 3rd ed, Prentice Hall, 2003. PB £42.99.
  Tidigare använd lärobok. Känns även i nya upplagan litet gammal.
- **Foley**/**van Dam**: Computer Graphics, 2nd ed, Addison-Wesley,1990
  En gång datorgrafikens bibel. HC £41.99 (2003).
- **Watt**/**Policarpo**: 3D Games, vol 1: Real-Time Rendering and Software Technology, Addison-Wesley, 2000. HC £36.99 (2003).
  Watt är en multiförfattare som nu gett sig på spel. Även vol 2.

Tidskrifter

- SIGGRAPH Computer Graphics
- IEEE Computer Graphics and its Applications
- ACM Transactions on Graphics
- EuroGraphics Computer Graphics Forum

Litteratur på nätet i obegränsade mängder

- Företag/organisationer/universitet/privat, spec `www.opengl.org`

Datorgrafik - Utveckling 1(1)

Återgivning

Streckfigurer (tråddiagram)

Fyllda ytor (dolda ytproblemet)

Texturerade ytor

Fotorealism (ljus, skugga)

Animering

Ljud

Multimedia

Nätbaserat

I telefon el dyl, stereo, VR (virtual reality)

Interaktion

Alltid varit centralt. Men utrustningen blivit allt mer sofistikerad.
Haptik (återkoppling - feedback - i form av virtuell känsla).
`http://haptic.mech.northwestern.edu/intro/gallery/`

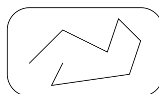
Datorgrafik - Utveckling hårdvara 1(1)

Presentationsutrustning

Analog teknik

Skrivare: Pennplottrar (kurvritare)

Skärm: Oscilloskopteknik, t ex Tektronix. Bilden (bara den) ritas oupphörligt.

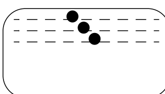


1980

Digital teknik

Skrivare: Laserskrivare (1986)
Bläckstråleskrivare

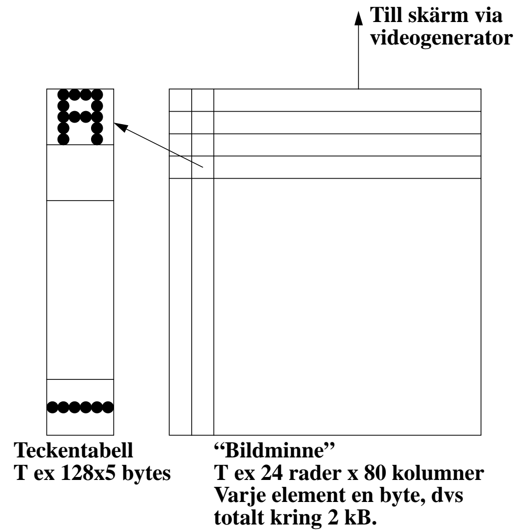
Skärm: TV-teknik (videosignal från bildminne)



Tid

Jfr Håkan Lans patentstrider (patent inlämnat 1979). Men det var mycket sådant i luften då.

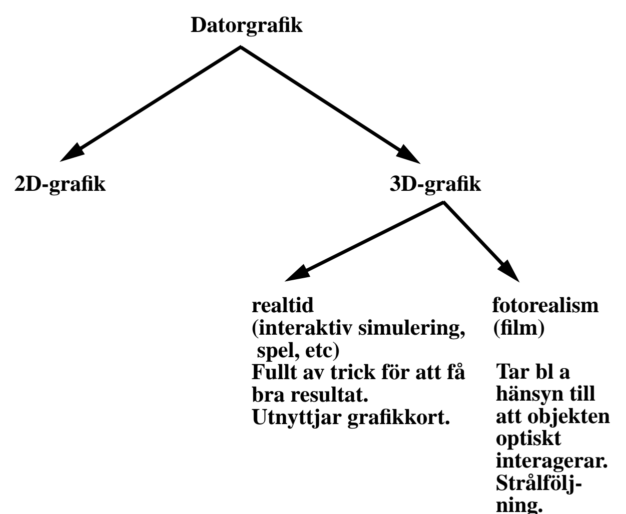
Datorgrafik - textterminaler kring 1980



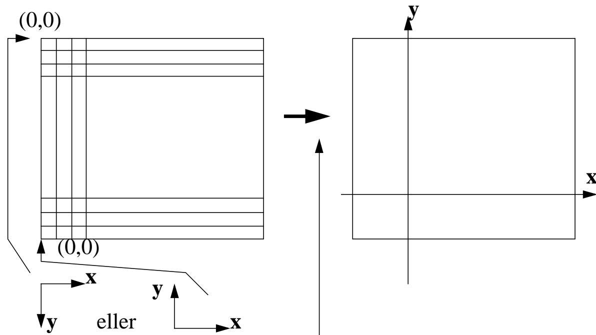
S k semigrafik (motsv Text-TV) möjlig

Numera är alla datorer "grafiska".

Datorgrafik - en kategorisering



2D-grafik

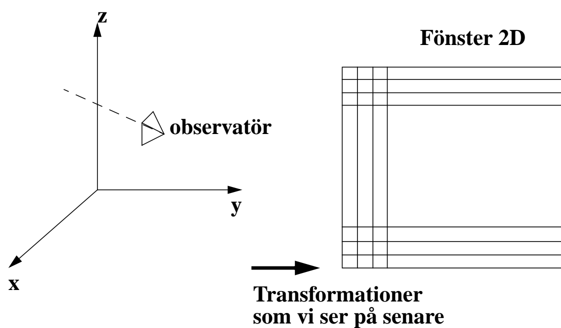


Diskret heltalskoordinatsystem (raster). Alltid i botten för all datorgrafik.

Användardefinierat reellt koordinatsystem (man anger värden för fönster-gränserna)

Alltid möjligt med enkla medel (se senare)

3D-grafik



Värld med
föremål
observatör (kamera)

Vad behövs?
observatörens position
tittriktning
uppåtriktning
synfält (vinkel el dyl)

Datorgrafik - Programvara 1(1)

Användare

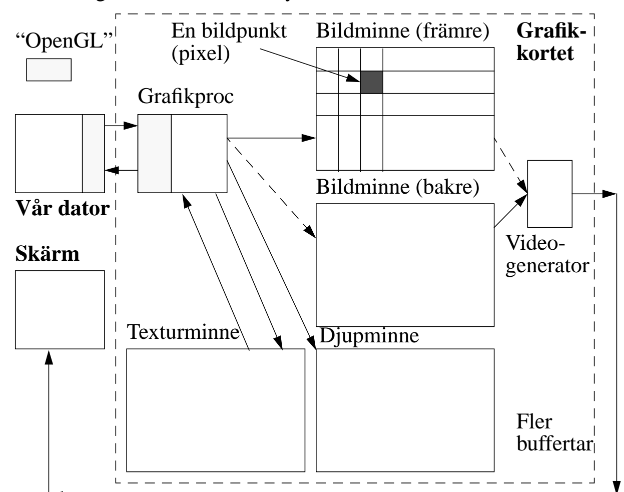
• 2D-grafik
Ritprogram (Photoshop, Paintshop, Gimp etc)
Dokumentprogram (FrameMaker, Word, OpenOffice)
• 3D-grafik
CAD
Matematikprogram (MATLAB etc), kalkylprogram
Spel etc
Modelleringsprogram (Alias, Maya, 3DStudio, ...)
Multimediaprogram

Utvecklare - programmerare

• 2D-grafik
(Lång historia! Enhetsberoende, Tektronixgrafik, ...)
X, Windows, QuickDraw, Java, PostScript, SVG
• 3D-grafik
(Lång historia! GKS, PHIGS, PEX, XGL)
OpenGL med påbyggnader
DirectX
Java 3D
Andra grafikmotorer (bibliotek). T ex finns kommersiella grafik/fysik-motorer som Havok, Novodex, Meqon (svensk). ODE är en fri fysikmotor. Spelföretagen har ofta interna spelmotorer.

Datorgrafik - hårdvara 1(1)

Så här fungerar det i stort i ett system med skärm som utenhet:

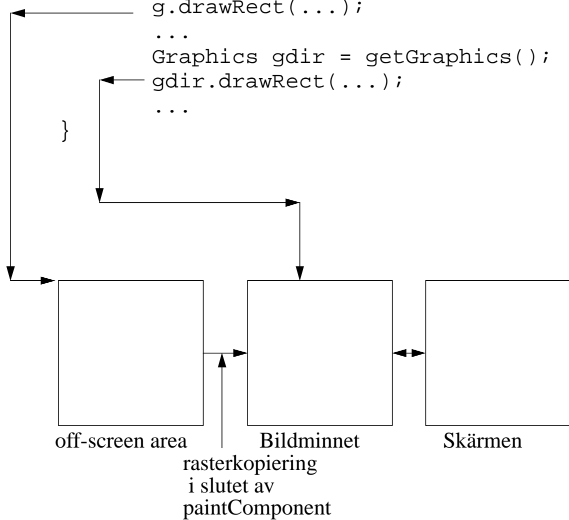


- Typiskt skickar vår dator punkter, linjeändpunkter eller triangelhörn till GPU'n som rastrerar och fyller med färg.
- Uppdatering till skärm sker från bildminnet minst 60 ggr/sekund.
- En modern GPU klarar (i princip) 10-30 miljoner trianglar per sekund, dvs bortåt 0.5 miljoner trianglar dynamiskt i realtid. Kräver även snabb CPU. Och ofta plattformsberoende kodning!
- Bildminnet av storleksordningen 1000x1000x4 bytes = 4 MB

Hur i Java - delvis odokumenterat 1(1)

All ritning skall ske i komponentens `paintComponent`:

```
public void paintComponent(Graphics g)
{
        ...
        g.drawRect(...);
        ...
        Graphics gdir = getGraphics();
        gdir.drawRect(...);
        ...
}
```



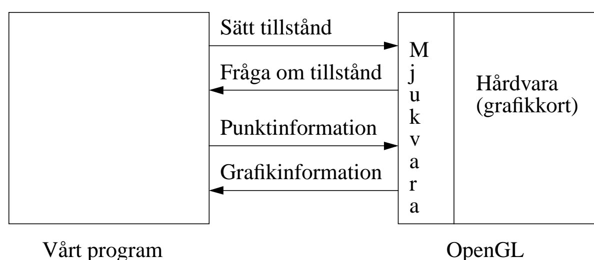
Konsekvens: Enbart g => ev väntan, enbart gdir => direkt ritning men ej korrekt uppdatering. Båda => OK.

OpenGL - i korthet 1(1)

Referens: Häftet “Introduktion till OpenGL”

OpenGL är

- ett bibliotek för 3D-grafik
- en tillståndsmaskin
- en pipelinemaskin, som i mer eller mindre hög grad kan implementeras i hårdvara
- plattformsoberoende (men språk som använder OpenGL är det inte)

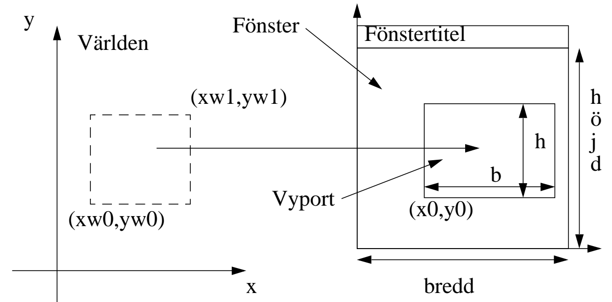


Konkurrent: Microsofts multimediasystem DirectX (inkluderar Direct 3D), nu i version 9. Konkurrensen välgörande. DirectX bara för Windows, medan OpenGL för SGI, Sun Solaris, Linux, Windows, Mac, ...

OpenGL - 2D-grafik

Kommentarer till ex 1 i “Introduktion till OpenGL”.

- `glutInitWindowSize(bredd, höjd)` bestämmer fönsterstorleken i bildpunkter.
- `glViewport(x0, y0, b, h)` bestämmer vilken del - vyport- av fönstret som vi vill utnyttja. Parametrarna är heltal. Ty-



  piskt hela fönstret, dvs `glViewport(0,0,bredd,höjd)`.
- `glOrtho(xw0,xw1,yw0,yw1,-1,1)` bestämmer vilken del av den tvådimensionella världen som skall avbildas på “vyporten”. Parametrarna är godtyckliga reella tal. Kan placeras i uppdateringsproceduren om vi vill ändra del, t ex för panorering/zoomning.
- Vi kan ange hörnen med `glVertex2i(heltalsparametrar)` eller `glVertex2f(reella parametrar)`.

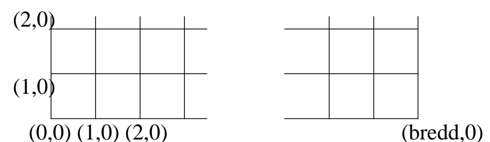
OpenGL - 2D-grafik - heltalskoordinater

Vi kan använda OpenGL som ett rent 2D-system med heltalskoordinater (X, Windows och Java är sådana!).

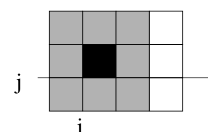
Vi låter då

- Vyporten sammanfalla med fönstret
- Delen av världen sammanfalla med fönstret, dvs `glOrtho(0,bredd,0, höjd, -1,1)`

Bildpunkterna numreras efter koordinaterna för nedre vänstra hörnet



Antag att x = i + dx, y = j + dy, där i och j är heltal och $0 \le dx, dy < 1$. Normalt (punktstorleken = 1) avser då `glVertex2i(i,j)` och `glVertex2f(x,y)` bildpunkten (i,j). Vid ritning med `glBegin(GL_POINTS); ... ; glEnd();` kommer den bildpunkten att markeras. Punktstorleken kan sättas med `glPointSize(storlek)`. Om storleken är udda > 1 kommer ytterligare punkter att markeras symmetriskt kring denna. T ex om storleken=3. (För jämn storlek, se litt.)



Kursplanen vid CTH

Syfte
Kursen avser att ge grundläggande kännedom om utrustning, programvara och principer för framställning av bilder i olika former med hjälp av dator (datorgrafik).

Innehåll
Teori: Översikt av tillämpningsområden för datorgrafik. Egenskaper hos utrustning för grafisk presentation. Rastrering illustrerad med algoritmer för generering av linjer. Två- och tredimensionella transformationer. Övergång från en 3D-värld till en 2D-bild. Dolda ytor. Fotorealism: Belysningsmodeller, strålföljning och radiositetsmetoden, texturering med tillämpningar. B-splines/NURBS för kurvor och ytor. Grafik i andra system (t ex PostScript, MATLAB, VRML, Java, X). Lagringsformat. Fraktaler och kaos. Animering. Beräkningsgeometri. Effektiviseringar. Utblickar mot virtuell verklighet och multimedia. Vertex- och fragmentprogram.
Programvara: Den etablerade grafiska standarden OpenGL (bibliotek för 3D-grafik) används för att illustrera många begrepp. Dessutom möter du bl a modellerings- och strålföljningsprogram.

Organisation
Undervisningen består av föreläsningar och laborationer. Laborationerna innebär konstruktion och utnyttjande av grafisk programvara, eventuellt i anslutning till tillämpningsområde av särskilt intresse för den studerande.

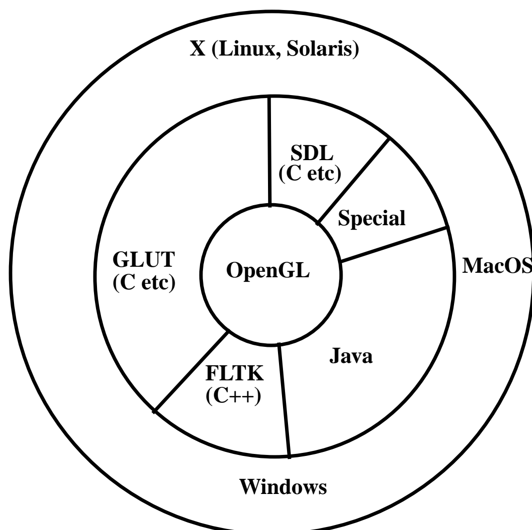
Litteratur
Kursen täcks med diverse småskrifter och OH-material. Vid senaste genomförandet av kursen rekommenderades bl a Hill: Computer Graphics Using OpenGL, 2nd ed, Prentice-Hall 2000 alternativt Akenine-Möller/Haines: Real-Time Rendering, 2nd ed, A K Peters 2002. Aktuell information om litteratur ges före kursstart på kursens webbsida.

Examination
Skriftlig tentamen. För slutbetyg fordras även godkända laborationer. Betygskala: U, 3, 4, 5.

Förkunskaper
Du måste ha kunskaper i något av de vanliga programspråken (Java, C/C++, Ada eller Pascal) och känna till datalogiska begrepp som listor och träd. I kursen finns matematiska inslag (men den har inte matematisk betoning). Du måste därför ha tillägnat dig kunskaper i linjär algebra (vektorer, matriser och transformationer). Kursen ger dig chansen att repetera dem och se dem praktiskt tillämpade.

Koppling Fönstersystem <-> OpenGL

Innerst grafiksystemet, sedan kopplingen (inkl ev GUI m m) och ytterst fönstersystemet



I X-fallet heter speciallösningen GLX. GLUT för X anropar internt GLX. I Windows används i stället WGL.

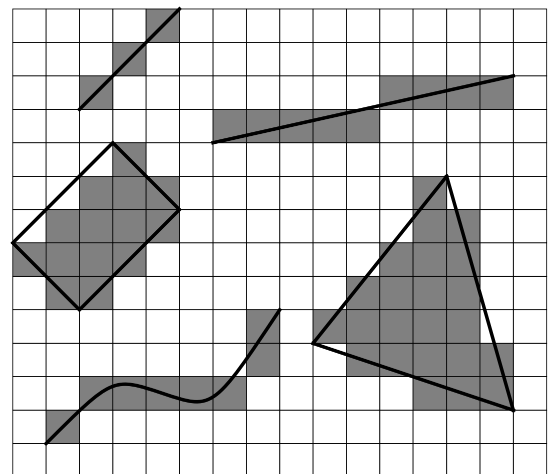
GLUT <-> JOGL (Java bindings for Open GL))

Funktion	GLUT	JOGL
OpenGL	glColor3d(...)	gl.glColor3d(...)
GLU	gluPerspective(...)	glu.gluPerspective(...)
Omritningsbegäran	glutPostRedisplay()	Automatiskt med Animator, annars display-anrop
Idle-funktion	glutIdleFunc(...)	Animator+display()
Mushantering	glutMouseFunc(...)	MouseListener
Tangenthantering	glutKeyboardFunc(..)	KeyListener
Omritning	glutDisplayFunc(...)	display()
Omskalning	glutReshapeFunc(...)	reshape(...)

I GLUT-fallet anger man i de fyra sista fallen namnet på den procedur som utför jobbet som parameter till anropet. Detta görs som en initieringsåtgärd. I JOGL-fallet placeras motsvarande metoder i vår underklass till *GLEventListener*.

Rastrering

För 2D-grafik är rastreringen den stora stötestenen. Att för olika primitiver avgöra vilka bildpunkter som skall markeras.



Låt oss titta en aning på det för linjer.

Rastrering - Linjer 1(3)

1. **Förutsättningar**

- Låt $\Delta x = x_L - x_0$ och $\Delta y = y_L - y_0$, där (x_0,y_0) är startpunkten och (x_L,y_L) är slutpunkten.
- Vi förutsätter att ändpunkterna har heltalskoordinater.
- Vi antar också att linjens genomloppsriktning är från vänster till höger, dvs att $\Delta x > 0$ samt att linjens riktningsvektor ligger i första oktanten, dvs att $k = \Delta y/\Delta x <= 1$. Annorlunda uttryckt "lutar linjen mest i x-led".
- Vi låter bildpunkternas mittpunkter ha heltalskoordinater (ej så i OpenGL).

Vi kan beskriva linjen med $f(x,y) = 0$, där

$$f(x,y) = y - kx + d$$

med k enligt ovan och där d är en konstant.

2. **Enkel algoritm**

- 1. Upprepa för $x = x_0, x_0+1, ..., x_L$
  1. Beräkna det reella talet
     $y = kx - d = (\Delta y/\Delta x)x - d$
  2. Markera bildpunkten (x, round(y)).

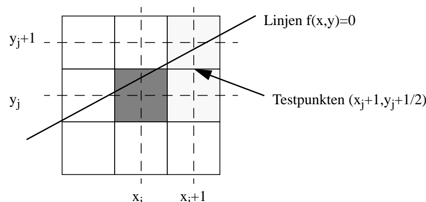
Nackdel: Flyttalsaritmetik. Men det kanske inte numera är en nackdel.

3. **Bresenhams algoritm**

Vi eftersträvar av "historiska" skäl en heltalsalgoritm och tar fram den med den s k mittpunktsmetoden, som är en generell metod för att konstruera kurvritningsalgoritmer.

Rastrering - Linjer 2(3)

Givet att punkten $P_j=(x_j,y_j)$ markerats gäller det att avgöra vilken av de två kandidatpunkterna (x_j+1,y_j) och (x_j+1,y_j+1) som skall väljas (svagt prickade i figuren nedan). Vi baserar valet på $\mathbf{q_j} = 2\Delta x f(x_j+1,y_j+1/2)$. Faktorn $2\Delta x$ gör att uttrycket är ett heltal, vilket man lätt förvissar sig om genom att beräkna d och stoppa in eller av det följande.



Om $\mathbf{q_j}>=0$, ligger "mittpunkten" $(x_j+1,y_j+1/2)$ ovanför den matematiska linjen, dvs den undre bildpunkten skall väljas. Om $\mathbf{q_j}<0$, väljs i stället den övre.

Talen $\mathbf{q_j}$ kan lätt beräknas successivt under kurvföljningen med heltalsaritmetik enligt

1. Välj $P_0=(x_0,y_0)$ och beräkna
   $\mathbf{q_0}=2\Delta x f(x_0+1,y_0+1/2)=2\Delta x f(x_0,y_0)-2\Delta y+\Delta x$, dvs
   $\mathbf{q_0} = \mathbf{\Delta x - 2\Delta y}$ (ty startpunkten ligger på linjen)
2. Om $\mathbf{q_j >= 0}$ väljs $P_{j+1}=(x_j+1,y_j)$ och då blir
   $\mathbf{q_{j+1}} = 2\Delta x f(x_j+2,y_j+1/2) = 2\Delta x f(x_j+1,y_j+1/2) - 2\Delta y = \mathbf{q_j - 2\Delta y}$
   Om $\mathbf{q_j < 0}$ väljs $P_{j+1}=(x_j+1,y_j+1)$ och då blir
   $\mathbf{q_{j+1}} = 2\Delta x f(x_j+2,y_j+3/2) = 2\Delta x f(x_j+1,y_j+1/2) - 2\Delta y + 2\Delta x =$
   $\mathbf{q_j + 2(\Delta x - \Delta y)}$

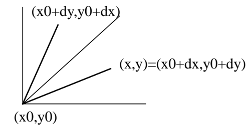
Rastrering - Linjer 3(3)

Vi kan alltså med heltalsaritmetik bestämma punkterna som skall markeras.

Ev sifferexempel.

4. **Anpassning av Bresenhams algoritm till andra oktanter**

Sker lätt med t ex reflektion



(vilket kostar några extra operationer) eller genom att man modifierar algoritmen (vilket inte kostar extra).

Rastrering - Cirkel och Ellips

Heltalsalgoritmer av Bresenham-typ finns även för cirklar och ellipser (inkl snett orienterade, se t ex Foley/van Dam).

Uppritning av allmänna kurvor 1(3)

En **kurva i 2D** kan matematiskt beskrivas på några olika sätt

- **Standardform** (explicit form): $y = f(x), a \le x \le b$
- **Implicit form**: $F(x,y) = 0$.
  Ex: Enhetscirkeln
  $$F(x, y) = x^2 + y^2 - 1$$
- **Parameterform**: $\left.\begin{array}{l} x = x(t) \\ y = y(t) \end{array}\right\}, \quad t \in [0,1]$
  Ex: Enhetscirkeln
  $$x = \cos(2\pi t), y = \sin(2\pi t), \quad 0 \le t \le 1$$

Den explicita formen kan naturligtvis ses som ett specialfall av parameterformen.

För **kurvor i 3D** är parameterformen den man möter:

$$\left.\begin{array}{l} x = x(t) \\ y = y(t) \\ z = z(t) \end{array}\right\}, \quad t \in [0,1]$$

Man ritar upp en allmän kurva på parameterform (vi tittar på det implicita fall senare) genom att approximera den med ett polygontåg. Dvs vi inför ett antal delningspunkter längs kurvan och ritar streck mellan dem. Enklast är att göra en likformig indelning av parameterintervallet [0,1].

Uppritning av allmänna kurvor 2(3)

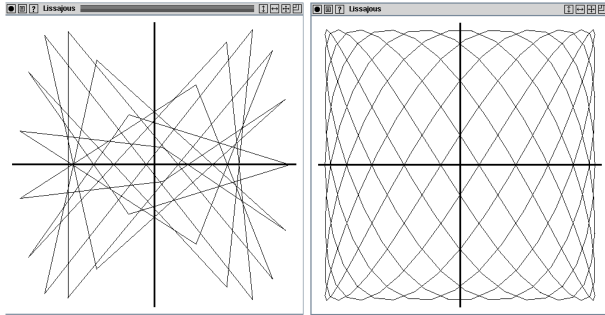
Algoritm (i 2D):

- 1. Låt N vara antalet delningspunkter
- 2. Placera pennan i startpositionen (x(0),y(0))
- 3. Upprepa fr o m i=1 till N
  - 3.1 Beräkna t = i/N
  - 3.2 Drag ett streck från pennans tidigare position till (x(t),y(t))

Exempel: Lissajouskurvan

$$x = \cos(14\pi t), y = \sin(22\pi t), \quad 0 \le t \le 1$$

uppritad med N = 25 resp N = 250:

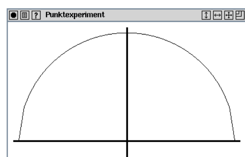


För fullgott resultat måste N väljas stort nog.

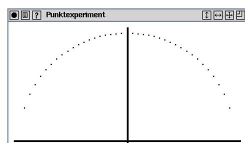
Uppritning av allmänna kurvor 3(3)

Exempel: Halvcirkeln $x^2 + y^2 = 1$, y≥0, uppritad med N=40 utifrån

$$y = \sqrt{1 - x^2}$$



Ser rätt hyggligt ut. Om vi använder punkter i stället ter sig resultatet mycket sämre. Vi ser att approximationen blir ojämn.



I just detta fall skulle vi få en jämn punktfördelning om vi utgick från parameterframställningen.

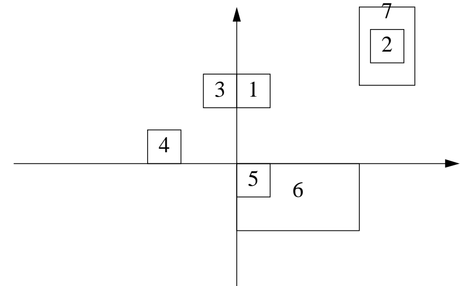
Det finns många andra sätt att rita en cirkel.

Approximera med linjer! Inte med punkter!

2D-Transformationer 1(6)

Vi är intresserade av tre speciella transformationer

- **Translation**: Alla punkter i ett objekt flyttas en viss bit i x-led och en viss bit i y-led.
- **Skalning**: Alla punkter i ett objekt töjs/krymps i förhållande till en vald fixpunkt (i eller utanför objektet).
- **Rotation**: Alla punkter i ett objekt roteras i förhållande till en vald fixpunkt (i eller utanför objektet).



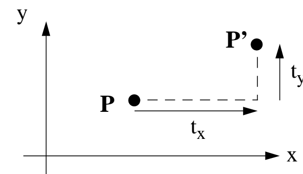
Kvadraten 2 i figuren har vi fått genom att translatera kvadrat 1. Kvadrat 3 genom att rotera kvadrat 1 kring sitt undre vänstra hörn. Kvadrat 4 genom att rotera kvadrat 1 runt origo. Rektangel 6 genom att skala kvadrat 5 med origo som fixpunkt. Rektangel 7 genom att skala kvadrat 2 med mittpunkten som fixpunkt.
Vår lycka: För objekt uppbyggda av linjer och polygonytor räcker det att transformera ändpunkterna respektive hörnen. Linjer bevaras ju.

2D-Transformationer: Translation. 2(6)

Låt oss här och i fortsättningen beteckna den ursprungliga punkten

med $\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}$ och den nya med $\mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}$.

Vid translation transformeras alla punkter enligt figuren,



dvs

$$x' = x + t_x$$
$$y' = y + t_y$$

eller

$$\mathbf{P}' = \mathbf{P} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

2D-Transformationer: Skalning. 3(6)

Vi antar att skalningen sker med avseende på origo, vilket innebär att alla x-värden multipliceras med en skalningsfaktor s_x och alla y-värden med en annan s_y.

Således

$$x' = s_x x$$
$$y' = s_y y$$

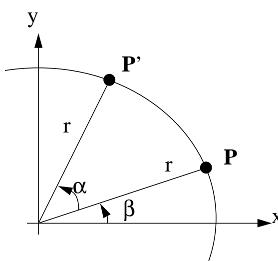
eller

$$\mathbf{P}' = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \mathbf{P}$$

2D-Transformationer: Rotation. 4(6)

Vi antar att rotationen sker kring origo. Vi räknar vinkeln moturs.

Ur figuren finner vi



att

$$x' = r\cos(\alpha + \beta) = r\cos\alpha\cos\beta - r\sin\alpha\sin\beta = x\cos\alpha - y\sin\alpha$$
$$y' = r\sin(\alpha + \beta) = r\sin\alpha\cos\beta + r\cos\alpha\sin\beta = x\sin\alpha + y\cos\alpha$$

Vi kan skriva detta på matrisform

$$\mathbf{P}' = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \mathbf{P}$$

2D-Transformationer: Homogena koordinater. 5(6)

Ofta följer ett antal transformationer på varandra. Det vore bekvämt om alla transformationer kunde representeras med matriser, ty då kunde man ersätta följden av transformationer med produkten av motsvarande matriser. Rotation och skalning kan efter vad vi sett representeras med matriser. Men det gäller inte translationer utan vidare. Men genom att vi inför s k homogena koordinater går det.

En punkt $P=(x,y)^T$ sägs ha de **homogena koordinaterna**

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ eller allmännare } \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} \text{ där } w > 0.$$

Om vi nu låter **P** och **P'** avse punkterna med homogena koordinater (med w=1) ser man att för translation

$$\mathbf{P}' = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

För skalning och rotation får vi ur de gamla formlerna

$$\mathbf{P}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{P} \text{ resp } \mathbf{P}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

dvs vi har uppnått ett enhetligt sätt att representera transformationer med matriser.

2D-Transformationer: Skalning och rotation kring godtycklig punkt. 6(6)

Låt fixpunkten vara $\mathbf{F} = \begin{bmatrix} f_x \\ f_y \end{bmatrix}$. Skalning resp rotation med avseende på fixpunkten kan då åstadkommas genom att

1. Fixpunkten (och resten av världen) translateras till origo
2. Skalning resp rotation sker m a p origo.
3. Fixpunkten translateras tillbaka.

Den totala transformationen kan därför representeras med matrisen,

$$\begin{bmatrix} 1 & 0 & f_x \\ 0 & 1 & f_y \\ 0 & 0 & 1 \end{bmatrix} M \begin{bmatrix} 1 & 0 & -f_x \\ 0 & 1 & -f_y \\ 0 & 0 & 1 \end{bmatrix}$$

där M är den tidigare skalnings- respektive rotationsmatrisen. Den som så önskar kan naturligtvis multiplicera ihop delarna till en enda matris.

Translation och rotation bevarar avstånd och vinklar samt parallella linjer. Skalning bevarar enbart parallella linjer. Alla linjära transformationer som bevarar parallella linjer kallas **affina transformationer**. Ytterligare en speciell sådan transformation, **skjuvning** (eng. shearing), brukar nämnas i läroböcker, men vi har inget behov av den.

Man kan normalt inte byta ordning på två transformationer. Exempel.

3D-Transformationer: Translation och skalning. 1(3)

Translation och skalning erbjuder inga nya utmaningar, bortsett från att en parameter tillkommer (t_z resp s_z). En punkt i homogena koordinater ser nu ut så här:

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Vi inser direkt att vid translation är

$$\mathbf{P'} = \begin{bmatrix} x+t_x \\ y+t_y \\ z+t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

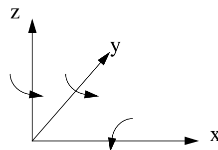
och vid skalning

$$\mathbf{P'} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

3D-Transformationer: Rotation 2(3)

Förut roterade vi kring en punkt. I tre dimensioner blir det fråga om rotation kring en axel. 2D-fallet kan uppfattas som rotation kring z-xeln. Vi ser först på rotation kring de tre koordinataxlarna.

Positiv riktning (pilens riktning):
Tankesätt 1: Motsols när man tittar i negativ riktning längs axeln.
Tankesätt 2: Som när man vrider en högergängad skruv längs axeln kortaste vägen från x till y (z-axeln), y till z (x-axeln) och z till x (y-axeln).



Rotation kring z (z ändras ej)

$$\mathbf{P'} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

Rotation kring x (x ändras ej; x->y, y->z i tidigare)

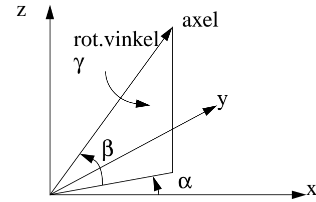
$$\mathbf{P'} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

Rotation kring y (y ändras ej; x->z, y->x jämfört med första)

$$\mathbf{P'} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

3D-Transformationer: Rotation kring godtycklig axel 3(3)

Förut roterade vi kring en punkt. I tre dimensioner blir det fråga om rotation kring en axel.



Man kan då göra ungefär som vid rotation i 2D kring godtycklig punkt. Vi translaterar först axeln (och världen) så att den går genom origo (det gör den redan i figuren). Därefter roterar vi axeln runt z-axeln -α så att den ligger i zx-planet. Därefter roterar vi axeln β runt y-axeln så att den sammanfaller med x-axeln. Varpå vi gör den önskade rotationen γ runt x-axeln. Vi återställer genom göra de tidigare transformationerna i omvänd ordning och med negerade parametrar. Transformationen kan därför uttryckas (beteckningarna är förhoppningsvis begripliga utan förklaring).

$\mathbf{P'} = T(\text{-ngt})R_z(\alpha)R_y(-\beta)R_x(\gamma)R_y(\beta)R_z(-\alpha)T(\text{ngt})\mathbf{P}$

I t ex Hills bok (sid 241) hittar man en formel som uttrycker transformationen direkt i de givna storheterna: rotationsaxeln och vinkeln γ. Men vi behöver inte den.

OpenGL och 3D-grafik

Vi har nu sett att de intressanta transformationerna kan representeras med matriser. Närmast skall vi titta på hur övergången från värld till skärm går till, vilket beskrivs i småskriften “Från värld till skärm”, avsnitten 1-4. Vi kommer att finna att även denna kan göras med matrisspråk.

Vi är sedan redo att se på hur vi kan bygga upp 3D-scener i OpenGL. Detta avhandlas i avsnitten 7-8 och 10 i OpenGL-häftet. Vi tar också upp begreppet scengraf.

Sedan tar vi upp avsnitten 5, 6 och 7 i “Från värld till skärm” (avsnitt 10 blir inte aktuellt förrän vi kommer in på texturer, avsnitten 8-9, 11 behandlas vid senare tillfälle).

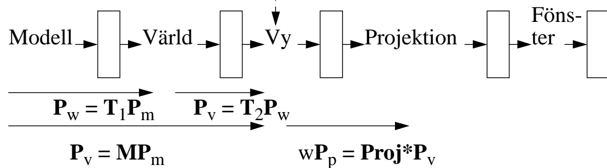
Efter det fortsätter vi med OH-bilder om lösning av dolda yt-problemet.

Men låt oss redan nu ge en kort introduktion till transformationer i OpenGL. Säg att vi gjort en procedur `House()` som ritar ett hus. Vi kan då lätt rita ett hus till med (`d` i st f `f` går lika bra!)

```
glTranslatef(tx,ty,tz);
glRotatef(vinkel i grader, 0,1,0);
glScalef(sx,sy,sz);
House();
```

Det nya huset skalas först, roteras sedan och translateras till sist! Mer om det här i häftet. `glRotatef` gör en rotation kring en godtycklig axel (som går genom origo) och vars vektorkomponenter anges som de tre sista parametrarna (i exemplet y-axeln).

OpenGL och 3D-grafik: Rörledningen



där (att w har två olika betydelser är olyckligt, men ...).

$$\mathbf{P}_m = \begin{bmatrix} x_m \\ y_m \\ z_m \\ 1 \end{bmatrix}$$

och motsvarande för övriga. **M** är modellvy-matrisen och **Proj** projektionsmatrisen.

Transformationerna görs numera helt av grafikprocessorn. Transformationsstegen är t o m programmerbara fr o m de under 2001 introducerade grafikprocessorerna NVidia:s GeForce 3 (variant ingår i Xbox) och ATI:s Radeon 8500 (men ej i GeForce 4MX, som är en GeForce 2!). Mer om det i slutet av kursen.

I OpenGL-häftet (sid 10) står $\mathbf{P}' = \mathbf{Proj}*\mathbf{V}*\mathbf{Mod}*\mathbf{P}$ med $\mathbf{P}=[x\ y\ z\ 1]^T$. och tänkt $\mathbf{P}' = [wx', wy', wz', w]^T$. Det blir mycket tydligare om man som vi gjort här skriver $w\mathbf{P}_p = \mathbf{Proj}*\mathbf{V}*\mathbf{Mod}*\mathbf{P}_m$ med $\mathbf{P}_m$ etc enligt ovan.

OpenGL och 3D-grafik: Matrisoperationer 1(2)

Normalt bygger vi upp modellvy-matrisen med
`glTranslatef`, `glScalef`, `glRotatef` och `gluLookAt`.
Dessa skapar motsvarande transformationsmatris T och **multiplicerar modellvy-matrisen från höger** med denna, dvs
Mod = Mod*T.
Detta innebär att om man skriver

```
glTranslatef(...);
glRotatef(...);
```

kommer rotationen att utföras först och translationen sist. **Transformationerna utförs alltså i omvänd ordning mot den i programkoden**.

Projektionsmatrisen **Proj** konstrueras normalt på motsvarande sätt med
`glOrtho` respektive `gluPerspective`.

I båda fallen behöver man göra en initiering med
`glLoadIdentity()`
Vilken matris man arbetar bestäms av ett tillstånd som sätts med
`glMatrixMode(GL_PROJECTION)`
respektive
`glMatrixMode(GL_MODELVIEW)`

OpenGL och 3D-grafik: Matrisoperationer 2(2)

Man märker således inte så mycket av de bakomliggande matriserna. Men det finns några situationer när man behöver vara dem litet närmare.

`glPushMatrix();`
`glPopMatrix();`
Sparar respektive hämtar aktuell matris på/från en stack. Är ofta bättre än att bygga upp matrisen på nytt. Om vi exempelvis i `display` i `Matriskoll.c` (en kommande OH) hade skrivit

```
   glPushMatrix();
   glRotated(45,0,0,1);
   glPopMatrix();
```

hade vi undvikit den förmodligen oavsedda ackumuleringen.

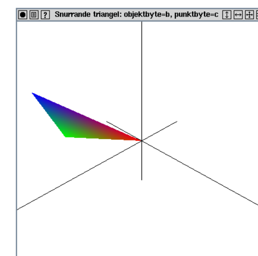
`glGetFloatv(vilken_matris,v);`
Läser av angiven matris (kolumn efter kolumn) till variabeln
`GLfloat v[16]`.
Första parametern kan vara `GL_MODELVIEW_MATRIX` eller `GL_PROJECTION_MATRIX` eller `GL_TEXTURE_MATRIX` (ev senare).

`glLoadMatrixf(v);`
Ersätter aktuell matris med innehållet i `GLfloat v[16]`.

`glMultMatrixf(v);`
Multiplicerar den aktuella matrisen **från höger** med matrisen motsvarande `GLfloat v[16]`.

Ett 3D-exempel 1(3)

Ett objekt i form av en triangel snurrar kring y-axeln. Man kan med tangent c byta till annat objekt och med tangent b byta position för observatören. Höger mustangent ger en liten meny. Musen måste i samtliga fall vara i fönstret.



```
> cat GL_3D_2003.c   (litet bearbetad)

// De vanliga include
...
#include <GL/glut.h>

// Globala
GLdouble vinkel = 0.0;
int POS1 = 1, OBJ1 = 1;

void koordinataxlar(GLdouble L) {
    glColor3f(0,0,0); // Svart
    glBegin(GL_LINES);
        glVertex3f(-L,0,0); glVertex3f(L,0,0);    //x-axel
        glVertex3f(0, -L,0); glVertex3f(0, L,0); //y-axel
        glVertex3f(0, 0,-L); glVertex3f(0, 0,L); //z-axel
    glEnd();
}
```

Ett 3D-exempel 2(3)

```
void rita(void) {
    glClearColor(1.0,1.0,1.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // Kan placeras i reshape om observatören orörlig
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    if (POS1) gluLookAt(0,0,2, 0,0,0, 0,1,0);
    else gluLookAt(1,1,1, 0,0,0, 0,1,0);

    koordinataxlar(2.0);
    glRotated(vinkel,0,1,0);
    if (OBJ1) {
      glBegin(GL_POLYGON);
         glColor3f(1,0,0); glVertex3f(0,0,0);
         glColor3f(0,1,0); glVertex3f(2,0,0);
         glColor3f(0,0,1); glVertex3f(2,1,0);
      glEnd();
    } else {
      glColor3f(1,0.5,0);
      glutSolidTeapot(1.0);
    }
    glutSwapBuffers();
}

void storleksbyte(int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(120,1,0.1,10);
}

void tangenthanterare(unsigned char key, int x, int y) {
   if (key == 'c') POS1 = !POS1;
   if (key == 'b') OBJ1 = !OBJ1;
   if (key == 'q') exit(0);
}
```


Ett 3D-exempel 3(3)

```
void myMenu(int item) {
   switch(item) {
       case 9: exit(0); break;
   }
}

void inget_att_gora() {
    vinkel = vinkel + 1;
    glutPostRedisplay();
}


int main(int argc, char** argv) {
    // Dubbla bildminnen vid all rörelse
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(400,400);
    glutCreateWindow(
           "Snurrande triangel: objektbyte=b, punktbyte=c");
    glutReshapeFunc (storleksbyte);
    glutDisplayFunc(rita);
    glutIdleFunc(inget_att_gora);
    glutKeyboardFunc(tangenthanterare);
    glutCreateMenu(myMenu);
        glutAddMenuEntry("Avsluta",9);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
    glutMainLoop();
    return 0;
}
```


Interaktion 1(2), bl a OpenGL-häftet avsnitt 16, 18

Aktivering av händelsehanterare

Mus:

```
glutMouseFunc(mushanterare);
glutMotionFunc(musrörelsehanterare);
```

Med nedtryckt knapp

```
glutPassiveMotionFunc(musrörelsehanterare);
```

Utan nedtryckt knapp

Tangenter:

```
glutKeyboardFunc(tangenthanterare);
glutSpecialFunc(stangenthanterare);
```

För piltangenter, funktionstangenter, ...

Meny (pop-up):

```
glutCreateMenu(menyhanterare);
   glutAddMenuEntry("menytext",7);
   glutAddMenuEntry("menytext",'d');
   glutAddMenuEntry("menytext",99);
   ...
glutAttachMenu(GLUT_RIGHT_BUTTON); // t ex
```

GLUT har stöd för andra interaktionsenheter.

Interaktion 2(2), bl a OpenGL-häftet avsnitt 16, 18

Hanterarna (C: **if** eller **switch** med **case**)

```
void mushanterare(int button, int state, int x, int y) {
     // button kan ha värdena GLUT_LEFT_BUTTON,
     //             GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON
     // state kan ha värdena GLUT_DOWN, GLUT_UP
     ...
     glutPostRedisplay(); // Om omritning önskas
}

void tangenthanterare(unsigned char key, int x, int y) {
        switch (key) {
              case 'd': .......; break;
              case 27 : exit(0); break;  // ESC ofta med
              ...
         }
         glutPostRedisplay(); // Om omritning önskas
}

void menyhanterare(int nr) {
        switch (nr) {
               case 7:  ...; break;
               case 99: exit(0);
               case 'd': tangenthanterare(nr, 0,0); break;
               ...
        }
        glutPostRedisplay(); // Om omritning önskas
}
```

Hanterarna för rörlig mus är litet annorlunda liksom hanteraren för specialtangenter. Man kan som synes kombinera tangent/meny.

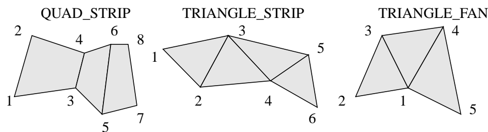
Geometriska objekt 1(2)

Geometribeskrivningarna görs genom att vi buntar ihop ett antal anrop av *glVertex*

```
glBegin(HUR_PUNKTERNA_SKALL_TOLKAS)
    glVertex3f(...); .....glVertex3f();
glEnd();
```

I det inledande exemplet använde vi GL_POLYGON som parameter till *glBegin* och den konstanten gör att punkterna tolkas som hörnen i en polygon och att polygonen (som standard; kan ställas om) ritas fylld med aktuell färg. Det finns flera andra värden, men vi tar inte upp alla.

GL_POINTS	En punkt ritas vid var och en av de angivna punkterna
GL_LINES	En linje ritas från P1 till P2, en annan från P3 till P4, o s v.
GL_TRIANGLES	En följd av (fyllda) trianglar ritas baserad på de tre första punkterna, de tre följande, o s v.
GL_QUADS	Motsvarande men fyrhörningar
GL_TRIANGLE_STRIP	P1, P2 och P3 ger en triangel. P3, P2, P4 nästa. P3, P4, P5 den tredje. O s v. Se figur nedan.
GL_TRIANGLE_FAN	P1, P2 och P3. Sedan P1, P3 och P4. Sedan P1, P4 och P5. O s v. Se figur nedan.
GL_QUAD_STRIP	Se figur nedan.



Geometriska objekt 2(2)

GLUT och GLU innehåller några färdiga modeller (kuber, koner, sfärer m m). Vi nöjer oss här med att ta upp några från GLUT (bygger ofta på motsvarande i GLU).

Kub (tråd- resp solidform) med sidan *size* och med mittpunkt i origo. Med *glScalef* kan vi lätt ordna allmänna rätblock.

```
void glutWireCube(GLdouble size);
void glutSolidCube(GLdouble size);
```

Sfär (och därmed ellipsoid med *glScalef*) med mittpunkt i origo.

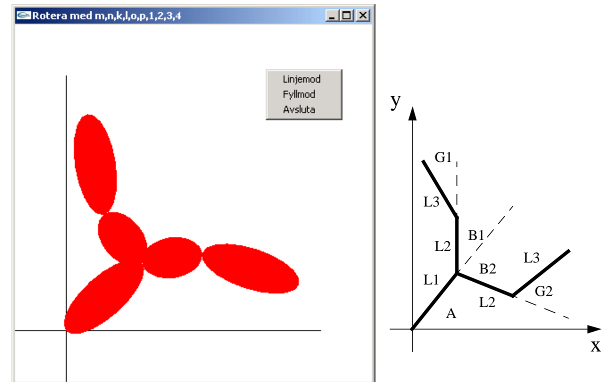
```
void glutWireSphere(GLdouble radius,
                    GLint slices, GLint stacks);
void glutSolidSphere(GLdouble radius,
                    GLint slices, GLint stacks);
```

Andra är (vi tar bara med solidformerna; mittpunkten är normalt origo)

```
void glutSolidCone(GLdouble base, GLdouble height,
         GLint slices, GLint stacks);
void glutSolidTorus(GLdouble innerRadius,
         GLdouble outerRadius, GLint sides, GLint rings);
void glutSolidDodecahedron(void); // Dodecahedron, 20 hörn
void glutSolidTeapot(GLdouble size); // Klassisk modell
void glutSolidOctahedron(void);   // Oktahedron, diamant
void glutSolidTetrahedron(void);  // Tetraeder
void glutSolidIcosahedron(void);  // Icosahedron, 12 hörn
```


Ett sammansatt objekt 1(3)

Först Exempel 6 från OpenGL-häftet. Och sedan detta allmännare exempel med en flerarmad robot (eller ett harhuvud), där varje led är fritt roterbar kring sin fästpunkt!



`> cat GL_ROT_2003.c` (bearbetad; kan mer än vad som framgår)

```
// Vanliga include
#include <GL/glut.h>

// Globala vinklar och längder enligt högra figuren
GLdouble A=0.0, B1=0.0, B2=0.0, G1=0.0, G2=0.0, X=0.0;
GLdouble L1=1.0, L2=0.6, L3=1.0;
```



```
void koordinataxlar() {
    glBegin(GL_LINES);
       glVertex3f(-0.5,0,0); glVertex3f(2.5,0,0); //x-axel
       glVertex3f(0, -0.5,0); glVertex3f(0, 2.5,0); //y-axel
    glEnd();
}

void enDel(GLdouble L) {
   GLdouble D = 0.2;
   // Ellipsoid 0<x<L, -D<y<D
   glPushMatrix();
   //Om ej ackumuleras transformationerna av upprepade anrop
   glTranslated(L/2,0,0);//=>ellipsoid med ena änden i origo
   glScaled(L/2,D,1);  //=>ellipsoid i origo med storaxeln L
   glutSolidSphere(1.0,20,20);
   glPopMatrix();
}

void rita(void) {
   glClearColor(1.0,1.0,1.0,0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glColor3f(0.0,0.0,0.0);
   koordinataxlar();
   glColor3f(1.0,0.0,0.0);
   glRotated(A,0,0,1);
   enDel(L1);
   glPushMatrix();
       glTranslated(L1,0,0); glRotated(B1,0,0,1); enDel(L2);
       glTranslated(L2,0,0); glRotated(G1,0,0,1); enDel(L3);
   glPopMatrix();
   glPushMatrix();
       glTranslated(L1,0,0); glRotated(B2,0,0,1); enDel(L2);
       glTranslated(L2,0,0); glRotated(G2,0,0,1); enDel(L3);
   glPopMatrix();
   glutSwapBuffers();
}
```

```
void storleksbyte(int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-0.5,3.0,-0.5,3.0,-1.0,1.0);
}

void tangenthanterare(unsigned char key, int x, int y) {
   if (key == ’m’) A = A + 3.0;
   if (key == ’n’) A = A - 3.0;
   if (key == ’k’) B1 = B1 - 3.0;
   if (key == ’l’) B1 = B1 + 3.0;
   ...
   if (key == ’q’) exit(0);
   glutPostRedisplay();
}

void myMenu(int item) {
   switch(item) {
       case 1: tangenthanterare(’L’,0,0); break;
       case 2: tangenthanterare(’F’,0,0); break;
       case 9: exit(0); break;
   }
}

int main(int argc, char** argv)  {
    // Dubbla bildminnen vid all rörelse
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(400,400);
    glutCreateWindow("Rotera med m,n,k,l,o,p,1,2,3,4");
    printf("Växla ritsätt med F, L och P\n");
    glutReshapeFunc (storleksbyte);
    glutDisplayFunc(rita);
    glutKeyboardFunc(tangenthanterare);
    glutCreateMenu(myMenu);
        glutAddMenuEntry("Avsluta",9);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
    glutMainLoop();
}
```


Ett mellanspel - polygonmoder

Normalt ritas ytor fyllda. Men med anrop av typen

`glPolygonMode(GL_FRONT_AND_BACK, ritsätt);`

där ritsätt är `GL_POINT` (hörnpunkterna), `GL_LINE` (kanterna) eller `GL_FILL` (standardbeteendet) kan vi få ritning av enbart hörnpunkterna, enbart kanterna respektive fyllnad. Och det går att kombinera (vilket för rätt effekt kräver vissa extra åtgärder).

I förra programmet finns i (den fullständiga) tangenthanteraren:

```
   if (key == ’F’)
         glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
   if (key == ’L’)
         glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
   if (key == ’P’) {
         glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
         glPointSize(4.0);
   }
```

I menyuppbyggnaden finns

```
glutAddMenuEntry("Linjemod",1);
glutAddMenuEntry("Fyllmod",2);
```

varför vi i menyhanteraren kan ta till

```
void myMenu(int item) {
   switch(item) {
       case 1: tangenthanterare(’L’,0,0); break;
       case 2: tangenthanterare(’F’,0,0); break;
       case 9: exit(0); break;
   }
}
```

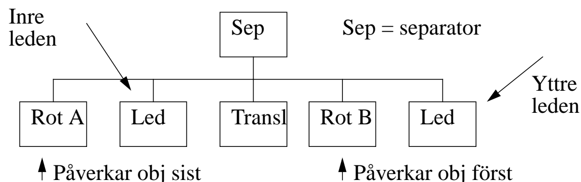

Scengrafer 1(3)

Scengrafer har huvudsakligen följande användningar:

- **Strukturering av bl a hierarkiska modeller**
- **Strukturering av modeller för optimering och selektering**

En scengraf är ett allmänt träd (eller en graf om vi återutnyttjar vissa objekt) med alla scenens objekt. Till objekten räknas de geometriska objekten men även transformationer och material. Och mycket annat som vi i varje fall inte just nu går in på. Man skapar objekten och lägger successivt in dem i grafen (objekten kallas därefter noder) utan att rita något. Ritning görs i ett separat steg utifrån scengrafen. Man kan när som helst ändra en egenskap hos ett objekt/nod. Om vi t ex ändrar en vinkel för en rotationsnod kan vi efter ny uppritning få rörelse.

Vi belyser nu scengrafer i anslutning till förra exemplet. Först bara två leder. Objektet Led är i det allmänna fallet i sig ett sammansatt objekt.



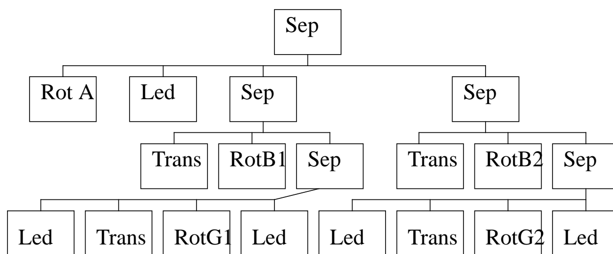
Transformationer ackumuleras precis som i OpenGL. Tillstånd ersätter tidigare gällande tillstånd. Scengrafen genomlöpes enligt “djupet först“ och från vänster till höger. Inplaceringsordningen är således betydelsefull.

Scengrafer 2(3)

OpenGL har inte scengrafer men vi kan simulera dem på låg nivå (med hjälp av bl a *glPushMatrix*/*glPopMatrix*) eller på högre nivå genom att införa egna verktyg (träd eller grafer och listor). Open Inventor och Java 3D har scengrafer och hjälpmedel för dem inbyggda. VRML (Virtual Reality Modelling Language) och dess efterföljare har scengrafer som en ingrediens.

Separationsnoder (Sep i figurerna) gör att alla förändringar under noden förblir lokala (kodas i OpenGL med *glPushMatrix*/*glPopMatrix*). Tillstånds- och transformationsändringar till vänster om och på samma nivå som separationsnoden ärvs. **Gruppnoder** - som vi inte exemplifierat - innebär att det som görs under noden påverkar till höger om noden.

Till slut en scengraf för fallet med 5 leder fördelade enligt tidigare figur:



Scengrafer 3(3)

Open Inventor är en produkt som ursprungligen togs fram av Silicon Graphics. SGI släppte för ett par år sedan källkoden vilket lett till att det finns ett par fria versioner. Se kursens webbsida för adresser.

Linux: En av dessa (fr o m ht 2005 FLTK:s) har jag fått till att fungera tillsammans med g++ under Linux. Används så här. Säg att källkodsfilen heter `test.c++`. Då kan man kompilera och länka med (som vanligt har jag gjort en kommandoprocedur, som den intresserade kan inspektera)
`INV_COMPILE test`
varvid det bildas en **stor** körbar fil test, som enklast körs med
`INV_RUN test`
I mappen `/users/course/TDA360/LINUX/FLINV` finns ett par exempel. I `/users/course/TDA360/LINUX/FLINV/fl-inventor/install/bin/unix` många fler.
Windows: En av mig installerad version går att använda direkt ihop med MSVC++. Man måste se till `Z:\DGKURSEN\PC\INVPC\VC98` räknas som inkluderingsmapp och att `Z:\DGKURSEN\PC\glut32.lib` samt `Z:\DGKURSEN\PC\INVPC\inventor.lib` länkas med. Vidare måste `inventor.dll` nås via PATH (t ex i aktuell mapp).

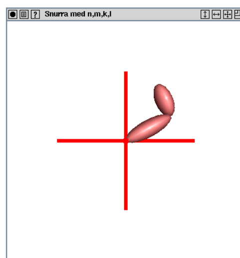
Open Inventor utvecklas kommersiellt av företaget TGS (nu uppköpt av Mercury).

Andra scengraf-system ovanpå OpenGL, som finns (i princip) för åtminstone Windows och Linux (se webbsidan för adresser):

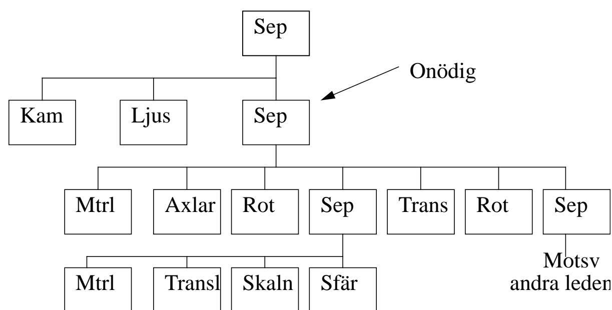
- **OpenScenegraph**
- **NVIDIA SG**
- **Gizmo (från Saab Training Systems AB)**

Open Inventor 1(6)

Vi belyser nu scengrafer med hjälp av Open Inventor. Materialet här enbart med för illustrationens skull. Tyvärr blir även det enklaste program rätt skrymmande. Men jag hoppas att när nu de flesta läst ett objektorienterat språk som Java, skall helhetsideerna framgå trots de många detaljerna. Vi nöjer oss med fallet med två leder. Vi behöver då



bygga en scengraf med följande struktur (vissa extra finesser):



Open Inventor 2(6)

Kommentarerna till programkoden får bli huvudsakligen muntliga.

> cat INV_2003.c++

```
// Först måste man inkludera filer för varje klass
#include <Inventor/SoDB.h>
#include <Inventor/SoSceneManager.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/actions/SoWriteAction.h>
...
#include <GL/glut.h>
#include <iostream>
using namespace std;

// Globala objekt m m
SoSceneManager *scenemanager;
SoSeparator *root ;
SoPerspectiveCamera *perspCamera;
SoRotationXYZ *rotation = new SoRotationXYZ;
SoRotationXYZ *rotationB = new SoRotationXYZ;
double A=3.0, B=1.5; //vinklar vid start
double Z = 16.0;

// Redraw on scenegraph changes.
void redraw(void * user, SoSceneManager * manager) {
     glEnable(GL_DEPTH_TEST); glEnable(GL_LIGHTING);
     manager->render();
     glutSwapBuffers();
}
```


Open Inventor 3(6)

```
// Vanliga omritaren
void rita(void) {
     glEnable(GL_DEPTH_TEST);  glEnable(GL_LIGHTING);
     scenemanager->render();
     glutSwapBuffers();
}

void storleksbyte(int w, int h) {
    scenemanager->setWindowSize(SbVec2s(w, h));
    scenemanager->setSize(SbVec2s(w, h));
    scenemanager->setViewportRegion(SbViewportRegion(w, h));
    scenemanager->scheduleRedraw();
}

// Måste vara med?
void idle(void) {
     SoDB::getSensorManager()->processTimerQueue();
     SoDB::getSensorManager()->processDelayQueue(TRUE);
}

SoSeparator* enLed(double length) {
    SoSeparator *led = new SoSeparator;
    SoTransform *translation = new SoTransform;
    translation ->translation.setValue(length/2, 0, 0);
    SoTransform *skalning = new SoTransform;
    skalning ->scaleFactor.setValue(length/2, 0.5, 1);
    led->addChild(translation);
    led->addChild(skalning);
    SoMaterial *mtrl = new SoMaterial;
    mtrl->ambientColor.setValue(1.0, .2, .2);
    mtrl->diffuseColor.setValue(1.0, .6, .6);
    mtrl->specularColor.setValue(1.0, .5, .5);
    mtrl->shininess = .5;
    led->addChild(mtrl);
    SoSphere *s = new SoSphere;
    led->addChild(s);
    return led;
}
```

Open Inventor 4(6)

```
SoCube* axel(double xL, double yL, double zL) {
        SoCube *x = new SoCube;
        x->width = xL; x->height = yL; x->depth = zL;
        return x;
}

SoSeparator* roboten() {
    // En robot med två leder och koordinataxlar
    SoMaterial *axelmtrl = new SoMaterial;
    axelmtrl->ambientColor.setValue(1.0, 0, 0.0);
    axelmtrl->diffuseColor.setValue(1.0, 0.0, 0.0);
    axelmtrl->specularColor.setValue(1.0, 0.0, 0.0);
    axelmtrl->shininess = .5;

    SoSeparator *robot = new SoSeparator;
    rotation->axis = SoRotationXYZ::Z;
    rotation->angle = A;
    robot->addChild(axelmtrl);
    robot->addChild(axel(8.0, 0.2, 0.2));
    robot->addChild(axel(0.2,8.0,0.2));
    robot->addChild(axel(0.2,0.2,8.0));
    robot->addChild(rotation);
    robot->addChild(enLed(3.0));
    SoTransform *translationB = new SoTransform;
    translationB->translation.setValue(3, 0, 0);
    robot->addChild(translationB);
    rotationB->axis = SoRotationXYZ::Z;
    rotationB->angle = B;
    robot->addChild(rotationB);
    robot->addChild(enLed(2.0));
    return robot;
}
```


Open Inventor 5(6)

```
void tangentbord(unsigned char key, int x, int y) {
        SoWriteAction writeAction;
        switch(key) {
            case ’m’: // Radianer (grader i OpenGL)
                      A=A+0.03; rotation->angle = A; break;
            case ’n’:A=A-0.03; rotation->angle = A; break;
            case ’k’:B=B+0.03; rotationB->angle = B; break;
            case ’l’:B=B-0.03; rotationB->angle = B; break;
            case ’p’: writeAction.apply(root); break;
            case ’+’: Z=Z+1;
                      perspCamera->position.setValue(0,0,Z);
                      break;
            case ’-’: Z=Z-1;
                      perspCamera->position.setValue(0,0,Z);
                      break;
            case 27: exit(0); // ESC
            default: break ;
        }
        scenemanager->scheduleRedraw();
}

void menu(int value) {
  // Vi samlar allt i tangentbord i stället
  tangentbord((unsigned char) value, 0, 0);
}

void initMenu() {
   glutCreateMenu(menu);
      glutAddMenuEntry("Skriv ut scengrafen",’p’);
      glutAddMenuEntry("Avsluta", 27);
   glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```


Open Inventor 6(6)

```
int main(int argc, char ** argv) {
  SoDB::init();
  printf("Open Inventor version: %s\n",SoDB::getVersion()) ;
  // Scengrafens rotnod
  root = new SoSeparator; root->ref();
  // En kamera
  perspCamera = new SoPerspectiveCamera;
  root->addChild(perspCamera);
  // Ljus
  root->addChild(new SoDirectionalLight);
  // Roboten
  SoSeparator *temp = roboten();
  if (temp) root->addChild(temp) ;
  // Initiera en scenhanterare
  scenemanager = new SoSceneManager;
  scenemanager->setRenderCallback(redraw, NULL);
  scenemanager->setBackgroundColor(SbColor(1.0, 1.0, 1.0));
  scenemanager->activate();
  scenemanager->setSceneGraph(root);
  // Vi vill se hela scenen
  perspCamera->viewAll(root,
               scenemanager->getViewportRegion());
  // GLUT
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowSize(400, 400);
  glutCreateWindow("Snurra med n,m,k,l");
  glutDisplayFunc(rita);
  glutReshapeFunc(storleksbyte);
  glutIdleFunc(idle); //Verkar nödvändig
  glutKeyboardFunc(tangentbord); initMenu();
  glutMainLoop();
  // Snarare på annan plats
  root->unref();
  delete scenemanager;
  return 0;
}
```


Open Inventor - Scengraf utskriven 1(2)

Scengrafen kan skrivas ut enkelt (och även läsas in). Värden skrivs inte ut om de är standard.

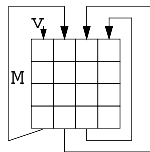
```
#Inventor V2.1 ascii
Separator {
    PerspectiveCamera {
        position        0 0 16.7262
        nearDistance    9.79796
        farDistance     23.6544
        focalDistance   16.7262
    }
    DirectionalLight {
    }
    Separator {
       # Här kommer koordinataxlarna; jag hoppar över en del
        Material {
            ambientColor        1 0 0
            ...
        }
        Cube {
            width       8
            height      0.2
            depth       0.2
        }
        Cube {...}
        Cube {...}
        RotationXYZ {
            axis        Z
            angle       6.78
        }
```

Open Inventor - Scengraf utskriven 2(2)

```
        # Här kommer ellipsoiden
        Separator {
            Transform {
                translation     1.5 0 0
            }
            Transform {
                scaleFactor     1.5 0.5 1
            }
            Material {
                ambientColor    1 0.2 0.2
                diffuseColor    1 0.6 0.6
                specularColor   1 0.5 0.5
                shininess       0.5
            }
            Sphere {
            }
        }
        Transform {
            translation 3 0 0
        }
        RotationXYZ {
            axis        Z
            angle       1.5
        }
        # Den andra ellipsoiden
        Separator {...}
    }
}
```


OpenGL och 3D-grafik: Avläs matriserna 1(2) (se även “Från värld till skärm”, avsnitt 5)

Man kan avläsa modellvy- och projektionsmatriserna!




```
> cat Matriskoll.c
// Diverse include
void CheckMatrix(GLenum vilken_matris) {
    GLfloat v[16]; // double duger även
    int i;
    glGetFloatv(vilken_matris,v);
    // glGetDoublev finns också

    if (vilken_matris==GL_PROJECTION_MATRIX) {
       printf("Utskrift av projektionsmatrisen\n");
    } else printf("Utskrift av modellvy-matrisen\n");
    for (i=0; i<4; i++) {
       printf("%f %f %f %f\n",v[i], v[i+4], v[i+8], v[i+12]);
    }
}
void myReshape(int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    glOrtho(0,width,0,width,-1,1);
    glMatrixMode(GL_MODELVIEW);glLoadIdentity();
}
void display(void) {
    glRotated(45,0,0,1); // Ackumuleras!!!!!!!!!!!!!!!!
    CheckMatrix(GL_MODELVIEW_MATRIX);
    CheckMatrix(GL_PROJECTION_MATRIX);
}
int main(int argc, char** argv) {
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(400,400);
    glutCreateWindow("Matriskontroll");
    glutReshapeFunc (myReshape); glutDisplayFunc(display);
    glutMainLoop();
}
```


OpenGL och 3D-grafik: Avläs matriserna 2(2)

Kompilering och körning på PC.

```
>gcc -o Matriskoll.exe Matriskoll.c -lglut32 -lglu32
     -lopengl32 -luser32 -lgdi32
>Matriskoll
Uppdatering                         // första gången
Utskrift av modellvy-matrisen
0.707107 -0.707107 0.000000 0.000000     // 45°
0.707107 0.707107 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
Utskrift av projektionsmatrisen
0.005000 0.000000 0.000000 -1.000000     // Ortografisk
0.000000 0.005000 0.000000 -1.000000
0.000000 0.000000 -1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
Uppdatering                         // framtvingad andra gång
Utskrift av modellvy-matrisen
0.000000 -1.000000 0.000000 0.000000     // Nu 90°
1.000000 0.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
Utskrift av projektionsmatrisen
0.005000 0.000000 0.000000 -1.000000     // Oförändrad
0.000000 0.005000 0.000000 -1.000000
0.000000 0.000000 -1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
```

Stämmer med teorin!

Animering (OpenGL-häftet avsnitt 9)

Vi vill visa upp en scen - förmodligen successivt förändrad - gång på gång.

- **Dubbelbuffra**, dvs använd båda bildminnena.
  Rita i det som inte visas. Begärs enligt modellen `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);`
  Växla bildminne i slutet av omritningsproceduren med `glutSwapBuffers();`
  JOGL: Automatiskt.
- Låt **all ritning ske i omritningsproceduren**
  Anropa den aldrig direkt utan skapa en omritningshändelse med `glutPostRedisplay();`
  JOGL: Automatiskt med den programvara vi utgår från.
- Gärna **tidsstyrd omritningshastighet**
  Se avsnitt 17.
  JOGL: Borde kunna styras med metod i klassen Animator, men jag hittar ingen sådan. Ett sätt är att lägga tidskontrollen i *display*.
- **Tangentbordet är sällan fullgott för styrning** på en dator med goda grafikprestanda
  P g a låg repetitionshastighet (som nog kan ändras) hos tangenterna. I stället kan man använda dem som enbart tillståndsändrare, t ex framåt <-> bakåt.

Hur styrningen av förändringarna skall gå till tar vi upp i samband med navigering efter att vi “löst” problemet med dolda ytor.

Dolda ytor 1(1)

Problemet: I 3D skyms vissa ytor eller linjer ofta av andra. Det betyder att utan särskilda åtgärder blir resultatet olika beroende på i vilken ordning man ritar ytorna (linjerna).

Vi skall titta litet på tre metoder:

- Djupbuffert (z-buffert), som är standardmetoden. Rastermetod.
- Målarns metod. Objektmetod.
- BSP. Objektmetod.

En rastermetod ger ett resultat med given upplösning. En objektmetod ger en bild vars kvalitet blir bättre ju bättre presentationsutrustningen är. Under färden kommer vi att lära oss en massa annat nyttigt, t ex hur man räknar ut normalen till en polygon.

1974 skrev Ivan Sutherland (f ö en av de första som sysslade med datorgrafik på 60-talet; “brilliant thesis” 1962 “Sketchpad,...”) m fl en översiktsartikel över olika sätt att lösa detta problem - kallat **dolda-yt-problemet** (eng. hidden surfaces eller hidden lines). Inte med ett ord antyddes den algoritm som används som standard på alla grafikkort idag! Inte heller BSP.

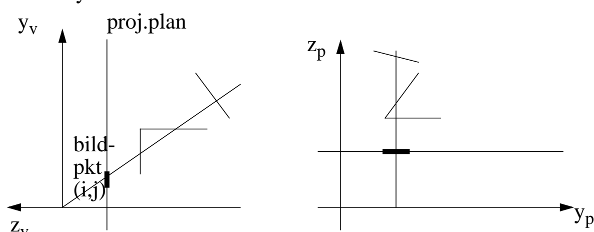
Förutsättningar: Vi tänker oss innehållet i världen uppbyggt av polygoner (kan vara enbart trianglar eller allmänna).

Dolda ytor: Djupbuffert 1(2)

Metoden bygger på att man har ett djupminne med ett element motsvarande varje bildpunkt. Se tidigare OH. Djupminnet innehåller “avstånd” till det närmsta av de objekt som ger upphov till bildpunkten.

Djupbuffertalgoritmen:
1. Initiera djupminnet till maxavstånd
2. För varje objekt
    2.1 För varje polygon i objektet
        2.1.1 För varje bildpunkt (i,j) som genereras av polygonen
            2.1.1.1 Undersök om djupet z(i,j) är mindre än det i djupminnet lagrade d[i,j]. I så fall rita bildpunkten och sätt d[i,j]=z(i,j).

Djupet kan vara $-z_v$ eller z_p (eller som i OpenGL $0.5*(z_p + 1)$, dvs $0 \le djup \le 1$). I båda fallen växer z(i,j) med avståndet till skärningspunkten. Man skulle också kunna använda verkligt avstånd som djup, men det blir dyrt.



Dolda ytor: Djupbuffert 2(2), forts

Hur beräknas djupet? För hörnpunkterna beräknas det i samband med hörntransformationerna. I övriga punkter med linjär interpolation under rastreringen.

Djupminnet kan teoretiskt sett emuleras med en matris. Se exempel i avsnitt 7 i “Från värld till skärm”.

Brister:
1. Eftersom djupminnet har en begränsad upplösning kan punkter som ligger på olika avstånd ge upphov till samma djup i djupminnet med åtföljande problem. Om objekten (t ex en linje på en polygonyta) har samma avstånd kan det hända att objekten framträder omväxlande.
2. Alla polygoner ritas

Dolda ytor: Djupbuffert i OpenGL 1(2)

1. Begär resursen med
`glutInitDisplayMode(GLUT_RGB|GLUT_DEPTH ....)`

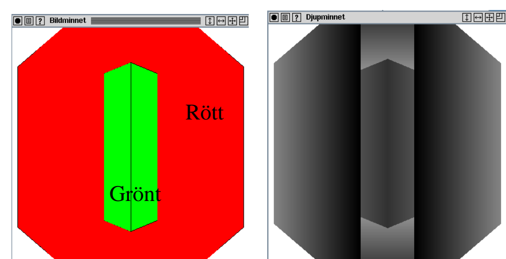
2. Slå på djupminnestestet med (skall göras efter det att man med `glutCreateWindow` skapat fönstret)
`glEnable(GL_DEPTH_TEST)`. Kan tillfälligt slås av med
`glDisable(GL_DEPTH_TEST)`.

3. Se till att inte bara bildminnet utan även djupminnet “suddas” i omritningsproceduren. Sker med

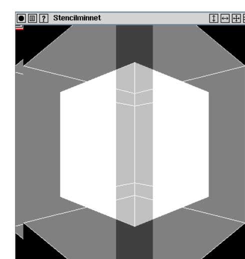
```
glClearColor(raderingsfärg);
glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);
```


Dolda ytor: Djupbuffert i OpenGL 2(2)

Djupminnet kan visualiseras (programmet GL_BUFFERTAR.c) genom att man “rasterkopierar” från djupminnet till bildminnet. Detaljerna finns i OGL-häftet, avsnitt 15. Till vänster ser vi en röd kub. Vi

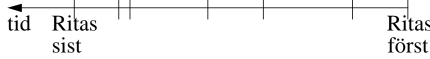


har gått så nära att man ser att det finns en grön kub inuti. Det är klippningen vid hitre klipp-planet som gör att den yttre kuben “öppnar” sig. Till höger har vi med gråskala avbildat djupminnet. Punkter längst bort har ju djupet 1.0, dvs de blir vita. Punkter närmare oss har mindre djup och blir följaktligen mörkare. Motsvarande bild nedan av stencilbufferten kommenteras senare.



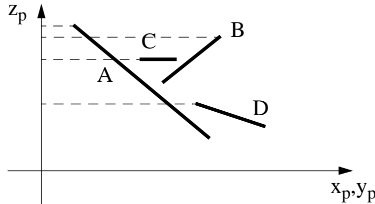
Dolda ytor: Målarns metod 1(2)

Idé: Måla bakgrunden först och sedan successivt de objekt som ligger närmare observatören. Man ordnar sidoytorna (polygonerna) i en lista så att



ger rätt resultat, dvs en tidigt ritad polygon får inte skymma en senare ritad.

Exempel: Vi har ett antal sidoytor som är vinkelräta mot zx-planet och bilden visar hur det ser ut uppifrån.

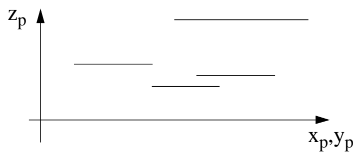


Vi inser att om vi låter listan vara ADBC blir uppritningen korrekt.

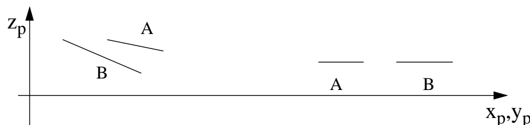
Ett första försök: Sortera efter växande $z_{p,max}$ (dvs hörnens största z_p-värde). Skulle i vårt exempel ge listan DCBA och uppritningsresultatet blir fel. En annan variant är att sortera efter $z_{p,medel}$. Men inte heller det sättet ger i allmänhet rätt resultat. Dessa enkla metoder brukar dock fungera bra för t ex funktionsytor (approximerade med

Dolda ytor: Målarns metod 2(2)

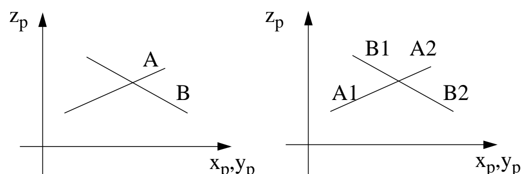
polygoner) och för s k 2 1/2-D-scener (objekten har inget djup och är vinkelräta mot betraktningsriktningen). MATLAB använde länge denna metod för uppritning av funktionsytor.



Generell algoritm: Omständlig och bygger på upprepade omsorteringar för att se till att ingen tidig polygon skymmer en senare. Innehåller många intressanta delproblem, t ex hur avgör man i figuren att ytan A inte skymmer B?



Det finns situationer då det inte går att ordna sidorna, t ex:



Då får man ta till uppdelning (klipp vardera planet mot det andra).

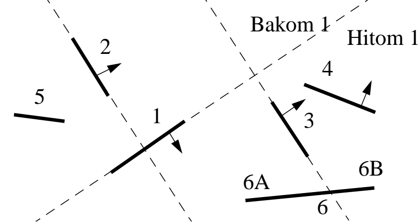
Dolda ytor: BSP 1(2)

Målarns metod innebär att sorteringen måste göras om för varje ny betraktningssituation. Finns det något sätt att undvika det? Svaret är en aning överraskande: Ja, om scenen är statisk, dvs inga objekt rör sig. Metoden som löser problemet kallas BSP (Binary Space Partitioning). Den presenterades kring 1980. Kan göras i världskoordinater.

Algoritmen består av två steg:

1. Polygonerna sorteras rekursivt i ett binärt träd. Detta steg kan göras vid programstarten eller utanför programmet.
2. Uppritningen

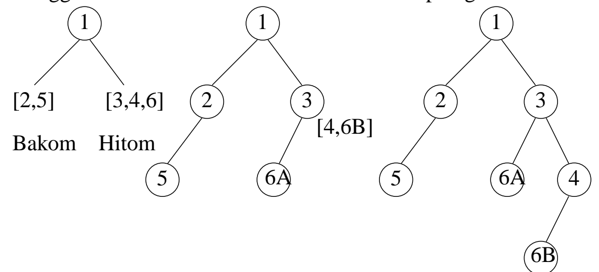
Låt oss först illustrera steg 1 med en artificiell exempelsituation,



Vi startar med en lista av polygoner [1,2,3,4,5,6]. Vi väljer en av dessa, säg 1, och låter motsvarande plan dela det tredimensionella rummet i två halvor: Bakom 1 resp Hitom 1. Vad som väljs som bakom resp framför spelar inte så stor roll, men vi skall om en stund ange ett deterministiskt sätt. Vi stoppar in polygon 1 överst i ett binärt träd. De övriga läggs i en bakomlista [2,5] resp en hitomlista [3,4,6].

Dolda ytor: BSP 2(2)

Vi tillämpar nu rekursivt samma resonemang på de två nya listorna och lägger motsvarande träd som vänsterträd resp högerträd.



Steg 1 mera formellt i pseudospråk:

```
Tree SkapaBSPT(PolygonLista L) {
      Om L tom returnera ett tomt träd;
      Annars: Välj en polygon P i listan.
          Bilda en lista B med de polygoner som ligger bakom
          P och en annan H med övriga. Returnera ett träd med
          P som rot och SkapaBSPT(B) och SkapaBSPT(H) som
          vänsterbarn respektive högerbarn.
}
```

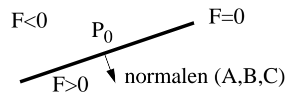
Uppritningsteget (kolla även om trädet tomt! Fick ej plats i koden):

```
void RitaBSP(Tree t) {
     Om observatören hitom roten i t:
         RitaBSP(t:s vänsterbarn);
         Rita polygonen i t:s rot;
         RitaBSP(t:s högerbarn);
     Annars:
         RitaBSP(t:s högerbarn); Rita polygonen i t:s rot;
         RitaBSP(t:s vänsterbarn);
}
```

Polygoner. Normaler. Hitom, bakom 1(2)

En plan polygonyta är ju en del av ett plan som kan skrivas på formen
$F(x,y,z) = Ax + By + Cz + D = 0$
Det är välbekant att (A,B,C) är en normal till polygonen, men låt oss ändå visa det. (x,y,z) får beteckna en godtycklig punkt på planet, medan $P_0 = (x_0,y_0,z_0)$ är en viss punkt. Då är
$Ax + By + Cz + D = 0$
$Ax_0 + By_0 + Cz_0 + D = 0$
dvs
$A(x-x_0) + B(y-y_0) + C(z-z_0) = 0$
Men det betyder ju att (A,B,C) är vinkelrät mot varje linje i planet, dvs är en normal.

Vi kan nu t ex låta
Bakom: De punkter (x,y,z) sådana att $F(x,y,z)<0$.
Hitom: De punkter (x,y,z) sådana att $F(x,y,z)>0$.

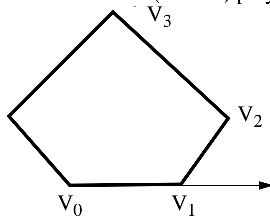


Vi inser att det förhåller sig som i figuren, ty om (x,y,z) ligger på den utritade normalens sida är skalärprodukten av $(x,y,z)-P_0$ och (A,B,C) större än noll.

En (konvex) polygon Q ligger därmed bakom en annan R om samtliga hörnpunkter (x,y,z) i Q uppfyller $F_R(x,y,z) < 0$, etc.

Polygoner. Normaler. Hitom, bakom 2(2)

Återstår att bestämma normalen till en (konvex) polygon



Ordna hörnen V_i motsols sett utifrån det objekt (polyeder) till vilket polygonen är en sidoyta. Det är uppenbart att kryssprodukten
$(V_1-V_0)x(V_2-V_1)$
är en normal som är riktad ut från papprets plan.
Med denna teknik kommer hitom att betyda att vi står på den sida om polygonen som gör att vi upplever hörnen som ordnade motsols.
Det finns ett annat sätt att beräkna normalen (A,B,C) som är pålitligare om polygonen inte är alldeles plan (se t ex Hill för fler detaljer) och gäller allmännare. Låt $V_i=(x_i,y_i,z_i)$, $0 \le i \le N-1$.

$$A = \sum_{i=0}^{N-1} (y_i - y_{i+1})(z_i + z_{i+1})$$

$$B = \sum_{i=0}^{N-1} (z_i - z_{i+1})(x_i + x_{i+1})$$

$$C = \sum_{i=0}^{N-1} (x_i - x_{i+1})(y_i + y_{i+1})$$

Dolda ytor: BSP. Variant 1(1)

BSP kan ses som en variant av målarns algoritm: polygonerna ritas i rätt ordning. Fortfarande ritas alla polygoner. De som ligger närmare ritas ju över de tidigare ritade som skyms. Vi slipper djuptestet men administrationen av polygonerna är något omständligare, så tidsvinsten blir oftast som mest marginell om ens någon.

En variant innebär att man ritar i omvänd ordning. Men då blir resultatet ju fel. Men om man kunde se till att inte rita mer än en gång per bildpunkt? Det finns ett antal algoritmer för detta. En lösning håller de flesta grafikkort och OpenGL med. Det finns nämligen ett s k stencilminne med samma layout som bildminnet (8 bitar/bildpunkt på datorerna i 6220 och 6217). Det kan bl a användas för att räkna antalet gånger man ritat i en bildpunkt. Och man kan före ny ritning undersöka om värdet är 0 eller inte. Mina experiment med detta visar dock inte några påtagliga tidsvinster i förhållande till vanlig BSP.

Dolda ytor: Genomskinliga ytor 1(1)

Djupbuffertalgoritmen, som tillåter att objekten ritas i godtycklig ordning, klarar inte genomskinliga ytor utan betydande ändringar. I enklaste fall är det närmsta objektet genomskinligt så att vi ser delar av det näst närmsta, vilket betyder att någon form av sortering är nödvändig.

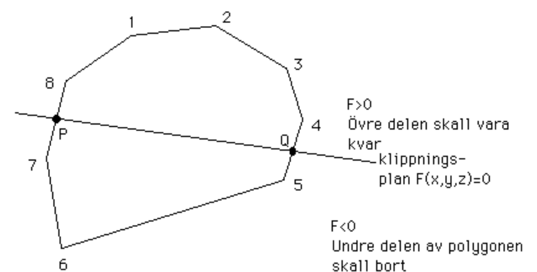
BSP i sin grundform klarar dem (bortsett från att vi inte ännu vet hur man skall blanda den genomskinliga och bakomliggande ytans färger).

BSP-varianten klarar dem inte.

Dolda ytor: Uppdelning i BSP 1(1)

Vi har en 3D-värld med objekt som är polyedrar, dvs kroppar med sidoytor som är polygoner. Ibland behöver vi tudela en polygon. Detta kan göras genom att man klipper två gånger mot delningsplanet. Låt oss förutsätta att polygonerna är konvexa. Då blir det principiellt enkelt. Skärningen mellan sidoytan och det aktuella klippningsplanet utgörs av ett enda streck (om nu inte planen sammanfaller).

En enkel (men inte den hastighetsmässigt bästa) algoritm för bestämning av den nya polygonen är:



1. Bestäm en hörnpunkt som inte skall vara kvar, t ex 6. Det är bara att leta efter en hörnpunkt med $F<0$.
2. Följ hörnpunkterna efter växande ordingsnummer, tills vi hittar en första hörnpunkt som skall vara kvar, dvs i figuren 8.
3. Fortsätt till den sista kvarvarande hittas, dvs här 4.
4. Start- och slutpunkterna P och Q i den nya polygonen P81234Q bestäms till sist med linjär interpolation.

Effektiviseringar 1(1)

Man kan säga att med en djupbuffert löses dolda-yt-problemet helt. Varje yta som skall ritas (kanske i form av trianglar) transformeras till normaliserade koordinater och klipps sedan av hårdvaran etc. Transformationerna har hittills skötts utanför grafikprocessorn (när det gäller konsumentkorten) men från och med NVidias GeForce läggs även dessa hos grafikprocessorn.

Pipeline-figur, se OH 37

Men detta betyder att scenens/världens alla ytor transformeras. Om man från början kunde sortera undan en del som ändå inte kommer att synas, skulle mycket vara vunnet. Grovt sett har vi kategorierna: **frånvända ytor** (face-culling), **ytor utanför synpyramiden** (frustum-culling) och **ytor som döljs av andra ytor** (occlusion-culling).

Bild som exemplifierar: Konvext objekt, utanför synpyramid, dold yta.

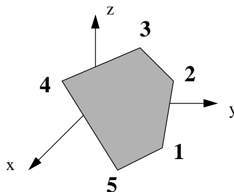
Men det betyder att vi måste lägga tester tidigare. Har vi en långsam dator och ett snabbt grafikkort uppnår vi troligen inte någon förbättring. Men annars. Och resultatet blir naturligtvis bäst om vi kan testa många polygoner i ett svep, dvs med begränsningsobjekt.

Hur skall detta nu gå till? Vi återkommer till frågan längre fram i kursen, men vill ge partiella svar redan nu.

- I världskoordinater, dvs modelleringssteget måste utföras. I princip måste vi nu kolla om punkten, ytan, begränsande objektet kolliderar med synpyramiden eller inte, vilket innebär klipptest mot 6 plan i världskoordinatsystemet. Om vi inte har kollision kan vi undanta objektet i fråga. I annat fall upprepas förfarandet på lägre nivåer eller också skickas innehållet till grafikprocessorn. Klipptestet görs i enklaste fall genom insättning av ett antal punkter i de olika planens ekvationer. Bara genom att testa mot främre klipp-planet skulle vi i en värld med likformig fördelning av objekten bli av med cirka hälften. Se även "Från värld till skärm", avsnitt 9, som dock tas upp senare i kursen.
- Man skulle också kunna tänka sig att grafikprocessorn matades med BB (begränsande boxar) (en eller två) och skickade svar. Detta eftersom grafikprocessorn ändå är bra på liknande saker. Jag känner dock inte till någon sådan.
- LOD (Levels Of Details, försvenskat detaljnivåer), som vi tar upp senare.

I verkligheten används scengrafer (Java3D, Optimizer etc) för att organisera det hela. Väsentligen organiseras objekten (objektsamlingar) - inte polygonerna - tillsammans med någon form av begränsningsinformation i en hierarkiskt träd (graf). Härigenom kan vi klippa bort stora delar på en gång. Observera att normalt går det fortare att rita när objekten blir mindre på skärmen, dvs vi får automatiskt en marginell förbättring. Men denna kan ökas på med klippning etc. Å andra sidan kostar extra tester.

Polygoner: Fram- och baksida



Begreppet **framsida** har betydelse i två situationer

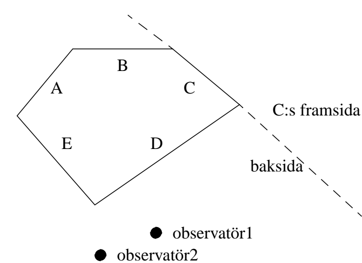
- Belysning (eng. lighting)
- Gallring (eng. culling)

Numrera polygonens hörn successivt. Den sida som syns när man tittar så att hörnen upplevs ordnade moturs är framsidan. I figuren ovan är alltså den sida som vetter mot origo baksida och den som läsaren ser framsida. Hade vi numrerat punkterna i omvänd ordning hade det varit tvärtom.

OpenGL: I standardfallet är det som ovan (men begreppet framsida definieras litet mer tekniskt med hjälp av den ytformel som nämns i pappret om beräkningsgeometri). Av detta skäl har jag rekommenderat att man alltid ordnar hörnen i slutna objekt moturs sett från utsidan. Man kan emellertid ställa om. Standardfallet motsvarar `glFrontFace(GL_CCW)` och det andra fallet `glFrontFace(GL_CW)`. CCW = Counter Clock-Wise, CW = Clock-Wise.

Polygongallring (frånvända ytor) i OpenGL

I normalfallet ritas en polygon alltid. Men om polygonerna utgör sidor till slutna objekt som vi inte avser att tränga in i finns det ingen anledning att rita dem som har yttersidan vänd från betraktaren. Genom att numrera konsekvent (moturs) är yttersidorna just framsidorna, dvs frånvända framsidor skall inte ritas.



C:s framsida kan inte ses av observatör1 och är alltså frånvänd. Detta gäller så länge som observatören finns på C:s baksida. Samma gäller sidorna A, B och E. För observatör2 är det sidorna A, B och C som är frånvända.

Med `glEnable(GL_CULL)` och `glDisable(GL_CULL)` kan vi slå på och av gallring. De polygoner som gallras bort är de som vänder framsidan från observatören. Härigenom kan man för stora scener bli av med ungefär hälften av alla polygoner med litet arbete. Gallringsmetoden kan ändras med `glCull(GL_BACKFACE)` och återställas med `glCull(GL_FRONTFACE)`.

Navigering: Ett steg fram, ett steg upp och ett åt sidan 1(8)

I interaktiva sammanhang vill observatören kunna röra sig i världen. Rörelsen kan styras med tangenter eller oftast bättre med musen (se OGL-häftet) eller annat organ. Vi väljer dock tangentalternativet här.

Den enklaste formen av rörelse är translation, dvs rörelsen består av steg i x-led, y-led eller z-led.

Vi inför sex globala variabler (snyggare med poster, dvs `struct` i C)

```
// observatörens pos
double pos_x=0.0, pos_y = 0.0, pos_z = 3.0;  // t ex
// betraktningspunkt
double at_x=0.0, at_y = 0.0, at_z = 0.0; // t ex
```

I uppdateringsproceduren (*display*, *update* el dyl) placerar vi

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(pos_x,pos_y,pos_z,at_x,at_y,at_z,0,1,0);
```

Vill vi interaktivt ändra även perspektiv placerar vi *gluPerspective* i samma procedur.

I tangentproceduren skriver vi (rörelse i x-led)

```
if (key == 'x') { pos_x = pos_x - dx; at_x = at_x - dx; }
if (key == 'X') { pos_x = pos_x + dx; at_x = at_x + dx; }
...
glutPostRedisplay();
```

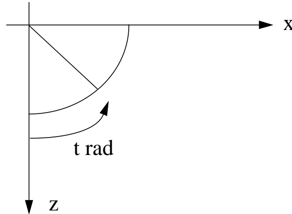
där *dx* är steglängden. Vi ändrar alltså betraktningspunkten samtidigt med positionen, vilket förefaller naturligt. Motsvarande för y- och z-komponenterna.Valet av tangenter bör ske med omsorg och med hänsyn till användaren. Piltangenterna räcker inte riktigt till för våra sex möjliga rörelser.

Navigering: Julgransdans 2(8)

En annan variant av rörelse är rörelse längs en kurva i zx-planet och med blicken fäst på en viss punkt, säg origo. Som exempel kan vi ta rörelse runt en cirkel

$$z = R\cos(2\pi t), x = R\sin(2\pi t), \quad 0 \le t \le 1$$

varvid utgångspositionen är (0,0,R). Världskoordinater.



Samma globala variabler som förut men med pos_z = R och dessutom
`double t = 0.0;`
Samma uppdateringsprocedur.

I tangentproceduren skriver vi

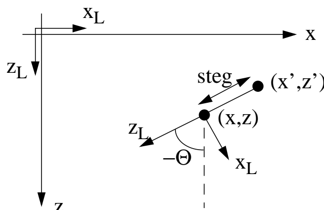
```
if (key == 't') t = t - dt;
if (key == 'T') t = t + dt;
pos_z = R*cos(2*M_PI*t); pos_x = R*sin(2*M_PI*t);
glutPostRedisplay();
```

där *dt* är steglängden.

Anm. *M_PI* brukar vara definierad i *math.h* och betecknar π.

Navigering: Fri som en fågel 2D 3(8)

De hittills beskrivna rörelserna är rätt tvungna. Vi vill röra oss friare. Och då vore det fint att kunna efterlikna en fågels (eller flygplans rörelse). Låt oss börja med motsvarigheten i 2D där även mänskliga rörelser kan vara förebild. Vi rör oss i en viss riktning, men kan när som helst få för oss att byta sådan genom att på plats rotera.



Vi befinner oss i (x,z) och tar ett steg i rörelseriktningen som vi i konsekvensens namn låter vara motriktad den lokala z-axeln. Givet en rotationsvinkel kan vi nu naturligtvis räkna ut den ny positionen (x',z') och därmed också en ny betraktningspunkt. Betraktningspunkten ändras vid vridning. Vi kan t ex låta steget vara 0.1 av skillnaden mellan betraktningspunkten och ögat.

Samma uppdateringsprocedur som förut. I tangentproceduren där vi hanterar vridning och rörelse framåt och bakåt skriver vi

```
if (key == 'v') {vinkel = vinkel - dv;at_x = ...;at_z = ...;}
if (key == 'h') {vinkel = vinkel + dv;at_x = ...;at_z = ...;}
if (key == 'f') {pos_x=...;pos_z=...;at_x=...;at_z=...:}
if (key == 'b') {pos_x=...;pos_z=...;at_x=...;at_z=...:}
glutPostRedisplay();
```

där ... står för uttryck som innehåller bl a cos(vinkel) och sin(vinkel).

Navigering: Ännu friare 2D 4(8)

Metoden på förra OH innebär att programmeraren måste räkna en aning (eftersom jag inte anger färdiga formler). Och var och en som försökt vet att man lätt gör fel. Och det blir än värre i 3D.

Men vi kan överlåta rutinarbetet till OpenGL mot att vi tänker extra. Hittills har vi räknat ut observatörens position och en punkt på synlinjen. Det behövs inte!

Resonemangen nedan bygger på att en translation eller rotation av observatören upplevelsemässigt är samma sak som att objekten i världen translateras eller roteras åt motsatt håll.

Praktiskt

- **Inför en variabel för aktuell modellvy-matris, t ex** `GLfloat modelview[16]`**.**
- **Utgångsläget sätts i reshape-proceduren** med `gluLookAt`:

  ```
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(pos_x,pos_y,pos_z, at_x,at_y,at_z,
            0.0,1.0,0.0);
  glGetFloatv(GL_MODELVIEW_MATRIX, modelview);
  ```

  Den sista lagrar modellvymatrisen i `modelview`. Parametrarna till `gluLookAt` förutsättes initierade. Med denna kodning kommer varje storleksförändring av fönstret att återskapa utgångsläget (vi kommer nämligen inte att ändra *pos_x* etc). Med lätt annorlunda kodning kan vi få ett vettigare beteende.

Navigering: Ännu friare 2D, forts 5(8)

- **I tangentproceduren**

  ```
  glMatrixMode(GL_MODELVIEW); // För säkerhets skull
  glLoadIdentity();
  if (key == 'f')  { // Framåt
     glTranslatef(0,0,0.01);// Världen flyttas närmare oss
  }
  if (key == 'v')  { // Vänstersväng kring min y-axel
     glRotatef(-1,0,1,0); // Världen roteras åt höger
  }
  ... // Övriga fall
  glMultMatrixf(modelview);
  glGetFloatv(GL_MODELVIEW_MATRIX,modelview);
  ```

  Här räknas ut en modellvy-matris motsvarande den ytterligare transformationen och denna multipliceras från höger med den tidigare modellvy-matrisen, vilket ger en total modellvy-matris som läses av och lagras i `modelview` (jfr OH 39, 54). När denna nya matris senare appliceras på punkter kommer den extra transformationen att utföras sist som sig bör.
- **I omritningsproceduren**

  ```
  glMatrixMode(GL_MODELVIEW); // För säkerhets skull
  glLoadIdentity(); // Helt onödig
  glLoadMatrixf(modelview);
  Rita det som ritas skall
  ```

Inte tillstymmelse till sinus och cosinus! Observatören bär med sig ett lokalt koordinatsystem som flyttar på sig och vrider sig i världen. Förändringarna görs alltid relativt detta lokala koordinatsystem. I koden ovan har vi valt att rotera 1° i taget och translatera 0.01 enheter i taget. Risken för felackumulering är nog inte obefintlig.

I 2D har vi "6 frihetsgrader" (de flesta skulle nog säga tre): flyttning fram, bak, vänster, höger samt rotation åt vänster resp höger.

Navigering: Ännu friare 2D 6(8)

Kanske det kan vara bra med ytterligare förklaringar. Låt V vara matrisen motsvarande den övergång mellan världskoordinater och vykoordinater som sattes upp med *gluLookAt*. Denna är OpenGL:s modellvymatris M i slutet av *reshape* och den läses där av och lagras i vår egen variabel *modelview*. När vi första gången trycker på en (korrekt) tangent, beräknas en ny modellvymatris L_1 motsvarande den lokala transformationen (t ex en liten translation eller rotation). Denna nya modellvymatris multipliceras med hjälp av *glMultMatrix* från höger med vår *modelview*, dvs nu är modellvymatrisen L_1V. Den läses av och lagras i *modelview* som därmed nu innehåller L_1V.
I omritningsproceduren ser vi till att modellvymatrisen verkligen är denna (även raden *glLoadMatrix* verkar onödig i detta enkla fall). Efter ett antal tangenttryckningar kommer OpenGL:s modellvymatris att vara
$M = L_n...L_2L_1V$,
där L_n är den sista lokala transformationen och L_1 den första. Detta gör att punkterna först transformeras till det vykoordinatsystem som bestämdes av *gluLookAt*, sedan i tur och ordning till de nya lägena av detta som åstadkommes av de successiva transformationerna. Dvs precis som vi tänkt oss. I OpenGL görs förstås allt i ett svep eftersom matriserna multiplicerats till en enda.

Om vi dessutom transformerar ett objekt som ritas med säg matrisen Mod och tar hänsyn till projektionen (uppsatt med *gluPerspective*) Proj, blir den totala transformationsmatrisen
$ProjL_n...L_2L_1Mod$

Återigen precis som sig bör.

Navigering: Fri som en fågel 3D 7(8)

Resonemangen på OH 83-85 kan direkt överflyttas till 3D. Det tillkommer "6 frihetsgrader": rotation åt två håll kring x- och z-axeln sam förflyttning uppåt och nedåt. För övrigt behöver inte kodningen ändras.

Skulle man av någon anledning vilja veta var man befinner sig, går det att räkna ut från modellvy-matrisen med det sista sambandet i avsnitt 2 i "Från värld till skärm" (det behövs en matrisinvertering).

Vid 3D-navigering använder man ofta flygteknisk terminologi:
Rullning (eng roll; rotation kring fågelns längsgående axel, z-axeln)
Girning (eng yaw; rotation kring uppåtaxeln, y-axeln)
Stigning (eng pitch; rotation kring sidoaxeln, x-axeln)
I matematikterminologi talar man i stället om **Eulervinklar**.

Det finns andra sätt att koda navigeringen. T ex ger "OpenGL Programmers Guide" förslaget att man ersätter det sedvanliga gluLookAt-anropet i omritningsproceduren med

```
glRotated(roll, 0, 0, 1);
glRotated(yaw, 0, 1, 0);
glRotated(pitch, 1, 0,0);
glTranslate(-pos_x, -pos_y,-pos_z)
```

eller med ett anrop av en procedur `fly()` med dessa rader. Position och betraktningspunkt måste räknas ut som förut. Nu kan ett annat problem uppstå (eng. gimbal lock = kardanknutslås?), som vi dock inte går in på närmare här (se Watts bok).
Mycket mer kan diskuteras. T ex rörelse längs föreskriven bana (kurva) med blicken framåt. Reglering av hastigheten både när det gäller vinkelförändringar och förflyttningar framåt. Men vi sätter stop!

Navigering: 8(8)

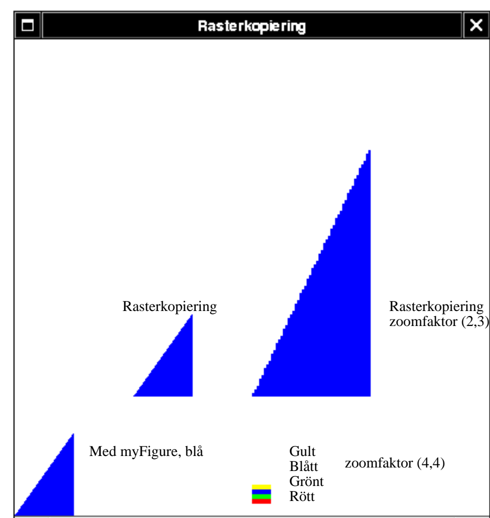
I OpenGL-häftet (avsnitt 16, exempel 10) visas ett intuitivt sätt att med musen styra rotationsvinklar.

Hittills har vi huvudsakligen använt tangenter för styrning. Repetionsfrekvensen för dem på min PC är låg (kanske bara en inställningssak), vilket gör att rörelsen blir slöare än nödvändigt. Ett sätt som gjorde mig gladare var att använda tangenterna bara som tillståndsändrare och starta/stoppa rörelsen genom att trycka ned respektive släppa upp en mustangent. För att tillståndsändring skall kunna göras under rörelsen behövs en idle-procedur (eller enklare med glutMotionFunc).

Praktiskt

- **Globala variabler**
```
int musen_nere=FALSE; char KEY;
```
- **I main**
```
glutKeyboardFunc(tangenth);
glutMouseFunc(mush);
glutIdleFunc(idle);
```
- **Mushanterare**
```
void mush(int btn,int state,int x,int y){
musen_nere=state;}
```
- **Tangenthanterare**
```
void tangenth(unsigned char key, int x, int y){KEY=key;}
```
- **Idle**
```
void idle() {
   if (musen_nere==TRUE) {
      if (KEY == 'f')  {pos_x=..., pos_y=..., ..... }
      ... // Övriga fall
      glutPostRedisplay();
     }
}
```


Rasterkopiering, exempel 9, utbyggt 1(2)



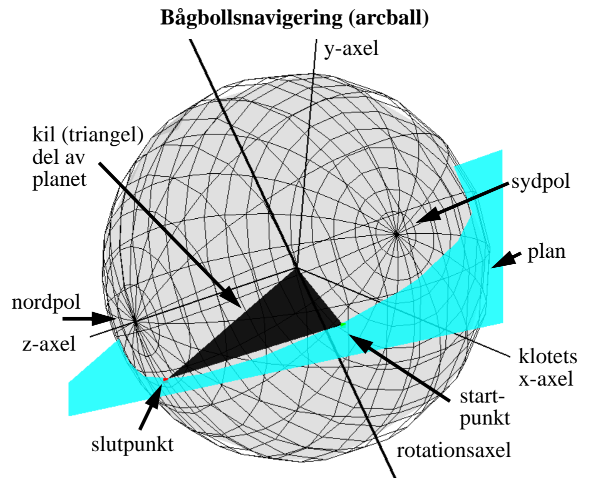
```
void myFigure(int x, int y) {
   glBegin(GL_POLYGON);
         glVertex2i(x,y);
         glVertex2i(x+50,y);
         glVertex2i(x+50,y+70);
    glEnd();
}

void display(void) {
    GLbyte Minne[30000];
    GLbyte M[48]={127,0,0,127,0,0,127,0,0,127,0,0,
           0,127,0,0,127,0,0,127,0,0,127,0,
           0,0,127,0,0,127,0,0,127,0,0,127,
           127,127,0,127,127,0,127,127,0,127,127,0};
    printf("Display called\n");
    glClearColor(1.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,0.0,1.0);
    myFigure(0,0);
    glRasterPos2d(100,100);
    glCopyPixels(0,0,50,70,GL_COLOR);
    glReadPixels(0,0,50,70,GL_RGB,GL_BYTE,Minne);
    glRasterPos2d(200,100);    glPixelZoom(2.0,3.0);
    glDrawPixels(50,70,GL_RGB,GL_BYTE,Minne);
    glRasterPos2d(200,10);    glPixelZoom(4.0,4.0);
    glDrawPixels(4,4,GL_RGB,GL_BYTE,M);
    glPixelZoom(1.0,1.0);

}

int GLOBAL_height;
void myReshape(int width, int height) {
    printf("Reshape called!\n");
    glViewport(0, 0, width, height);
    GLOBAL_height = height;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho (0.0, width, 0.0, height, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}
```


Rasterkopiering i JOGL

I förhållande till JOGL1_MB.java behöver vi som vanligt bara ändra *reshape* och *display*-

```
private void myFigure(int x, int y) {
   gl.glBegin(GL.GL_POLYGON);
           gl.glVertex2d(x,y);
           gl.glVertex2d(x+50,y);
           gl.glVertex2d(x+50,y+70);
   gl.glEnd();
}
public void reshape(GLDrawable drawable, int x, int y, int
width, int height) {
   gl.glViewport(0,0,width,height);
   gl.glMatrixMode(GL.GL_PROJECTION);
   gl.glLoadIdentity();
   gl.glOrtho(0.0, width, 0.0, height, -1.0, 1.0);
   gl.glMatrixMode(GL.GL_MODELVIEW); gl.glLoadIdentity();
   GLOBAL_height = height;
}
public void display(GLDrawable drawable) {
   System.out.println("Tid=: "+System.currentTimeMillis());
   byte[] Minne = new byte[30000];
   byte[] M={127,0,0, 127,0,0, 127,0,0, 127,0,0,
             0,127,0, 0,127,0, 0,127,0, 0,127,0,
             0,0,127, 0,0,127, 0,0,127, 0,0,127,
             127,127,0, 127,127,0, 127,127,0, 127,127,0};
   gl.glClearColor(1.0f,1.0f,1.0f,1.0f);
   gl.glClear(GL.GL_COLOR_BUFFER_BIT |
             GL.GL_DEPTH_BUFFER_BIT);
   gl.glColor3d(0,0,1); myFigure(0,0);
   gl.glRasterPos2d(100,100);
   gl.glCopyPixels(0,0,50,70,GL.GL_COLOR);
   gl.glReadPixels(0,0,50,70,GL.GL_RGB,GL.GL_BYTE,Minne);
   gl.glRasterPos2d(200,100); gl.glPixelZoom(2.0f,3.0f);
   gl.glDrawPixels(50,70,GL.GL_RGB,GL.GL_BYTE,Minne);
   gl.glRasterPos2d(200,10); gl.glPixelZoom(4.0f,4.0f);
   gl.glDrawPixels(4,4,GL.GL_RGB,GL.GL_BYTE,M);
   gl.glPixelZoom(1.0f,1.0f);
}
```


Bågbollsnavigering (arcball)



Om man vill kunna se på ett föremål från olika håll, kan man använda en s k bågboll (eng. arcball, virtual trackball), som är ett tänkt klot omgivande föremålet. Klotet (det tänkta) kan man sedan snurra på genom att trycka ner en mustangent och dra med musen. Detta ger en mycket intuitiv interaktion. Utgångspunkten och aktuell punkt definierar tillsammans med klotets mittpunkt ett plan (som ger en storcirkel på klotet och rörelsen kan beskrivas som en rotation kring planets normal. När vi väl känner start- och slutpunkt, vilket kräver en omvandling från muskoordinater till modellkoordinater, är det lätt att beräkna planet och dess normal och därmed rotationsaxeln och rotationsvinkeln. Det första steget görs med en metod *glUnproject*, som tas upp någon gång senare. Se även uppgift 3-4 på lab 2.

Fotorealism 1(1)

Den värld du skapar i laboration 2 liknar föga verkligheten. Vad är det som fattas?

- Ljus inkl reflektion av olika slag
- Texturering av olika slag, t ex för gräsmatta
- Skuggor

Vi skall här se på en enkel **belysningsmodell** och sedan se hur den förverkligas i OpenGL. Därefter ett par tekniker för att återge scener: **strålföljning** och **radiositet**. Sedan **texturering**. Snabba skuggor kommer separat senare.

Hittills har vi angett en färg för det som skall ritas. Men i verkligheten har objekten inte färger utan materialegenskaper. Det är dessa egenskaper i kombination med belysningen som bestämmer hur vi upplever världen.

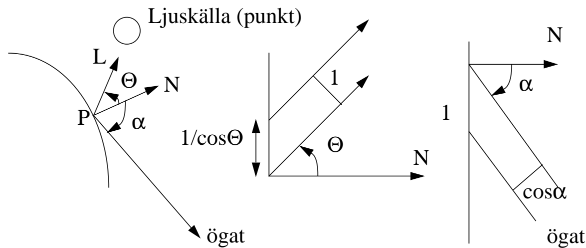
När vi säger att ett föremål har röd färg menar vi att det ser rött ut i normal belysning (vitt ljus). Orsaken är att av det vita ljus som träffar föremålet reflekteras bara den röda delen. Resten absorberas. Om vi belyser ett "rött" föremål med blått ljus, finns inget rött att reflektera och föremålet ter sig svart.

Vi byter alltså ut färgsättning mot materialegenskaper. Bland dessa finns:
Reflektionsfaktor för bakgrundsljus
Diffus reflektionsfaktor
Spegelreflektionsfaktor
Speglingskoncentrationsgrad

Fotorealism: En belysningsmodell 1(3)

Diffus reflektion

Vi vill beräkna intensiteten (energi/ytenhet) i punkten P, eller rättare sagt hur ögat upplever intensiteten i den punkten. N är den normerade normalen till ytan i P. L är en normerad vektor mot ljuskällan.



Från ljuskällan kommer i riktningen bestämd av L ljus med intensiteten I_e. Det betyder att energin i röret (genomskärningsytan 1) i mellersta figuren är I_e. Denna energi sprids på en något större area, dvs intensiteten på ytan blir $I_e \cos\Theta$. Delar av den infallande energin reflekteras. Vi antar att diffusa reflektionsfaktorn är k_d. Det betyder att den utgående intensiteten är $I_d = k_d I_e \cos\Theta = k_d I_e (L \bullet N)$. Vid diffus reflektion brukar man säga att denna upplevs lika oberoende av ögats position. Men egentligen är det litet omständligare. Ljuset sprids proportionellt mot $\cos\alpha$, dvs i röret i den högra figuren kommer energin $I_d * \cos\alpha$ (vi glömmer en fördelningskonstant; vi glömmer också den tredje dimensionen). Denna fördelar sig på arean $\cos\alpha$ (vinkelrät mot ögat), dvs den upplevda intensiteten blir I_d, precis som det sas för tre meningar sedan.

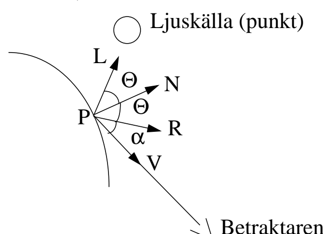
Fotorealism: En belysningsmodell 2(3)

Bakgrundsljusreflektion

$I_a = k_a I_{ambient}$

Speglingsreflektion

Samma förutsättningar som för diffus reflektion. För ett perfekt speglande material skulle allt ljus reflekteras i **huvudreflektionsriktningen R**, men i verkligheten sker en viss spridning som avtar med vinkeln α, vinkelskillnaden mellan huvudreflektionsriktningen och vektorn V (normerad) mot betraktaren



En modell (Phong) - det finns andra - för spegelreflektionen är

$I_s = k_s I_e (\cos \alpha)^n = k_s I_e (R_{norm} \bullet V)^n$ där $R_{norm} = R/|R|$, $R = 2(N \bullet L)N - L$

I en variant (mindre räknearbete) ersätter man R med H = L + V (riktning mittemellan L och V) och låter α vara avvikelsen från normalen.

Strålande ytor

Vi kan ta med en sådan genom att införa ett bidrag $I_{e,ytan}$.

Fotorealism: En belysningsmodell 3(3)

Totalt får vi för den upplevda intensiteten i punkten
$I = I_a + I_d + I_s + I_{e,ytan}$

Ovanstående beräkningar måste i praktiken genomföras för varje basfärg för sig. Dvs faktorerna är olika beroende på om det gäller R, G eller B. Se tabell längre fram.

I praktiken dämpas ljus med avståndet och man har en gång fått lära sig att intensiteten avtar med kvadraten på avståndet. I datorgrafiksammanhang struntar man ofta i detta beroende och de gånger man tar hänsyn till dämpningen antar man ofta att dämpningen är linjär. I OpenGL kan man arbeta med en dämpningsfaktor av formen

$$\frac{1}{a + bd + cd^2}$$

där d är avståndet.

Om vi har flera ljuskällor, N st:

$$I = I_a + \sum_{1}^{N} I_{e,i}\left(k_d(N \bullet L_i) + k_s(V \bullet R_i)^n\right)$$

Det finns mer komplicerade belysningsmodeller (se t ex Hills bok, avsnitt 14.7.3 som beskriver Cook-Torrance modell) som kan ge ett mera realistiskt resultat. Den vi beskrivit är den som vanligen förekommer i grundläggande datorgrafikböcker.

Fotorealism: Materialvärden

McReynolds och Blythe anger i sin kurs “Avancerad OpenGL” (finns på CD’n i *OPENGL2000/SGICOURSE*; se även länk på kurssidan) given vid SIGGRAPH ett par gånger lämpliga värden för ett stort antal material. När flera rader förekommer gäller de i tur och ordning R-, G- och B-komponenterna.

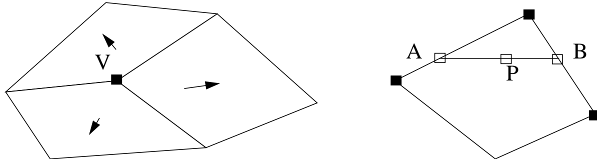
Material	k_a	k_d	k_s	**n**
Silver	0.19225	0.50754	0.508273	51.2
Koppar	0.19225 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.137622 0.086014	12.8
Guld	0.24725 0.1995 0.0745	0.75164 0.60648 0.22648	0.628281 0.555802 0.366065	51.2
Mässing	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974

Hill har en fullständigare tabell.

Fotorealism: Toningssätt (eng shading; målningssätt?)

Vi har en modell uppbyggd av polygoner och vill återge den i en värld med belysning. Tre olika sätt (jfr t ex programmet FACES):

- **Platt toning**: Man räknar ut ett belysningsvärde med hjälp av polygonens normal och belysningsmodellen. Värdet används som intensitet över hela polygonen.
- **Gouraudtoning**: I allmänhet är våra modeller approximationer av krökta modeller. Platt toning bidrar till att approximationen upplevs som kantig. Gouraud-toning gör att man får variation över ytan och samtidigt kontinuitet vid kanterna. Metoden innebär att man beräknar en medelnormal för varje hörn utifrån angränsande polygoners normaler. Med denna normal och belysningsmodellen beräknar man ett belysningsvärde i hörnet. För att beräkna ett värde i punkten P interpolerar man först fram ett värde i punkterna A och B och interpolerar sedan mellan dessa. Stöds av OpenGL.



- **Phong-toning**: Med Gouraud-toning kommer ljusintensiteten att variera linjärt över en enskild polygon. Men detta är uppåt väggarna speciellt när det gäller spegelreflektion. I Phongs metod anstränger man sig litet mer. I varje hörnpunkt beräknas som förut en medelnormal. Utifrån dessa beräknar vi med interpolation normaler i punkterna A och B och med ny interpolation en normal i P. Slutligen beräknas med denna och belysningsmodellen en intensitet i P. Phong-toning har inget stöd i OpenGL (jo med VFP).

Fotorealism: Belysning i OpenGL 1(2)

Se OpenGL-häftet. Väsentligen den modell som beskrivits. Men i ett par fall litet fler detaljer (som vi förbigår).

Materialegenskaper (ersätter tillsammans med *glNormal glColor*)

```
             GL_FRONT,          GL_AMBIENT,
glMaterialfv(GL_BACK,           GL_DIFFUSE, 4-vektor)
             GL_FRONT_AND_BACK,GL_SPECULAR,
                                GL_EMISSION
glMaterialf(-"-,GL_SHININESS,koncentrationsgrad n)
```

Normaler (ersätter tillsammans med *glMaterial glColor*)

`glNormal3f(a,b,c)`

Skall vara normerad vid belysningsberäkningen, vilket kan kräva `glEnable(GL_NORMALIZE)` (åtminstone om *glScale* finns med i transformationskedjan).

Ljuskällor

```
                    GL_AMBIENT,
glLightfv(GL_LIGHTx,GL_DIFFUSE,4-vektor)
                    GL_SPECULAR,
                    GL_POSITION,
```

Man låter ljuset bestå av tre delar, medan vi tidigare betraktade det som enhetligt (lättast att låta alla vara lika). När det gäller positionen betyder en vektor med 0 i fjärde komponenten att en riktning till ljuskällan anges, medan ett värde 1 där betyder en äkta position.

Aktivera belysning

`glEnable(GL_LIGHTING)`+per ljuskälla `glEnable(GL_LIGHTx)`

Val av toningsmodell

`glShadeModel(`*`modell`*`)`, där *modell* kan vara `GL_FLAT` (platt) eller `GL_SMOOTH` (Gouraud).

Om vi vill se på ut- och insidor

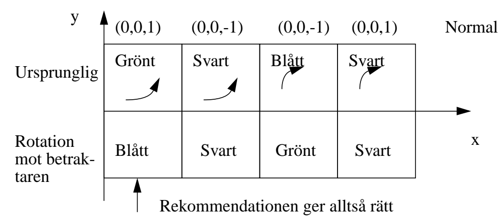
`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE)`

Fotorealism: Belysning i OpenGL 2(2)

Man ger ju ofta olika materialegenskaper åt polygoners fram- och baksida.

I en del grafiksystem låter man begreppet framsida definiera även polygonens normal. Så inte i OpenGL. Utan den (de) normaler som behövs för belysningsberäkningen är helt oberoende och anges med `glNormal` för en eller flera hörnpunkter. Vid flat toning (`glShadingMode(GL_FLAT)`) används en normal för hela polygonen. Vid Gouraud-toning (`glShadingMode(GL_SMOOTH)`) behövs en normal per hörnpunkt. **Normalerna skall vara utåtriktade från framsidan**, dvs om hörnen anges i motursordning och normalen är utåtriktad blir resultatet som vi tänkt oss. De flesta andra alternativ ger fel resultat, t ex en svart yta.

Exempel: En fyrkant i xy-planet med grön framsida och blå baksida (enligt glMaterial). Hörnen angivna i olika ordningar (enligt pilarna). Normalerna också olika. Program: GL_BELYS.

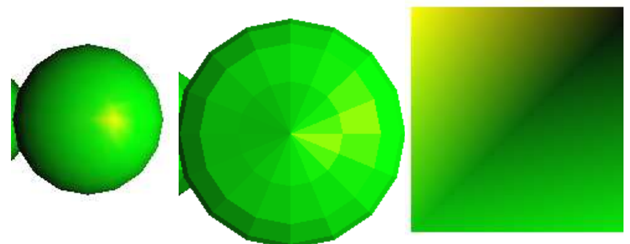


Gouraud- resp flat toning i OpenGL

I enklare fall anger man en normal per polygon (innanför `glBegin` eller utanför). Den gäller då samtliga hörn i polygonen. Eftersom samtliga hörn har samma normal ger flat toning och Gouraud-toning samma resultat.

I allmänhet anger man en normal per hörn (innanför `glBegin`). Om man begär flat toning används den först angivna för beräkning av polygonens belysningsvärde. Om Gouraud-toning begärs används i stället samtliga normalvärden på det sätt som tidigare beskrivits. För att resultatet skall bli annorlunda än vid flat toning måste normalvärdena skilja sig åt, vilket de gör om man **bildar hörnnormalerna som medelvärden av angränsande polygoners riktiga normaler. Detta steg måste du själv göra**.

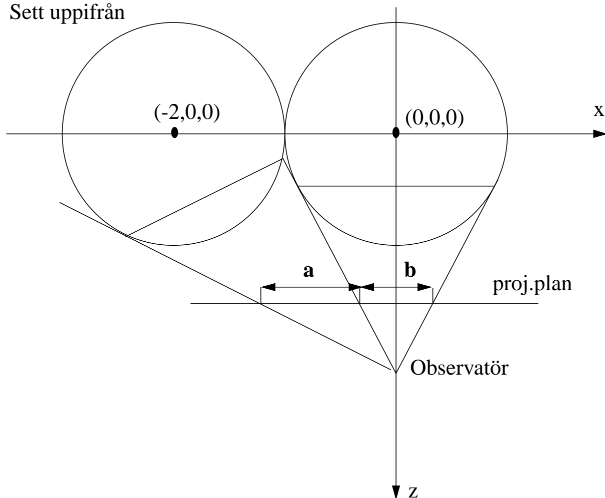
Alla GLUTs procedurer för färdiga objekt tillverkar automatiskt normaler för samtliga hörn. Alla utom `glutSolidCube` gör det så att Gouraud-toning fungerar. Detta val är naturligt, eftersom ett kantigt objekt som en kub bör se kantigt ut. T v en sfär med Gouraud-toning,



i mitten sfär med flat toning och t h fyrhörning med Gouraud-toning.

Exempel 17 i OGL-häftet: GL_ANIM.c (utv till GL_LJUS.c)

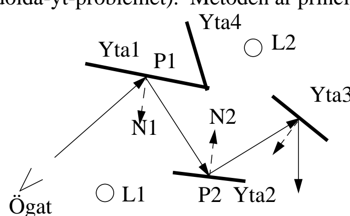
“Varför ser den vänstra sfären ut som en ellipsoid?” frågade en uppmärksam kursdeltagare. Orsaken är att vi bara ser delar av de båda sfärerna. Den högra döljer delvis den vänstra (vilket vi också noterade i datorbilden). Man kan naturligtvis räkna fram att a>b, men jag hoppas att det framgår ändå.



På längre observatörsavstånd blir det mera normalt. Blir det så här i verkligheten också? Blås två jättelika ballonger eller gjut stora betongklot och kontrollera.

Fotorealism: Strålföljning 1(3)

För att få global fotorealism får man ta i mer. En metod är **strålföljning** (eng ray tracing; eng ray casting är en förenklad variant som bara löser dolda-yt-problemet). Metoden är principiellt enkel.



Ögat tittar från sin position i en viss riktning. Vi bestämmer vilka ytor som träffas av strålen och motsvarande skärningspunkter. Vi tar reda på den närmsta av dessa. I figurens fall träffas Yta1 och Yta4. Punkten P1 på Yta1 är närmst. Vi bestämmer till att börja med ljusvärdet i P1 med den tidigare ljusmodellen. Vi tar med bidrag enbart från de ljuskällor som är synliga från P1, dvs vi måste kontrollera om linjen från P1 till ljuskällan skär något någon yta. I figurens fall döljs inte L1 men däremot L2 (av såväl Yta1 som Yta2). Vi tar alltså med enbart bidraget från ljuskälla L1. Till det nu beräknade ljusvärdet lägger vi den från P2 eller motsvarande reflekterade ljuset (efter multiplikation med sedvanlig reflektionsfaktor). Vi följer dock bara huvudreflektionsriktningarna. Vi är därmed tillbaka i nästan samma situation som från början. Men nu gäller det att bestämma hur en betraktare i P1 upplever punkten P2. Vi tar därmed lämpligen till rekursion. Rekursionen avbryts när bakgrunden (ingen yta) träffas av strålen, rekursionsdjupet är för stort, objektet är föga reflekterande eller ett strålande material påträffas (utbredd ljuskälla).

Fotorealism: Strålföljning 2(3)

Blender använder enbart OpenGL:s belysningsverktyg. Ingen strålföljning (man kan koppla en strålföljare till Render-steget).

Ett trevligt strålföljningsprogram är *PovRay*. Finns för PC och Sun/ Solaris. Modellen byggs i en textfil. Eller med hjälp av en modellerare (*Moray* - bara för PC och kostar; *Art Of Illusion* - javaprogrammet).

I ett strålföljningsprogram använder man obetydligt av OpenGL. Egentligen bara 2D-grafik med ritning av en punkt. Dock görs runt om i världen diverse försök att använda fragmentprogram för strålföljning. NVIDIA har en produkt (Gelato; programvara + Quadro FX) som möjligen fungerar så (dokumentationen är oklar).

Strålföljning lämpar sig för parallellism.

Strålföljning klarar även genomskinliga föremål, men vi tar inte upp några detaljer här. Vad man gör är att man följer inte bara en reflektionsriktning utan även en brytningsriktning genom det genomskinliga materialet. Det är förhållandevis enkelt att lägga till när resten av programmet är klart. Man behöver naturligtvis nu utökade materialdata (brytningskoefficient och genomsläpplighetsfaktor).

Eftersom strålföljning i sina rena form inte är en realtidsalgoritm, finns det diverse “fusk”-sätt som kan lura den oinitierade.

Fotorealism: Strålföljning. Program 3(3)

Ett programskelett:

0. Vi har en lista (vektor) med alla objekt
1. För varje bildpunkt
   1. Beräkna motsvarande punkt i världen
   2. Beräkna strålen (startpunkt, riktning) från ögat till denna punkt
   3. Beräkna färg(stråle,anropsdjup)

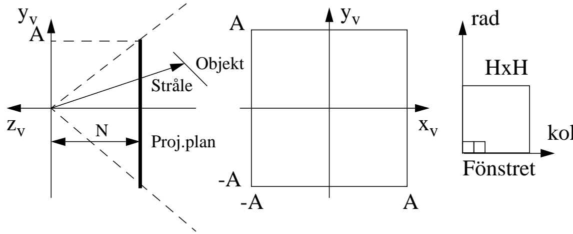
Funktionen färg(stråle,anropsdjup):

1. Beräkna strålens skärningar med ytor. Träffar “bakom” strålens utgångspunkt ointressanta.
2. Om det inte finns några intressanta träffar returnera med bakgrundsfärgen.
3. Låt annars P vara den närmsta
   1. Ljusvärde = emissionsljus + omgivningsljus (vi måste själva räkna; OpenGL hjälper inte längre till).
   2. För alla punktformiga ljuskällor
      1. Om ljuskällan synlig ljusvärde = ljusvärde + diffus- och speglingsbidrag från ljuskällan (dito)
   3. Om punkten P reflekterande nog och anropsdjupet lågt nog ljusvärde = ljusvärde +
      färg(stråle i huvudreflektionsriktningen, anropsdjup+1)
   4. Returnera ljusvärdet

Huvudproblem: Träffar.

Strålföljning: Bildpunkt till värld 1(1)

Med bildens beteckningar får vi (för enkelhets skull antas att fönstret är kvadratiskt och att synkvoten är 1)



att en bildpunkt (kol,rad) motsvarar i vykoordinatsystemet $P_v = (x_v,y_v,z_v)^T$, där

$x_v = A(-1 + 2*kol/H)$, $y_v = A(-1 + 2*rad/H)$, $z_v = -N$

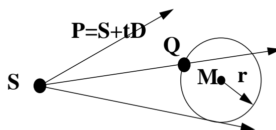
Uttryckt i världskoordinater blir punkten

$$\mathbf{P}_w = \mathbf{P}_0 - N\hat{\mathbf{z}}_\mathbf{v} + x_v\hat{\mathbf{x}}_\mathbf{v} + y_v\hat{\mathbf{y}}_\mathbf{v}$$

där P_0 är ögats placering i världskoordinatsystemet och $\hat{\mathbf{x}}_v$ etc avser de normerade vykoordinataxlarna uttryckta i världskoordinater (se “Från värld till skärm”).

Strålföljning: Skärning stråle och sfär 1(1)

En sfär med mittpunkt M och radie r är given. Vi vill veta om strålen med utgångspunkt i punkten S och riktningen D (gärna normerad så att A nedan är 1; då är t avståndet från utgångspunkten) träffar sfären. Vi kommer att bestämma t för att få ett svar på frågan. I figuren syns



tre olika fall. Notera att lösningar med t<0 saknar intresse (de ligger bakom betraktaren eller strålens utgångspunkt). Punkten Q skall ligga på både strålen och sfären, dvs

$$\begin{cases} Q = S + tD \\ |Q - M| = r \end{cases}$$

Insättning av första ekvationen i andra ger $|S + tD - M|^2 = r^2$.

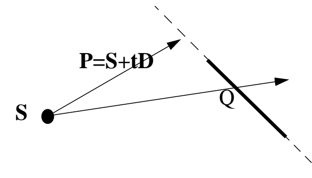
Med $S=(S_1,S_2,S_3)$ etc och beteckningarna (vektornotation kortare)

$$A = \sum_{i=1}^{3} D_i^2 \qquad B = \sum_{i=1}^{3} (S_i - M_i)D_i \qquad C = \sum_{i=1}^{3} (S_i - M_i)^2 - r^2$$

får vi andragradsekvationen $At^2 + 2Bt + C = 0$ med lösningar $t = \frac{1}{A}\left(-B \pm \sqrt{B^2 - AC}\right)$. Diskriminanten avgör huruvida det finns 0, 1 eller 2 lösningar. Vi accepterar bara t>0.

Strålföljning: Skärning stråle och polygon 1(1)

En plan polygon med hörn V_1, ..., V_N är given. Vi vill veta om strålen med utgångspunkt i punkten S och riktningen D träffar polygonen.



Vi kan som vanligt lätt bestämma ekvationen F(x,y,z)=Ax+By+Cz+D=0 för det plan i vilket polygonen ligger. Vi börjar med att leta efter eventuella skärningar mellan strålen och detta plan. I figurens fall skär båda strålarna planet, men bara den understa skär polygonen. Vi får ekvationerna

$$\begin{cases} Q = S + tD \\ F(Q) = 0 \end{cases}$$

och efter insättning av den första i den andra

$$t = \frac{AS_1 + BS_2 + CS_3 + D}{-(AD_1 + BD_2 + CD_3)}$$

Som förut accepteras bara värden t>0. Återstår sedan att kolla att punkten verkligen hamnat inom polygonen. Vi tar upp detta problem senare under Beräkningsgeometri.

Strålföljning

Detta var en rask introduktion till strålföljning. Det finns åtskilliga andra problemställningar förknippade med strålföljning, som man inte anar omedelbart.

Eftersom metoden i sig är långsam är det extra viktigt att kunna göra olika slag av effektiviseringar, som t ex hindrar att vi försöker beräkna skärningar med objekt som ligger utanför synpyramiden.
Realtidsförsök: Parallellism, algoritmer, lägre upplösning vid rörelse, ny hårdvara.

Ett strålföljningsprogram PovRay 1(3)

Finns i såväl UNIX- som PC-version (snyggare användargränssnitt). Fritt. Det använder sig av ett scenbeskrivningsspråk. Parametrarna har namn som påminner om de vi mött, men betydelsen är inte alltid lika. En utmärkt manual finns. Koordinatsystemet är vänsterorienterat - och inte som hittills högerorienterat - om du börjar undra. Nu version 3.7.

Om scenbeskrivningen ligger i filen `Ex3.pov` (i `$DG/EXEMPEL_MB`) startar man programmet med

```
povray +Ex3.pov +X +D +W512 +H512
```

Då växer en fotorealistisk bild fram i sakta mak. Den sparas automatiskt som `Ex3.png` (från 3.6; tidigare användes det gammaldagsa formatet tga)). P g a rättighetsmekanismer bäst att vara i den mapp där pov-filen finns.

Vill man ha radiositet lägger man i version 3.7 till en rad `radiosity{ }` i `global_settings`. Allt tar då längre tid.

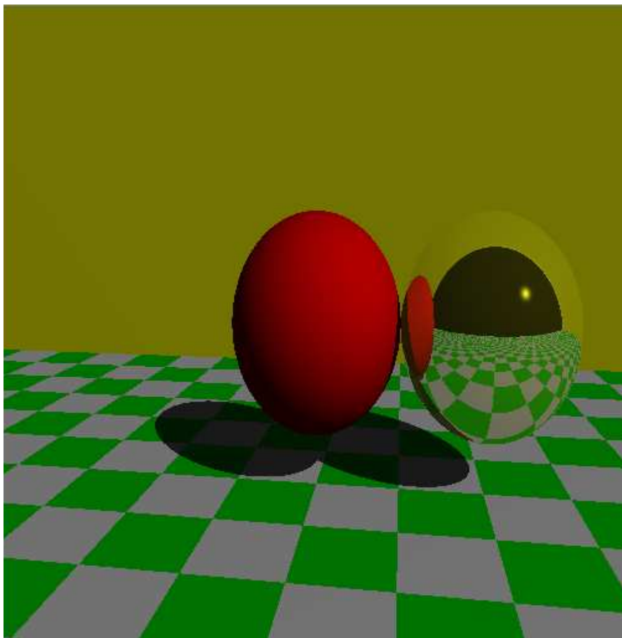
Exempel (nästa OH): Scen med två klot på ett plan. Bild efter koden.

Ett strålföljningsprogram PovRay 2(3)

```
> cat Ex3.pov
// En scen för PovRay
#include "colors.inc"
global_settings { assumed_gamma 2.2 }
// En kamera i (1,1,-7) som tittar mot (0,0,0)
camera { location <1, 1, -7>
         look_at <0, 0, 0>
         angle 56
}

// En vit ljuskälla snett och långt bakom kameran
light_source { <1000, 1000, -1000> White }

// Ett oändligt gult plan med normalen (0,0,1)
// och 6 enheter i den riktningen
plane { <0,0,1>, 6
        pigment {color rgb <1, 1, 0>}
}

// Ett horisontellt plan -1.5 under xy-planet
// med ett schackrutemönster
plane { <0, 1, 0>, -1.5
         pigment { checker Green, White }
}// En röd sfär i origo med radien 1
sphere { <0,0,0>, 1
         pigment { Red }
}

// En sfär med radien 1 och materialegenskaper
sphere { <2,0,0>, 1
         pigment { BrightGold }
         finish {
                 ambient .1  diffuse .1
                 specular 1 roughness .001
                 reflection .75 metallic
         }
}
```


Ett strålföljningsprogram PovRay 3(3)

Bilden



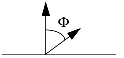
Exempel: Ett vitt ägg på ett bord med och utan radiositet (`Ex4R.pov` resp `Ex4.pov` i samma mapp).

Fotorealism: Radiositetsmetoden 1(3)

Vid strålföljning tar vi hänsyn till spegelreflektionen, men struntar helt i den diffusa reflektionen mellan objekt. Detta är inte korrekt. T ex kan ett vitt föremål placerat i ett rum med röda väggar få en rödaktig ton. **Radiositetsmetoden** tar hänsyn till den diffusa reflektionen mellan objekten och struntar i spegelreflektionen, men kan på olika sätt kombineras med strålföljning.

Metoden i stora drag

1. Förutsättningar



Alla ljuskällor och alla ytor är diffusa (dvs ingen spegelreflektion). Begreppet diffus innebär att ljuset sprids i de olika riktningarna med energin proportionell mot cos Φ, där Φ är vinkeln mot ytans normal. Intensiteten blir därmed (se figur på OH om diffus reflektion) lika oberoende av riktning, som vi antog i samband med belysningsmodell-resonemanget. Maximalt ljusflöde fås alltså i normalens riktning och inget alls längs ytan.

Ljuskällorna kan vara utbredda. Ytorna kan vara krökta. Scenen skall vara **sluten**.

2. De olika stegen

1. Samtliga ytor delas in i mindre delar A_1, A_2, A_3, ..., A_N. Intensiteten B_i på en sådan yta A_i antages vara konstant, vilket naturligtvis innebär en approximation och betyder att indelningen måste göras adaptivt, dvs så att delarna är små där intensiteten kan förväntas variera raskt.

Fotorealism: Radiositetsmetoden 2(3)

2. Nu beräknas alla B_i genom att man
   först bestämmer N^2 formfaktorer F_{ij}, $1 \leq i,j \leq N$, som ger ett rent geometriskt samband mellan ytorna A_i och A_j.
   sedan löser ett linjärt ekvationssystem med N ekvationer och N obekanta.
3. Nu kan observatören placeras godtyckligt utan att beräkningarna i 3 behöver göras om. Man löser bara synlighetsproblemet.
4. Ytorna ritas upp med flat toning eller Gouraud-toning.

3. Ekvationssystemet

Intensiteten B_j hos ytan A_j mäts i W/m^2. Om A_j får beteckna inte bara själva ytan utan även dess area, betyder det att den totala ljuseffekten från ytan A_j är B_jA_j. B_jA_j byggs upp av två komponenter. Dels kan ytan vara en ljuskälla med den emitterande intensiteten E_j (också med sorten W/m^2). Dels reflekterar den diffust ljus som når ytan A_j från andra ytor. Reflektionskoefficienten, som är en materialkonstant och antages vara konstant över A_j, betecknar vi med ρ_j. Detta innebär att

$$B_jA_j = E_jA_j + \rho_j \sum_{i=1}^{N} EnerginSomNarA_jFranA_i$$

Termen innanför summatecknet kan skrivas $F_{ij}B_iA_i$, där F_{ij} anger hur stor del av den totala energin från A_i som når A_j. Vi kommer senare att visa att $A_iF_{ij}/A_j = F_{ji}$, varav

$$B_j = E_j + \rho_j \sum_{i=1}^{N} F_{ji}B_i$$

Detta är vårt ekvationssystem (kallas radiositetsekvationen), som kan lösas med t ex gausselimination eller något iterativt förfarande (Gauss-Seidel brukar nämnas; systemet är diagonaldominant dvs den

Fotorealism: Radiositetsmetoden 3(3)

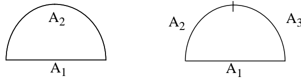
metoden konvergerar; diagonaldominansen följer av att $\sum_j F_{ij} \le 1$

Man kan ange komplicerade formler för formfaktorerna, t ex

$$F_{ij} = \frac{1}{\pi A_i} \int_{A_i} \int_{A_j} H_{ij} \frac{\cos\Phi_i \cos\Phi_j}{r^2} dA_i dA_j$$

men i praktiken måste de ändå approximeras.

4. Ett exempel



A2 A3
A1

I båda fallen är ytorna i praktiken för stora och borde delats in i mindre. Men vi vill bara få ett begrepp om vad formfaktorerna innebär.
I vänstra fallet når allt ljuset från ytan A_1 ytan A_2, dvs $F_{12}=1$ och $F_{11}=0$. Sambandet mellan de båda formfaktorerna F_{ij} och F_{ji} ger sedan $F_{21}=A_1F_{12}/A_2=A_1/A_2$ (där nu A_i betecknar area, varvid A_1=ytan av en enhetscirkel=π och A_2=ytan av en enhetshemisfär=2π). Resten av ljuset från A_2 måste nå A_2 själv, varför $F_{22}=1-A_1/A_2$. I högra fallet ger symmetrin att $F_{12}=F_{13}=1/2$ och som förut är $F_{11}=0$. Härur får vi som förut $F_{21}=A_1F_{12}/A_2=0.5A_1/A_2$. Men att bestämma F_{22} och F_{23} (och därmed F_{32} och F_{33}) förefaller inte helt lätt, så vi ger upp.

Radiositetsmetoden är en dyr metod, varför många system, bl a *Pov-Ray*, använder en approximativ metod, som innebär att det från varje bildpunkt skickas strålar i olika riktningar som undersöker om det finns näraliggande ytor som skall ge ett “diffust” bidrag.

Kurvor och ytor

Det finns två huvudmetoder för konstruktion av kurvor och ytor:

- Olika former av **splines**, framför allt B-splines och NURBS. Utgående från ett antal punkter sätts ett uttryck för kurvan eller ytan upp på parameterform. Kurvan ritas utifrån denna parameterframställning. Upplevs av de flesta som rätt matematiskt och krångligt. Stöds av OpenGL. Behandlas i det separata pappret *Kurv- och ytapproximation*, samt i OpenGL-häftet, avsnitt 24. Sammanfattas och kompletteras med ett antal följande OH.
- **Uppdelningsmetoder** (eng. subdivision). Ytligt sett mera praktiskt. Utgående från ett antal punkter inför man successivt nya punkter och modifierar samtidigt de gamla. Därefter ritas kurvan genom ett polygontåg genom punkterna. Man får automatiskt sina objekt i flera upplösningar. Att analysera metoderna matematiskt är däremot inte lätt. Inget direkt stöd i OpenGL. Liksom alla kommersiella modelleringsprogram har *Blender* och *Art Of Illusion* verktyg (om än begränsade) för uppdelning. Användningsområdet är ytor. Blev populära i slutet av 1990-talet. Vi ger en kort introduktion i form av några OH.

I båda fallen skiljer man på **interpolerande** kurvor/ytor och **approximerande** kurvor/ytor. Vi ägnar oss mest åt den senare typen. Kurvan/ytan går då inte säkert igenom de olika styrpunkter som man utgår ifrån.

B-splines: Sammanfattning 1(2)

Givet: n+1 punkter $\mathbf{P}_0$, $\mathbf{P}_1$, ..., $\mathbf{P}_n$.

Approximera med en kurva som interpolerar startpunkten $\mathbf{P}_0$ och slutpunkten $\mathbf{P}_n$ och styrs av övriga punkter. Kurvan skall vara styckevis sammansatt av tredjegradskurvor. Resultatet blir en kurva med kontinuerlig andraderivata.

Allmänt behövs n-2 kurvsegment $\mathbf{P}_i(t)$, $t_i \le t \le t_{i+1}$, i=1,..., n-2. Segmenten $\mathbf{P}_{i-1}(t)$ och $\mathbf{P}_i(t)$ går ihop vid $t=t_i$, som kallas **skarv** (eng knot). Speciellt sätter vi $t_1=0$ (första segmentets start) och vid likformiga B-splines $t_i=i-2$, dvs $t_{n-1}=n-2$ (sista segmentets slut).

Man kan skriva

$$\boldsymbol{P}_i(t) = B_{i-1}(t)\boldsymbol{P}_{i-1} + B_i(t)\boldsymbol{P}_i + B_{i+1}(t)\boldsymbol{P}_{i+1} + B_{i+2}(t)\boldsymbol{P}_{i+2}$$

I inre punkter gäller i likformiga fallet $B_i(t) = B(t-t_i)$, där

$$B(t) = \begin{cases} \frac{1}{6}(2-|t|)^3 & 1 \le |t| \le 2 \\ \frac{1}{6}[1 + 3(1-|t|) + 3(1-|t|)^2 - 3(1-|t|)^3] & |t| \le 1 \\ 0 & 2 \le |t| \end{cases}$$

B-splines: Sammanfattning 2(2)

Allmänt är basfunktionen $B_i(t)$, $0 \le i \le n$, bestämd av **skarvföljden** (eng. knot vector) $[t_{i-2},t_{i-1},t_i,t_{i+1},t_{i+2}]$. Den är "centrerad" kring t_i (i likformiga fallet $t_i=i-1$) och 0 utanför intervallet $[t_{i-2},t_{i+2}]$. För detta behövs dock 3 extra skarvar $t_{-2} = t_{-1} = t_0 = 0$ före de andra och 3 extra $t_n = t_{n+1} = t_{n+2} = n-2$ i slutet. $B_0(t)$ bestäms då av $[0,0,\underline{0},0,1]$, medan $B_1(t)$ bestäms av $[0,0,\underline{0},1,2]$ (om antalet punkter är minst 5) och $B_2(t)$ bestäms av $[0,0,\underline{1},2,3]$. På motsvarande sätt i den andra änden. Den ursprungliga skarvföljden om n-1 värden $[t_1,t_2,...,t_{n-1}]$ utökas på detta vis med 6 skarvar, dvs omfattar totalt (n+1)+4 skarvar eller 4 mer än antalet punkter. För interpolation i ändpunkterna skall de fyra första och de fyra sista i den nya följden vara sinsemellan lika.
Motsvarande NURBS-approximation är

$$\boldsymbol{P}_i(t) = \frac{\sum_{j=i-1}^{i+2} w_j B_j(t) \boldsymbol{P}_j}{\sum_{j=i-1}^{i+2} w_j B_j(t)}$$

Ett annat allmännare sätt att uttrycka basfunktionerna är med en **rekursionsformel** (detaljer i småskriften *Kurv- och ytapproximation*) över gradtalet (våra B_i är $B_{3,i}$, $B_{0,i}(t) = 1$ för $t_{i-1} \le t < t_i$ och 0 för övrigt).

$$B_{k,i}(t) = \frac{t-t_v}{t_{v+k}-t_v} B_{k-1,r}(t) + \frac{t_h - t}{t_h - t_{h-k}} B_{k-1,r+1}(t)$$

Med denna klaras godtyckligt gradtal och godtyckliga intervall-längder bekvämt, men det blir ju litet mer att göra för datorn. Och vi har inte härlett formeln.

OpenGL: Kubiska B-Splines och NURBS

OpenGL (snarare GLU) har stöd för B-splines med godtyckligt gradtal och de allmännare NURBS=**N**on-**U**niform **R**ational **B-S**plines (skämtsamt Nobody Understands Rational B-Splines).
OBS! Alla vektorer skall vara av float-typ (double duger inte).

- **Arbetsarea**:
```
GLUnurbsObj* theNurb;    (Java: long theNurb)
theNurb =  gluNewNurbsRenderer();
```
- **Styrpunkter**:(3->4 om NURBS)
```
GLfloat styrpunkter[antal_pkter][3];
// alt. GLfloat styrpunkter[3*antal_pkter];
```
- **Skarvar**:
```
int antal_skarv_punkter = antal_punkter+4;
GLfloat skarvar[antal_skarv_punkter] = {0,0,0,0,1,.. };
```
- **Vikter i NURBS-fallet**:
```
GLfloat vikter[antal_pkter];
```
- **Uppritning**:
```
gluBeginCurve(theNurb);
    gluNurbsCurve(theNurb, antal_skarv_punkter,
          skarvar,
           3, // avstånd i GLfloats mellan styrpunkter,
              // dvs 4 i NURBS-fallet
           styrpunkter,
           4, // gradtalet+1
           GL_MAP1_VERTEX_3 // eller motsv
    );
gluEndCurve(theNurb);
```

Exempel 20 i OGL-häftet.

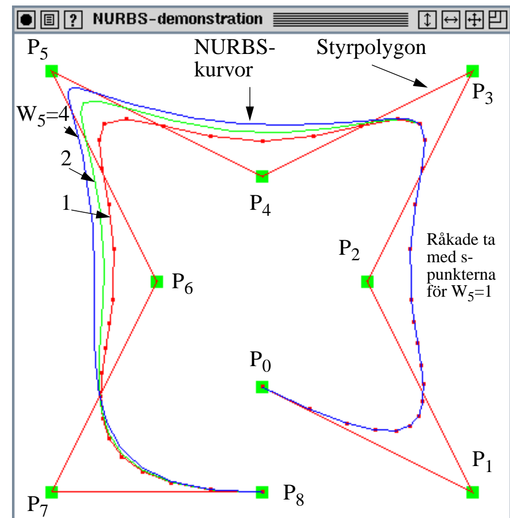
Exempel 20 i OGL-häftet med JOGL

JOGL har av någon anledning missat stödet för B-splines/NURBS, varför jag inte bifogar någon kod. Fungerade däremot i en av föregångarna, nämligen *OpenGL for Java*. Officiellt anges att man kommer att skriva om den del av GLU-koden som behövs i Java, i stället för att försöka anropa motsvarande kod skriven i C. Men man har inte lyckats hitta någon som är villig att göra jobbet. Att skriva en klass för sådana kurvor och ytor är inte svårt, men skall den ha lika många metoder som i GLU och utnyttja det stöd som OpenGL har för Bezierkurvor blir det värre. F ö frodas JOGL och i kombination med kommande Java 6.0 (våren 2006 kanske; betaversion går att hämta) fungerar OpenGL bra i Swing-fönster (GLJPanel; se sid 52 Anmärkningar i OpenGL-häftet).

C#/Tao verkar inte heller klara B-splines/NURBS. Det går visserligen att kompilera program med sådana, men vid körningen skapas inte de objekt som behövs utan man får nollreferenser. Tao tycks för övrigt stå still just nu.

NURBS i OpenGL 1(2)

Se OpenGL-häftet och kurvskriften. Här litet till.



Kubiska NURBS-kurvor. Alla vikter W_i=1 utom W_5 som varierar. Större värde ökar punktens attraktionskraft. Samplingspunkterna (se separat OH) råkade komma med för W_5=1.

NURBS i OpenGL 2(2)

Utdrag ur koden

```
Point PV[50]; // 2D-punkter
int Nr_Of_Points=9;
GLfloat CtrlPoint[Nr_Of_Points][4];
GLfloat W[] = {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0};
GLfloat t[] = {0,0,0,0,1,2,3,4,5,6,6,6,6};

// Bilda 3D-data i homogena koordinater
for (i=0; i<Nr_Of_Points; i++) {
  CtrlPoint[i][0]=W[i]*PV[i].x;
  CtrlPoint[i][1]=W[i]*PV[i].y;
  CtrlPoint[i][2]=0.0;
  CtrlPoint[i][3]=W[i];
}
// Ritningen (när alla vikterna lika)
gluBeginCurve(theNurb);
         gluNurbsCurve(theNurb,Nr_Of_Points+4,t,4,
         (GLfloat*)CtrlPoint,4,GL_MAP1_VERTEX_4);
gluEndCurve(theNurb);

// Ändra en av vikterna
W[5] = 2;
// Upprepa
```

Att notera betr parametrarna till *gluNurbsCurve*:

4:e parametern är nu 4 eftersom varje punkt består av 4 floats.

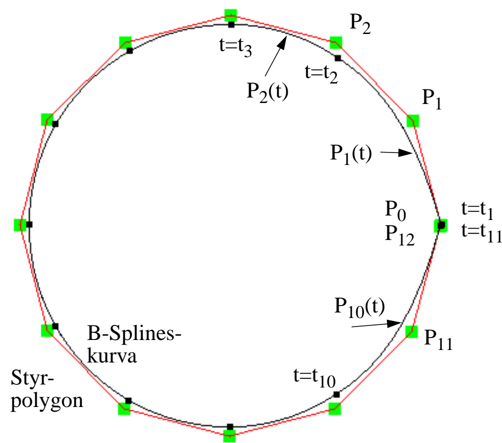
I 5:e parametern står `(GLfloat*)` enbart för att hindra en varning från kompilatorn.

6:e parametern är som vanligt gradtalet + 1 (gradtalet=3 eftersom det handlar om kubiska NURBS).

7:e parametern är `GL_MAP1_VERTEX_4` eftersom vi nu arbetar med homogena koordinater.

Cirkelapproximation med B-splines i OpenGL

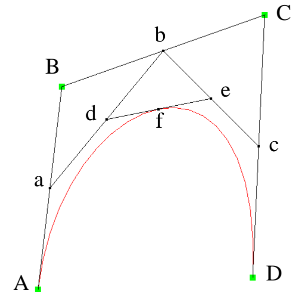
Vi väljer 12 (13) punkter ekvidistant belägna på cirkelns omkrets. Dessa är markerade med större (gröna) fyrkanter. Approximation med kubiska B-splines (med interpolation i start- och slutpunkt) ger en kurva som
a) är "spetsig"
b) ligger innanför den riktiga cirkeln



Spetsen kan förbättras genom att man t ex inför flera identiska start- och slutpunkter. Med NURBS (se följande OH) kan man beskriva cirkeln exakt.

Uppdelningsmetoder 1(7)

Vi börjar med att nämna de Casteljau's[1] geometriska metod för konstruktion av en enstaka Bezier-kurva.

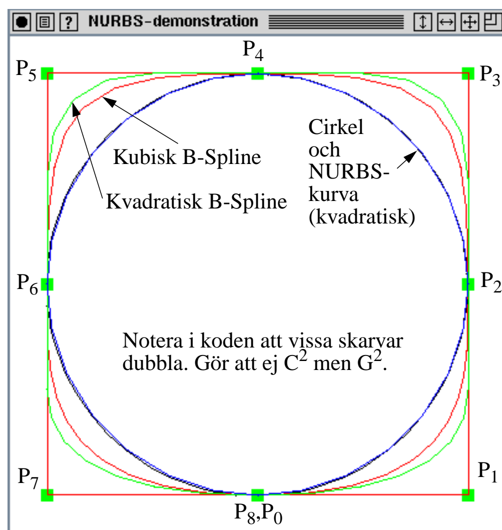


I figuren (`GL_BEZIER2005.c`) har vi fyra punkter A, B, C och D och är alltså ute efter en kubisk Bezier-kurva P(t). Vi konstruera punkten f, som är P(0.5), genom att först införa mittpunkter a, b och c på tre av styrpolygonens sidor. Därefter punkterna d och e som mittpunkter på linjerna ab respektive bc. Slutligen f som mittpunkt på linjen de. Processen kan upprepas. T ex för att beräkna P(0.25) arbetar vi på samma sätt på styrpolygonen Aadf. En fördel med metoder av detta slag är att vi har geometrisk kontroll. Hade t ex polygonen Aadf varit mycket smal hade vi kanske kunnat nöja oss med ett streck från A till f.

I själva verket kan man med metoden konstruera vilken som helst punkt på kurvan genom att göra delningen litet annorlunda. En motsvarighet till detta för B-splines är de Boor's metod, som dock är mycket omständligare.

1. Paul de Casteljau och Pierre Bezier var bilingenjörer. Den förre vid Peugot och den andre vid Renault. Båda jobbade med Bezier-kurvor utan att känna till varandras arbete.

Cirkelapproximation med NURBS i OpenGL




```
GLfloat t[] = {0,0,0,1,1,2,2,3,3,4,4,4};
GLfloat w = 1.0/sqrt(2.0);
GLfloat W[] = {1.0, w, 1.0, w, 1.0, w, 1.0, w, 1.0};
// Som förut och sedan ritning med kvadratiska NURBS
gluBeginCurve(theNurb);
              gluNurbsCurve(theNurb,Nr_Of_Points+3,t,4,
              (GLfloat*)CtrlPoint,3,GL_MAP1_VERTEX_4);
gluEndCurve(theNurb);
```

Det finns flera andra sätt att välja punkter. Man kan klara sig utan ett par av de vi valt. Se t ex Les Piegl: The Nurbs Book, Springer 1997.

Uppdelningsmetoder 2(7)

Låt oss nu titta på en uppdelningsmetod för kurvor. Precis som i splines-fallet startar vi med n+1 punkter $\mathbf{P}_0$, $\mathbf{P}_1$, ..., $\mathbf{P}_n$. och vill ha en kurva som interpolerar första och sista punkten och går i närheten av de andra.

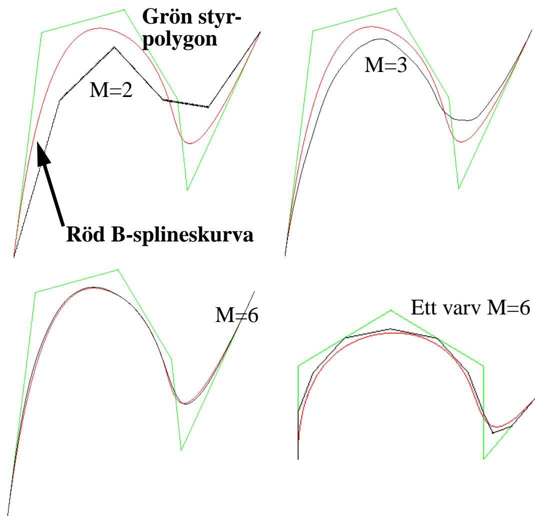
Mittpunktsmetoden innebär ett upprepande av följande steg:
1. Inför nya punkter $\mathbf{Q}_{2i+1}$, i =0, 1, ..., mittemellan de tidigare, dvs
   $\mathbf{Q}_{2i+1} = (\mathbf{P}_{i+1} + \mathbf{P}_i)/2$.
2. Modifiera de ursprungliga punkterna nu kallade $\mathbf{Q}_{2i}$, i = 0, 1, ..., enligt
   $\mathbf{Q}_{2i} = (\mathbf{P}_{i-1} + (M-2)\mathbf{P}_i + \mathbf{P}_{i+1})/M$
3. Låt **P** beteckna den nya punktvektorn **Q**.

För varje steg närmar sig punkterna alltmer en kurva (kan bevisas). Dock inte i allmänhet i ett ändligt antal steg.

De två stegen är typiska för alla uppdelningsmetoder. Först har man ett **förfiningssteg**, där nya punkter tillförs och sedan ett **utjämningssteg** som innebär att vissa punkter modifieras.

I figuren (med programmet `GL_BSPLINES_2005.c`) nedan har vi en kubisk B-splineskurva (röd) hörande till styrpunkterna (6 st) som är markerade med en (grön) styrpolygon. Vi har dessutom en svart kurva från mittpunktsmetoden med olika M-värden. I samtliga fall lika många varv i mittpunktsmetoden. För M=6 noterar vi att kurvan ser ut att sammanfalla med B-splines-kurvan. Att det är så kan bevisas (jag känner mig personligen inte helt övertygad om den visuella bevisningen i figuren).

Uppdelningsmetoder 3(7)



Längst ner till höger visas ett enda varv för metoden.

På adressen `http://www.multires.caltech.edu/teaching/demos/` hittar du Java-appletar som beskriver och demonstrerar två andra uppdelningsmetoder för kurvor: Chaikin (approximerande) och 4-punktsmetoden (interpolerande).

Uppdelningsmetoder 4(7)

Vi övergår nu till yt-fallet för vilket det finns en uppsjö metoder. Först en bild hämtad från SIGGRAPHS kurs 1998 om “Subdivision” (avsnittet *Subdivision Surfaces in Character Animation* av Tony DeRose m fl från Pixar Animation Studios) .

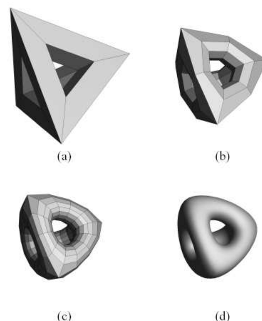


Figure 3: Recursive subdivision of a topologically complicated mesh: (a) the control mesh; (b) after one subdivision step; (c) after two subdivision steps; (d) the limit surface.

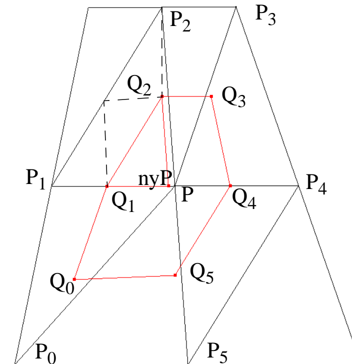
Metoden i figuren kallas *Catmull-Clark* och arbetar med fyrhörningar och är approximerande.

Låt oss se på en metod *Loop* (upphovsmannen heter Loop), som arbetar med trianglar i anslutning till följande figur.

Uppdelningsmetoder 5(7)

Förfiningssteget: För varje styrpunkt P inför ytterligare styrpunkter Q_i som bestäms av dels P, dels de punkter P_i som P är förbunden med. Om P är förbunden med 6 punkter:
$Q_i = (3P + 3P_i + P_{i-1} + P_{i+1})/8$, varvid $P_{-1} = P_5$, $P_6 = P_0$.

Utjämningssteget: Varje styrpunkt P (ej de nya) modifieras enligt (fallet 6 även nu)
$P = (10*P + P_0 + ... + P_5)/16$



Vi inser att specialåtgärder måste vidtas vid kanterna och när styrpunkterna inte har 6 förbundna. I figuren har vi med streckade linjer markerade delar av ett par trianglar som punkten P_2 ger upphov till.

Uppdelningsmetoder 6(7)

Ett exempel (kunde varit roligare om de boolska operationerna fungerat som de borde) med *Art Of Illusion*. Vi startar med en kub. Väljer den. Konverterar till triangelnät med **Object/Convert to Triangle Mesh**. Påbörjar redigering av kuben med **Object/Edit Object**, som visar upp ett nytt fönster med enbart vår kub. Väljer FACE längst ned till vänster i det fönstret. Markerar med MK3 de fyra trianglarna på ovansidan. Nu ser det ut som i figur 1. Gör **Mesh/Bevel/Extrude Selection** två gånger. Därefter väljer vi med **Mesh/Smoothing Method Approximate** (som - inbillar jag mig - ger en approximerande uppdelning; metoden anges ej). Nu enligt figur 2.

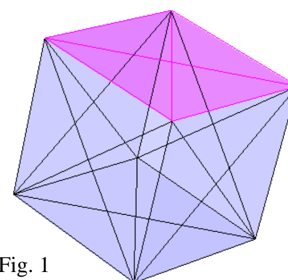


Fig. 1

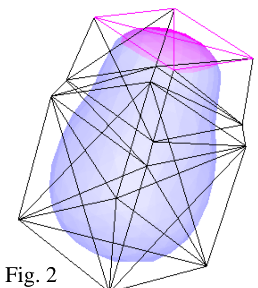
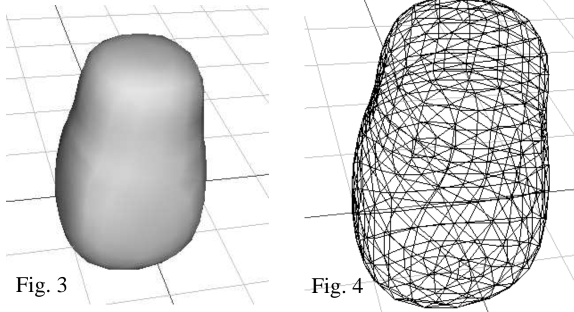


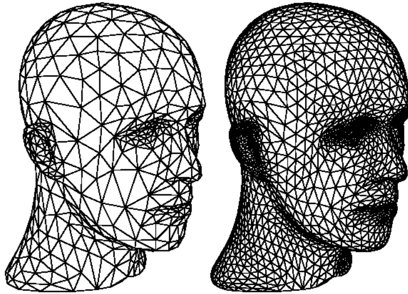
Fig. 2

Vi lämnar kubredigeringen och ser då i det vanliga fönstret något som liknar figur 3 eller figur 4 (beroende på visningssättet i **Scene/Display Mode**). Med **File/Export/VRML** (välj bort komprimeringen) får vi en massa triangeldata för det skapade objektet i en fil.

Uppdelningsmetoder 7(7)



Slutligen två steg i en interpolerande triangelmetod (bilden hämtad från SIGGRAPH-kursen)



P g a en miss kommer ytterligare material om kurvor och ytor med splines här.

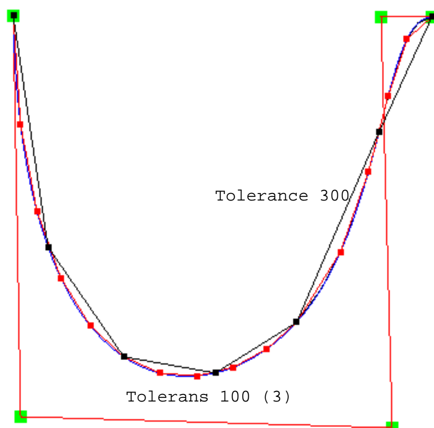
B-Splines i OpenGL: Rastreringsnoggrannhet

Denna och nästa OH (dvs OH130-131) mindre viktiga.

Rastreringsnoggrannheten (avståndet) kan styras med

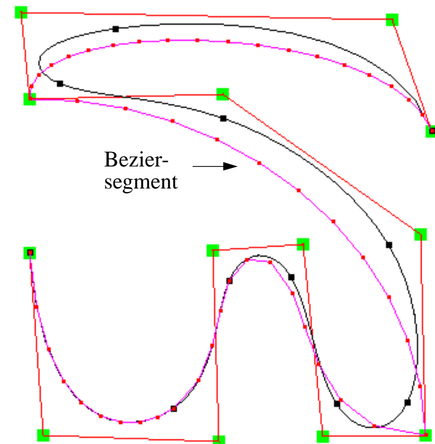
```
gluNurbsProperty(theNurb,
GLU_SAMPLING_TOLERANCE, värde);
```

Standardvärdet är 50. I figuren nedan har vi använt 100 och 300 (även 3, som i stort sett sammanfaller med 100). Värdena kan ha litet olika betydelser beroende på övriga inställda egenskaper, men normalt innebär de maximal kurvlängd räknad i bildpunkter. För 100 och 300 har vi satt ut **samplingspunkterna** (se även separat OH, ej med 2005).



B-Splines i OpenGL: Interpolation i inre punkter

Man låter tre skarvar sammanfalla (utan att ytterligare öka antalet skarvar). Har man successiva sådana trippelskarvar blir motsvarande segment en Bezier-approximation. I figuren nedan finns dels en vanlig B-splines-approximation (med sammanfallande skarvar bara i ändarna), dels en med två trippelskarvar på varandra. I det senare fallet har jag satt ut även samplingspunkterna vid kurvgenereringen.

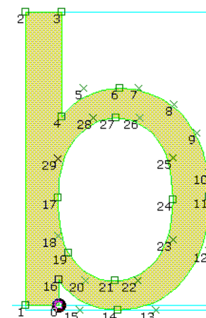


Typsnitt

Kurvapproximationer används inte bara i traditionellt ingenjörsarbete. En sentida tillämpning gäller typsnittsproduktion. Fram till 1980 fanns normalt bara ett typsnitt och i en enda storlek för dator-skärmar och skrivare och det låg i rasterform i ett läsminne (ROM). Mac hade när den kom 1984 åtskilliga typsnitt och i olika storlekare. Dessa var tillgängliga i rasterform. När man skulle skriva i en storlek som inte fanns färdig, gjordes omskalningar utifrån ett eller ett par befintliga storlekar. Resultatet blev i sådana fall halvdant. I laserskrivare med PostScript används i stället s k konturtypsnitt (eng. outline fonts). Varje tecken i ett typsnitt beskrivs i matematisk form en gång och oberoende av storleken. När en ny storlek behövs genereras rastret för tecknen i typsnittet. I PostScript (Type 1 och efterföljare) sker beskrivningen med kubisk Bezier-approximation. Av olika skäl ville Apple senare ha ett eget system. Företaget har därför konstruerat TrueType (TT), som bygger på kvadratiska B-splines (i själva verket kvadratiska Bezier). Detta har sålts till Microsoft, IBM m fl.

Vidareutvecklingar är TrueType GX (Apple), TrueType Open (Microsoft) och OpenType (Microsoft och Adobe). Liksom på andra områden slåss det vilt om herraväldet.

En referens: `http://www.truetype.demon.co.uk/`



Figur: Definitionspunkter och motsvarande kurva för bokstaven b i ett visst typsnitt. Den slutliga rastreringen bygger på omfattande ytterligare manipuleringar.

B-Splines/NURBS-ytor

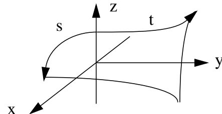
Formeln för en bi-B-Splineyta (vanligen utelämnas bi-) är :

$$P(u, v) = \begin{cases} \sum_{0 \le i \le m} \sum_{0 \le j \le n} B_i(u) B_j(v) P_{i,j} \\ 0 \le u \le m-2, 0 \le v \le n-2 \end{cases}$$

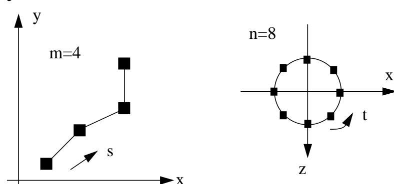
där $\mathbf{P}_{ij}$ är styrpunkterna och B_i är de vanliga basfunktionerna. Om m=n är antalet punkter lika i de båda "riktningarna" och då kan samma skarvvektor användas för båda parametrarna, annars behövs två olika. Vi är bara intresserade av det kubiska fallet.

Exempel på parametriseringar

1. Yta över en rektangel 0<x<A, 0<y<B



2. Rotationsyta



OpenGL: Ytor (kubiska B-Splines och NURBS)

OpenGL (snarare GLU) har stöd även för ytor.
OBS! Alla vektorer skall vara av float-typ (double duger inte).

- **Arbetsarea**:
  ```
  GLUnurbsObj* theNurb;    (Java: long theNurb)
  theNurb =  gluNewNurbsRenderer();
  ```
- **Styrpunkter**:(3->4 om NURBS)
  ```
  GLfloat styrpunkter[m][n][3];
  // Java: float styrpunkter[3*m*n];
  ```
- **Skarvar**:
  ```
  int sKnotCount = m+4, tKnotCount = n+4;
  GLfloat sKnots[sKnotCount] = {0,0,0,0,1,.. };
  GLfloat tKnots[tKnotCount] = {0,0,0,0,1,.. };
  ```
- **Vikter i NURBS-fallet**:
  ```
  GLfloat vikter[m][n];
  ```

Uppritning:

```
gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb, sKnotCount, sKnots,
           tKnotCount, tKnots
           n*3, 3, // avstånd i GLfloats mellan
                   // styrpunkter; 4 i NURBS-fallet
           styrpunkter,
           4, 4, // gradtalet+1
           GL_MAP2_VERTEX_3 // eller motsv
    );
gluEndSurface(theNurb);
```


Ytor i OpenGL: Ett manualblad

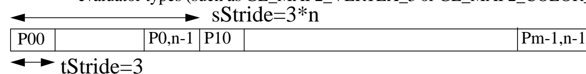
> man gluNurbsSurface

C SPECIFICATION

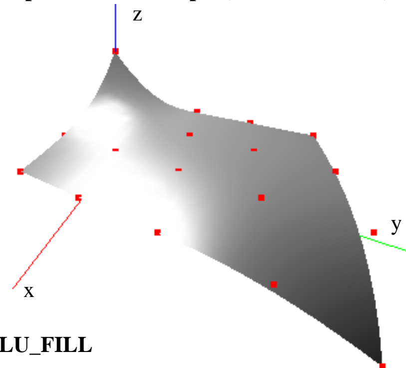
```
void gluNurbsSurface( GLUnurbs* nurb,
                GLint sKnotCount, GLfloat* sKnots,
                GLint tKnotCount, GLfloat* tKnots,
                GLint sStride, GLint tStride,
                GLfloat* control,
                GLint sOrder, GLint tOrder,
                  GLenum type )
```

PARAMETERS

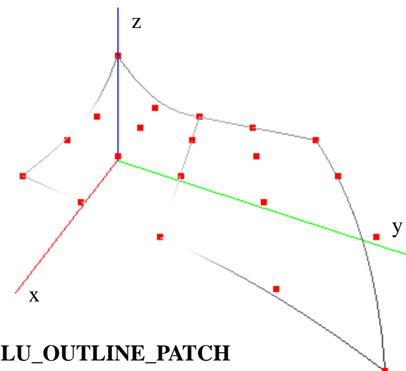
- **nurb** Specifies the NURBS object (created with gluNewNurbsRenderer).
- **sKnotCount** Specifies the number of knots in the parametric u direction.
- **sKnots** Specifies an array of sKnotCount nondecreasing knot values in the parametric u direction.
- **tKnotCount** Specifies the number of knots in the parametric v direction.
- **tKnots** Specifies an array of tKnotCount nondecreasing knot values in the parametric v direction.
- **sStride** Specifies the offset (as a number of single-precision floating point values) between successive control points in the parametric u direction in control.
- **tStride** Specifies the offset (in single-precision floating-point values) between successive control points in the parametric v direction in control.
- **control** Specifies an array containing control points for the NURBS surface. The offsets between successive control points in the parametric u and v directions are given by sStride and tStride.
- **sOrder** Specifies the order of the NURBS surface in the parametric u direction. The order is one more than the degree, hence a surface that is cubic in u has a u order of 4.
- **tOrder** Specifies the order of the NURBS surface in the parametric v direction. The order is one more than the degree, hence a surface that is cubic in v has a v order of 4.
- **type** Specifies type of the surface. type can be an of the valid two-dimensional evaluator types (such as GL_MAP2_VERTEX_3 or GL_MAP2_COLOR_4).



Ytor i OpenGL: Ett exempel (ex 21 modifierat) 1(2)



GLU_FILL



GLU_OUTLINE_PATCH

Ytor i OpenGL: Ett exempel (ex 21 modifierat) 2(2)

Styrpunkter (4 i x-led, 5 i y-led)

```
GLfloat c[4][5][3];

for (i=0; i<4; i++) {
       for(j=0; j<5; j++) {
            c[i][j][0] = i; c[i][j][1] = j;
            c[i][j][2]=2;
       }
}
c[0][0][2] = 3;  // Höj ytan vid origo
c[3][4][2] = 1.5; // Sänk diagonala hörnet
```

Skarvar

```
   GLfloat knotsu[8] = {0.0, 0.0, 0.0, 0.0,
                        1.0, 1.0, 1.0, 1.0};
   GLfloat knotsv[9] = {0.0, 0.0, 0.0, 0.0,
                      1.0, 2.0, 2.0, 2.0, 2.0};
```

Automatisk normalberäkning (fungerar för dessa ytor)

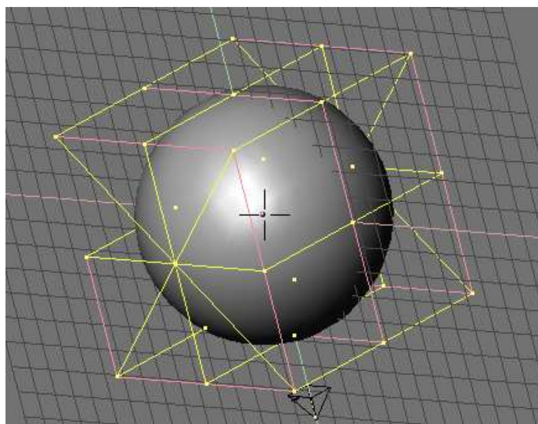
```
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
```

Rita

```
   gluBeginSurface(theNurb);
        gluNurbsSurface(theNurb, 8, knotsu, 9,
          knotsv, 5*3, 3, &c[0][0][0], 4, 4,
          GL_MAP2_VERTEX_3);
   gluEndSurface(theNurb);
```


NURBS-ytor i Blender

Välj **Add/Surface/Sphere**. Vi har tidigare nämnt att cirklar kan repre-



senteras exakt med NURBS. Motsvarande gäller en sfär i 3D. På bilden ser vi vilka styrpunkter som behövs (som i 2D, punkterna som verkar vara extra lyser bara igenom sfären).

Man kan “redigera” sfären genom att dra iväg med en styrpunkt (tangent G och sedan musen).

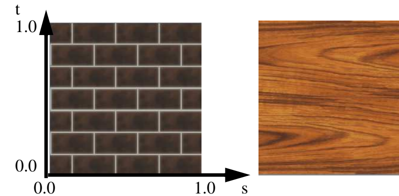
Tyvärr exporteras inga geometridata i detta fall när vi sparar (exporterar) på VRML- eller DXF-format (kollat i version 2.34).

Texturering 1(3)

Syfte: Att lägga ett mönster på en yta utan att geometriskt modellera de enskilda detaljerna.

Exempel: Gräsmatta, tegelvägg, träkloss, ...

Texturkarta:



I praktiken är texturkartan diskret, t ex 256x256 (men genereras kanske av någon procedur). De enskilda punkterna kallas **`texlar`** (eng. texel - texture element - i analogi med pixel).

Om ytan parametriserad: Säg att ytan kan beskrivas med $x = x(u,v)$, $y=y(u,v)$, $z=z(u,v)$ med $0 \le u,v \le n$. Gäller t ex NURBS-ytor och sfärer. Då kan vi t ex låta

$T(x,y,z) = T(u/n, v/n)$

eller

$T(x,y,z) = T(\mathrm{frac}(u),\mathrm{frac}(v))$,

där T med 2 parametrar beskriver texturkartan och T med 3 parametrar beskriver ytans textur, dvs vi låter i bästa datalogistil T beteckna två olika funktioner (polymorf). Detta är en rent matematisk texturering.

Texturering 2(3)

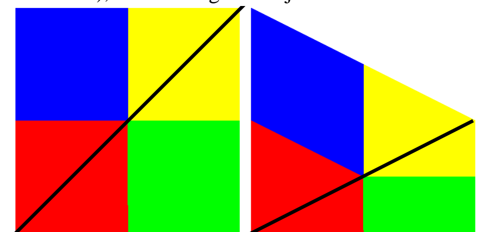
Exempel sfär med radie 1 och mittpunkt i origo: Kan beskrivas med

$$x = \cos\alpha \cdot \cos\theta$$
$$y = \sin\alpha \cdot \cos\theta \qquad 0 \le \alpha < 2\pi, -\pi/2 \le \theta < \pi/2$$
$$z = \sin\theta$$

Färgen i (x,y,z) på sfären hämtar vi från $(\alpha/2\pi,(\theta+\pi/2)/\pi)$ i texturen.

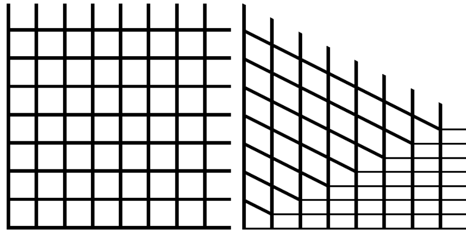
Det finns inte något entydigt sätt. I många fall vill man att påläggningen skall vara linjär (i världen), vilket ställer krav på ytan. Exempelvis kan en triangel avbildas linjärt på en annan triangel och en rektangel på en annan rektangel, men det gäller i allmänhet inte för fyrhörningar. En kvadrat kan inte avbildas linjärt på en sfär (omvändningen av kartritarnas problem), men däremot på en torus.

Exempel: Texturen till vänster avbildas på en kvadrat med hörnen (0,0), (1,0), (1, 0.5), (0,1). Man ser att det blir deformationer. Praktiskt har systemet avbildat två trianglar på två andra trianglar (de svarta dragna i efterhand), vilket kan göras linjärt.

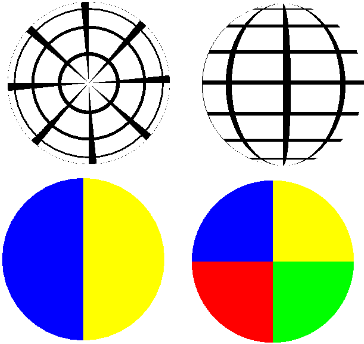


På nästa OH visas samma avbildning med en annan textur.

Texturering 3(3)



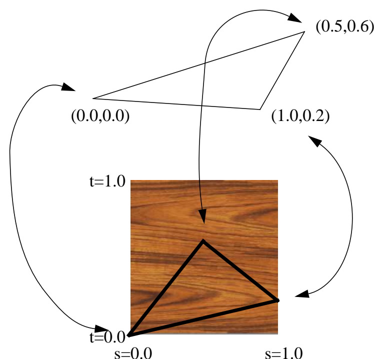
Exempel: De följande figurerna visar hur de tidigare texturerna kan se ut när de lagts på en sfär (till vänster ser vi längs z-axeln; till höger längs y-axeln; gjord med modifierad version av *glutSolidSphere* och ortoprojektion)



Texturer har visat sig vara ett kraftfullt verktyg för andra ändamål, vilket vi tar upp senare i kursen.

Texturering i grafiksystem

I grafiksystemen: Varje polygon (motsv) i världen förses med texturkoordinater i hörnen, ungefär som när vi lägger på färger eller normaler i hörnen. T ex



med innebörden att den markerade triangeln i texturkartan skall avbildas på vår triangel. Vid rastreringen (bildpunkt för bildpunkt) interpolerar (jfr nedan) man fram texturkoordinater för bildpunkten (ungefär som när man vid Gouraudtoning interpolerar fram en intensitet utifrån vad som beräknats i hörnen) och hämtar information från en (eller flera) motsvarande texel i texturkartan inför färgläggningen. Rastreringen sköts i sin helhet av grafikprocessorn och textureringen kostar inget eller obetydligt extra. Texturkartan lagras med fördel i grafikkortets texturminne.

Observera dock att interpolationen inte kan ske linjärt (om perspektiv används), se pappret "Från värld till skärm", avsnitt 10.

Texturering i OpenGL

Texturkartan: Typiskt ett rutnät med MxM punkter (kan dock vara okvadratiskt) punkter. M skulle före version 2.0 vara en 2-potens, dvs $M = 2^N$, men nu kan M vara godtyckligt. I praktiken finns någon begränsning på M, t ex att M är högst 256. Varje punkt innehåller typiskt ett RGB-värde (eller RGBA-värde; varianter finns). Som exempel en 64x64-textur

```
#define TexHeight 64
#define TexWidth 64
GLubyte Image[TexHeight][TexWidth][3];
```

I en matris av denna typ kan vi lägga tal mellan 0 och 255, varvid 255 betyder högst intensitet. Alternativt typen GLbyte med tal mellan 0 och 127.

Initieringar

Ge texturkartan innehåll: Läs från fil (konvertera vid behov) eller hårdkoda. T ex en gul texel:

```
Image[i][j][0] = (GLubyte)255;
Image[i][j][1] = (GLubyte)255;
Image[i][j][2] = (GLubyte)0;
```

Skapa ett texturregister: Ett register för 2 texturer

```
GLuint texName[2];
glGenTextures(2, texName);
```

Vi refererar i fortsättningen till en viss textur med `texName[i]`, som i själva verket är ett heltal och kallas texturens namn. Bara en i taget är aktiv och vi gör en viss textur aktiv med

```
glBindTexture(GL_TEXTURE_2D, texName[i]);
```

Lägg in en viss textur i texturregistret: Vi skapar ett texturobjekt utan vidare beroende av den tidigare texturkartan och kopplar det till ett visst namn.

```
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, TexWidth,
  TexHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Image);
//GL_RGB för RGB-värden och GL_RGBA för RGBA.
//Nollorna i glTexImage2D har ej med index 0 att göra
```


Texturering i OpenGL, forts

Sätt egenskaper: Filtreringsegenskaperna gäller per textur, men påläggningssättet gäller till dess det ändras, dvs om det skall vara olika måste det sättas vid uppritningen

```
// Filtreringsegenskaper vid förstoring respektive
// förminskning
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER
                ,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER
                ,GL_NEAREST);
// Påläggningssätt: GL_DECAL och GL_REPLACE skriver
// över, GL_MODULATE och GL_BLEND påverkas av bl a
// framräknad belysningsfärg
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
         GL_DECAL);
```

Vid uppritning

Slå på texturering:

```
glEnable(GL_TEXTURE_2D)
```

Välj textur:

```
glBindTexture(GL_TEXTURE_2D, texName[0]);
```

Ange texturkoordinat för varje hörn:

```
glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0, 0.0)
```

Anm 1. För vissa färdiga objekt (t ex glutSolidTeapot men inte för glutSolidCube och glutSolidSphere görs det utanför vår kontroll). Det finns också möjlighet att generera texturkoordinaterna utifrån koordinaterna i världs- eller vykoordinatsystemet.

Anm 2. Om en texturkoordinat är större än 1 upprepas (kan ändras till annat) texturen. Vi kan därigenom klä t ex en stor husvägg med en mindre textur.

Exempel: Se OpenGL-häftet, avsnitt 23. Utbyggt som GL_TEXTUR2004.c

Andra texturer: I OpenGL finns även 1D-texturer och 3D-texturer (v 1.2).

Texturering i JOGL (Ex. 18)

De viktigaste ändringarna beror på

- att matriselement i Java inte säkert lagras i sekvens (vilket OpenGL antar)
- att Java saknar motsvarighet till `GLubyte` (C:s `unsigned char`) för tal 0-255.

Vi får därför använda en vanlig vektor respektive datatypen **`byte`** för tal -128 till 127. Härvid betyder 0 0.0 och 127 1.0 (i min version av JOGL verkar det som om -128 också blir 0.0 och -1 blir 1.0, men det är nog en bugg; alla negativa tal borde bli 0.0).

Variabler i *MyGLEventListener*:

```
private const int TexWidth= 4;
private const int TexHeight=4;
private byte[] Image = new byte[TexHeight*Tex-
Width*3];
int[] texName = new int[2];
```

Initieringarna görs i *init*. I stället för t ex

```
Image[i,j,1] = (GLubyte)255;
```

får vi skriva

```
Image[i*TexWidth*3+j*3+1] =  127;
```

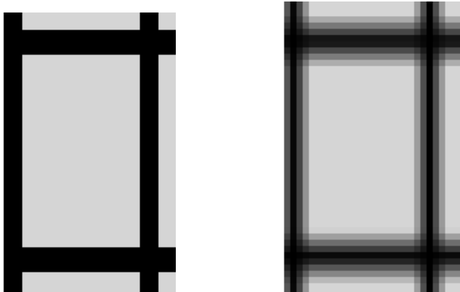
Slutligen byter vi i anropet av *glTexImage* den parameter beskriver texelns lagring i *Image* `GL_UNSIGNED_BYTE` mot GL_BYTE, dvs

```
gl.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGB,
      TexWidth, TexHeight, 0, GL.GL_RGB, GL.GL_BYTE, Image);
```


Texturering - förstoring 1(2)

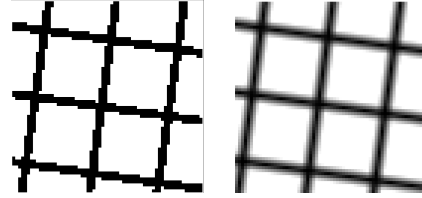
Problem: Till varje bildpunkt hör texturkoordinatvärden (t ex via mittpunkten). Det verkar då naturligt att som texel välja den som i någon mening ligger närmast dessa värden. Detta är också vad som sker då man har filtreringsegenskapen GL_NEAREST. Men vid uppförstoring, dvs då en texel berör flera bildpunkter (inträffar då **betraktaren är nära den texturerade ytan**) kan det skapa oönskad taggighet. Det gäller speciellt om texturerna utgörs av bilder med kraftig kontrast. Med GL_LINEAR kommer i stället en linjär interpolation med hjälp av fyra texlar intill den närmsta att äga rum, vilket ger en utjämnad bild.

Exempel: Vi har en rutnätstextur där varje linje är en texel bred. Vi lägger den på en yta så att varje texel täcker 4x4 bildpunkter. Till vänster visas resultatet med resultatet GL_NEAREST, till höger med GL_LINEAR.



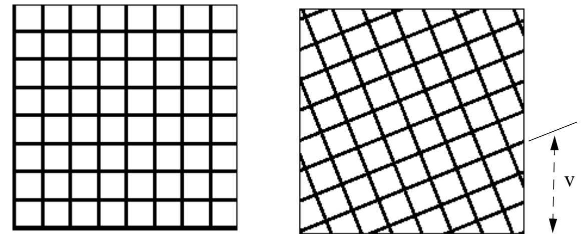
Texturering - förstoring 2(2)

Kanske tycker du att den vänstra bilden (dvs GL_NEAREST) ser bäst ut. Nyttan med GL_LINEAR i förhållande till GL_NEAREST ser vi om linjerna inte är axelparallella



I de flesta fall är en utjämning av typen till höger mest tilltalande för ögat, men det finns situationer då man vill bevara de skarpa övergångarna. För krökta ytor är det inte självklart hur matematiken bakom ser ut. Vi nöjer oss med att konstatera att det brukar fungera bra.

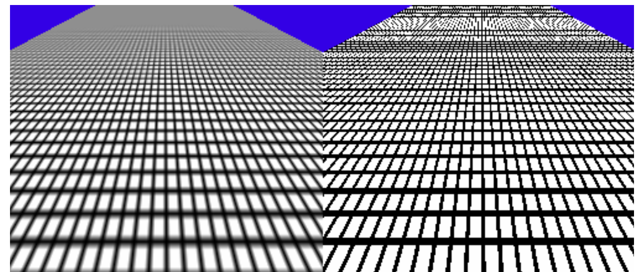
Texturering - placering av texturen



I högra figuren har vi vridit texturen vinkeln v innan den lagts på kvadraten, vilket t ex kan ske med andra texturkoordinater.

Texturering - förminskning

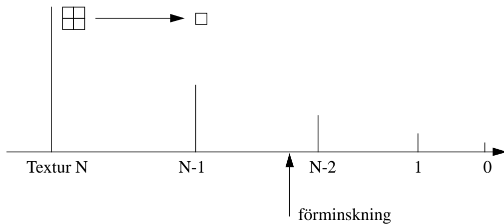
Problem: Vid förminskning kommer flera texlar att höra till samma bildpunkt. Detta inträffar **när vi tittar på en texturerad yta på stort avstånd**. Vilken texel skall då få bestämma? Vi kan ta den närmsta (GL_NEAREST), men detta blir i verkligheten ett närmast slumpmässigt val och skapar brister, speciellt om användaren rör sig i en scen. GL_LINEAR reducerar problemet bara en aning. Om bildpunkten täcker åtskilliga texlar, så är det enda vettiga att bilda ett medelvärde av alla dessas värden, men det blir kostsamt. I högra delen av figuren, ser vi ett fall utan att några åtgärder vidtagits. Till vänster har man använt s k mipmapping, som ger ett för ögat mycket mera tilltalande resultat. Bilden ritad med en modifierad variant av programmet `mipmap.c` i GLUT-distributionen.



Demo: Från S3 (öppna filen /users/course/TDA360/S3/index.htm med någon webbläsare; det sista exemplet visar mipmaps).

Mipmapping

Man bildar en hel följd av texturkartor med utgångspunkt från den ursprungliga. Om den första har dimensionen MxM ($M=2^N$) så har nästa dimensionen M/2xM/2 osv. Den sista består av en enda texel. Totalt har vi N+1 texturer. Vid halveringen slås fyra texlar ihop till en ny med medelvärdesbildning.



Vid användningen av texturkartorna räknas det fram ett ungefärligt mått på förminskningen och man går in mellan de omgivande texturkartorna och gör (bi)linjär interpolation dels inom dessa, dels mellan dem. Benämningen trilinjär interpolation står för detta.

Ordet mip kommer av latinets "Multum in parvo", som betyder mycket på ett litet utrymme. Utrymmesmässigt krävs ungefär 33 % mer än utan mipmapping.

OpenGL (enklast via GLU) stödjer mipmapping. Och de förändringar som behövs är små. Anropet av glTexImage2D ersätts av

```
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB,
    TexWidth, TexHeight, GL_RGB, GL_UNSIGNED_BYTE,
    Image);
```

och filtreringsegenskapen vid förminskning sätts med

```
glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR);
```

Det är allt! (Det finns ett par andra varianter.)

Tekannan och andra färdiga GLUT-objekt 1(2)

Litet inkonsekvent! Dessa kan delas in i plana (*glutSolidCube* m fl), andragradsytor (*glutSolidSphere* m fl) och allmänt krökta (*glutSolidTeapot*). De senare är baserade på B-splines el dyl, varigenom normaler och texturkoordinater kan genereras (med lämpliga tillstånd). Andragradsytorna är baserade på Quadrics-objekt, som vi inte tagit upp här.

	glutSolidCube &Co, glutSolidTorus &Co	**glutSolidSphere &Co**	**glutSolidTeapot &Co (FINNS EJ I JOGL nu)**
Normaler	Ja	Ja	Ja (GL_AUTO_NORMAL satt)
Texturkoordinater	Nej (man kan med möda ändra i källkoden)	Nej (man kan lätt ändra i källkoden)	Ja, lämpligt tillstånd är satt i koden
Belysning	Ja	Ja	Ja, om vi ändrar på begreppet framsida, dvs om `glFaceMode(GL_CW)`, det normala är GL_CCW.

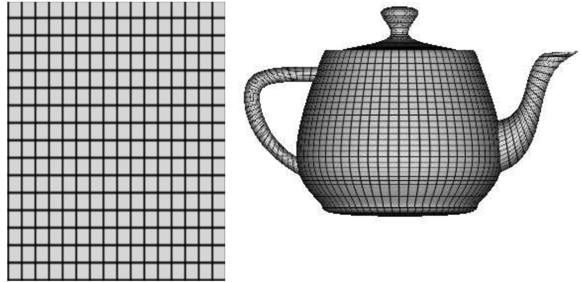
Källkoden till *glutSolidSphere* modifierad så att texturkoordinater genereras

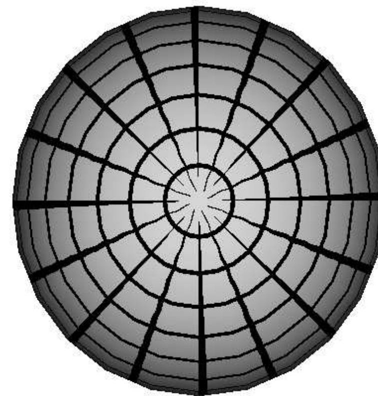
```
void glutSolidSphereMB(GLdouble radius, GLint slices,
                       GLint stacks) {
   gluQuadricDrawStyle(quadObj, GLU_FILL);
   gluQuadricNormals(quadObj, GLU_SMOOTH);
   gluQuadricTexture(quadObj,GL_TRUE); // default GL_FALSE
   gluSphere(quadObj, radius, slices, stacks);
}
```

Man måste ha deklarerat `static GLUquadric *quadObj;` och före användning initierat `quadObj = gluNewQuadric();`

Tekannan och andra färdiga GLUT-objekt 2(2)

En rutnätstextur pålagd en kvadrat, en tekanna och en sfär ritad med den modifierade *glutSolidSphereMB*.





I z-led (uppifrån)
Enl källkoden till gluSphere:

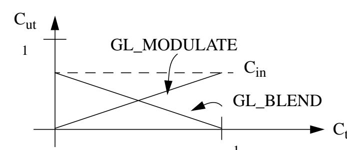
If texturing is turned on (with gluQuadricTexture), then texture coordinates are generated so that t ranges from 0.0 at z=-radius to 1.0 at z=radius (t increases linearly along longitudinal lines), and s ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

Texturer och ljus

I OpenGL ligger ljusberäkningen före texturpåläggningen, vilket är naturligt eftersom beräkningen gäller hörn och inte bildpunkter, men det är inte invändningsfritt. Det finns fyra storheter som påverkar hur ljusvärdena och texturvärdena kombineras. Vi skall bara beskriva dem i huvuddrag.

Vi har nämnt att GL_DECAL och GL_REPLACE bara lägger på den aktuella texelns färg och alltså struntar i belysningsvärden. Vårt hopp står då till GL_MODULATE och GL_BLEND. Låt C stå för någon av de tre grundfärgerna R, G och B. Låt C_{in} vara värde enligt belysningsmodellen, C_t texelns intensitet och C_{ut} resultatet. Då gäller för (viss skillnad om RGBA)

GL_MODULATE: $C_{ut} = C_{in}C_t$
GL_BLEND: $C_{ut} = C_{in}(1-C_t)$ (egentligen $C_{ut} = C_{in}(1-C_t)+C_tC_c$, där C_c är förvald till 0, men kan ändras med *glTexEnv*)



Eftersom alla intensiteter ligger mellan 0 och 1, kommer texturen att förmörkas vid användning av GL_MODULATE (om inte $C_{in} = 1$). Likaledes förmörkas ljusvärdet (om inte $C_t=1$). Men det värdet måste ändå bli vår favorit. Det hade varit önskvärt med fler kombinationsmöjligheter (jämför med andra kommande situationer).

Automatisk texturkoordinatgenerering

I princip har vi hittills anget texturkoordinater med någon version av *glTexCoord*. Men det finns också några mer eller mindre väl uttänkta sätt att få texturkoordinaterna automatgenererade. Den första texturkoordinaten kallas i OpenGL *s* och den andra *t* (det finns dessutom *r* och *q*). Man måste då

1. Tala om hur respektive texturkoordinat skall genereras
2. Aktivera genereringen av respektive texturkoordinat. Eventuellt explicit angivna texturkoordinater ignoreras.

Hur respektive texturkoordinat skall genereras

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR)
          GL_T,                     GL_EYE_LINEAR
          GL_R                      GL_SPHERE_MAP
                                    GL_NORMAL_MAP_ARB
                                    GL_REFLECTION_MAP_ARB
```

De två sista alternativen för tredje parametern är utvidgningar som införts i OpenGL 1.3 och som fungerar på grafikkort med bl a GeForce-processorer och nu även Sun. De tre sista (beskrivs i detalj senare) kräver att man genererar tvådimensionella texturkoordinater (`GL_S` och `GL_T`), medan för de två första det räcker med endimensionella (`GL_S`). I dessa två fall blir texturkoordinaten avståndet från den aktuella punkten till ett plan angivet antingen i modellkoordinater (`GL_OBJECT_LINEAR`) eller vykoordinater (`GL_EYE_LINEAR`).

Aktivering

T ex av s-koordinaten: `glEnable(GL_TEXTURE_GEN_S)`
Dessutom måste vi ha aktiverat texturering med `glEnable(GL_TEXTURE_1D)` (bara *s*) eller `glEnable(GL_TEXTURE_2D)` (*s* och *t*)

Avståndsstyrd texturkoordinatgenerering 1(2)

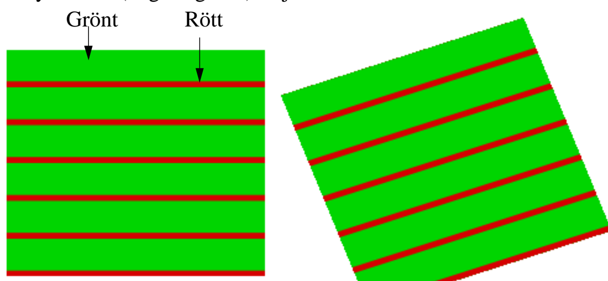
Det handlar om `GL_OBJECT_LINEAR` respektive `GL_EYE_LINEAR`.

Planet beskrivs med ett anrop av formen
`glTexGenfv(GL_S, GL_OBJECT_PLANE,vektorvariabel)`
respektive
`glTexGenfv(GL_S, GL_EYE_PLANE,vektorvariabel)`
där vektorvariabeln innehåller de fyra koefficienterna i planets ekvation F(x,y,z) = Ax+By+Cz+D = 0. Som vanligt ger F(x,y,z) avståndet från (x,y,z) till planet om normalen (A,B,C) är normerad.

Exempel: Vi har en 1D-textur bestående av 32 texlar, där de första fyra är röda och resterande är gröna. Vi lägger den på en fyrkant vinkelrät mot planet y=0. Vi låter texturkoordinaten genereras som avståndet till planet y = 0 i **modellkoordinatsystemet**, dvs

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGenfv(GL_S, GL_OBJECT_PLANE,plane);
```

med `static GLfloat plane[] = {0.0, 1.0, 0.0, 0.0};`. Om vi vrider fyrkanten (högra figuren) följer texturen som väntat modellen.

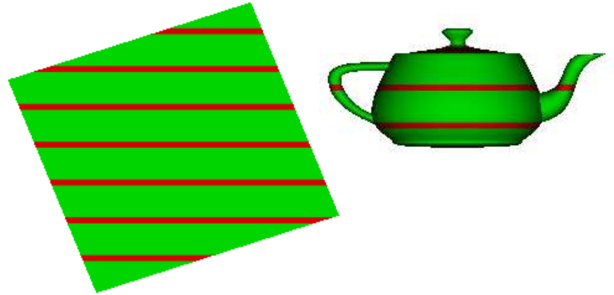


Avståndsstyrd texturkoordinatgenerering 2(2)

Om vi i stället låter texturkoordinaten genereras som avståndet till planet y = 0 i **vykoordinatsystemet**, dvs

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,GL_EYE_LINEAR);
glTexGenfv(GL_S, GL_EYE_PLANE,plane);
```

med `plane` som förut, så ligger texturen fast i vykoordinatsystemet och följer inte modellen (vänstra figuren nedan).



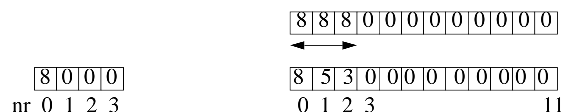
Man kan naturligtvis lika gärna lägga texturen på ett allmännare objekt, t ex den välkända tekannan (högra figuren).

Texturkoordinaterna genereras efter transformationen till vykoordinater. I fallet `GL_OBJECT_LINEAR`/`GL_OBJECT_PLANE` transformerar därför OpenGL koefficienterna till vykoordinatsystemet på det sätt som vi beskrivit i "Från värld ...", avsnitt 8.

1D-textur

En sådan är

Låt oss för att en aning illustrera den matematiska skillnaden mellan `GL_NEAREST` och `GL_LINEAR` anta att svart har värdet 8 och vitt värdet 0 (det riktigare är ju 0 och 255 eller 0.0 och 1.0). Då utgörs texturen av vänstra figuren nedan. Om vi applicerar den på en horisontell linje med 12 bildpunkter så skulle `GL_NEAREST` ge resultatet överst till höger, medan `GL_LINEAR` ger ungefär undre figuren.



Matematiken bakom skulle kunna vara som följer. Om antalet texlar i texturen är T och antalet bildpunkter är B, så har vi följande samband mellan texelnummer *s* och bildpunktsnummer *x*:
s = Tx/B, dvs i figurens fall s = x/3. Om vi betraktar *s* som ett reellt tal s = i + a, där *i* är heltalsdelen och *a* bråkdelen, har vi följande möjliga medelvärdesbildning (t_i = intensitet i texel *i*)
intensitet i bildpunkt x = $(1-a)t_i + at_{i+1}$.
som ger värdena i figuren.

x	s	i	a	intensitet
0		0	0	8
1		0	1/3	16/3≈ 5
2		0	2/3	8/3≈3

Textur från fil 1(4)

I allmänhet läser man väl texturkartan från en fil. Om filen bara skall uppfattas som en följd av bytes (vilket gäller t ex textfiler) är det lätt att skriva en läsprocedur. T ex för det byte-baserade formatet PPM P6 (`$DG/EXEMPEL_MB/ReadPPM_P6_Texture.c`)

```
GLubyte* ReadTexture(char* filename, int* width,
                     int* height) {
   int i,j, w, h; GLubyte *ptr, *start;
   // Open file
   FILE *texfile = fopen(filename,"r");
   if (!texfile) { printf("No such file\n"); exit(1)}
   // Ignore first row P6
   for (i=1; i<=3; i++) getc(texfile);
   // Read width and height
   fscanf(texfile, "%d %d", &w, &h);
   *width = w; *height = h;
   ptr = malloc(w*h*3); start = ptr;
   // Ignore last line
   for (i=1; i<=5; i++) getc(texfile);
   // Read the texture
   for (i=0; i<h; i++){
        for (j=0; j<w; j++){
             *ptr=(GLubyte)getc(texfile); ptr++;
             *ptr=(GLubyte)getc(texfile); ptr++;
             *ptr=(GLubyte)getc(texfile); ptr++;
        }
   }
   printf("Texturen läst\n") fclose(texfile);
   return start;
}
```

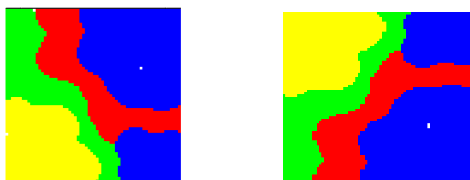

Textur från fil 2(4)

PPM-formatet (Portable PixMap) utgörs av en följd av bytes, som om texturen är 256x256 inleds med 15 bytes (tecken) (\n är ett tecken, tecknet för nyrad)
`P6\n256 256\n255\n`
varpå följer 3 bytes med RGB för översta vänstra punkten och sedan motsvarande för övriga punkter rad för rad. Första 256 avser bildbredden, andra bildhöjden. Avslutande 255 anger maximalt färgvärde. Det kan finnas varianter (t ex lägger *xv* in en kommentarrad om cirka 60 tecken efter P6-raden. Du kan titta på inledningen av en PPM-fil med ett vanligt redigeringspreogram. Inleds filen med P4 eller P5, så är det fråga om variantformat PBM resp PGM.

Funktionen *ReadTexture* används typiskt så här

```
GLubyte* Image;
GLint texwidth, texh;
Image = ReadTexture("brick.ppm", &texwidth, &texh);
printf("Texturen %d x %d\n", texwidth, texh);
```

Texturer lagras nedifrån i OpenGL, dvs en textur som i bildbehandlingsprogram som *xpaint* eller *xv* ser ut som till vänster blir med vår funktion upp och nedvänd som till höger.



Detta kan vi lätt bota, vilket vi visar i en motsvarande Java-metod.

Textur från fil 3(4)

För enkelhets skull antar jag att texturstorleken är känd och att vi har en global variabel *Image* där texturen skall placeras.

```
void readImage(String filnamn) {
   try {
      FileInputStream fin = new FileInputStream(filnamn);
      int i, p = 0, j, r = 1;
      byte b;
      // Assuming size nxn with 10<n<100
      // Skips the 13 characters P6\n64 64\n255\n
      for (j=1; j<=13; j++) { i = fin.read(); b = (byte)i; }
          while ((i = fin.read()) != -1) {
              b = (byte)(i/2);
              Image[(TexHeight-r)*TexWidth*3+p]= b; p = p+1;
              if (p==3*TexWidth) { p = 0; r=r+1;}
              //Image[p] = b; p = p+1;
          }
       }
       catch (FileNotFoundException e) { System.exit(1); }
       catch (IOException e) { System.exit(1); }
}
```

Javas `(byte)i` omvandlar heltal enligt

`i`	`0`	`127`	`128`	`255`
`(byte)i`	`0`	`127`	`-128`	`-1`

vilket gör att såväl 255 som 127 ger full intensitet (åtminstone i JOGL enl tidigare), vilket naturligtvis inte är så lyckat. Därför `i/2` i koden.

Textur från fil 4(4)

Det finns många olika sätt - format - att lagra en bild som t ex en texturkarta på. Vi återkommer till det i slutet av kursen. Bildbehandlingsprogrammet *xv* (under Linux/UNIX) kan användas för att konvertera mellan de vanligaste. *GIMP* (för Linux/UNIX eller PC) klarar också det. Formatet `.ppm` (Portable PixMap) är transportabelt, dvs kan läsas med program skrivet i datoroberoende källkod (t ex med den ovan).
I GLUT-distributionen ingår en procedur (finns i filen `progs/advanced/texture.c`) för läsning av `.rgb`- och `.bw`-format, vilka härrör från Silicon Graphics. Det senare enbart för svartvitt.

```
unsigned * read_texture(char *name,
          int *width, int *height, int *comps)
```

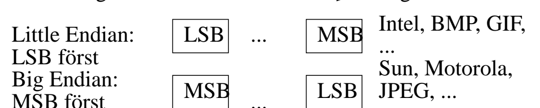
Koden är utformad så att den fungerar oberoende av dator. Typisk användning

```
int texwidth, texheight, texcomps; GLubyte *image;
image=read_texture("brick.rgb",
                 &texwidth,&texheight,&texcomps);
```

Vi får reda på texturens bredd och höjd, liksom antalet bytes per texel (3=GL_RGB, 4=GL_RGBA). Behöver alltså inte införa någon matris! Även ett antal bilder på något av dessa två format medföljer.

Innehåller filen heltal eller flyttal på binärform blir det besvärligare åtminstone om man vill ha datoroberoende kod. Förklaringen är att olika datorer/format lagrar bytarna i tal i olika ordning (Sun: Big Endian, PC: Little Endian). Kod på nätet är oftast utformad för PC-miljön.

Låg adress → Högre adress



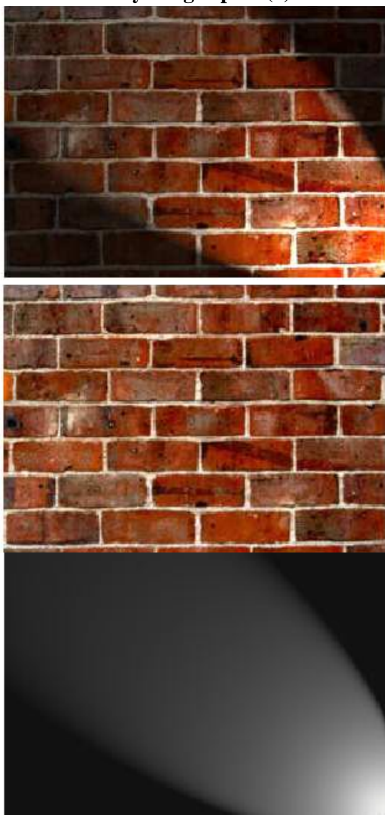
Belysning i spel 1(3)

Säg att vi belyser en tegelvägg med en lampa. Då vill vi ha ett resultat liknande det i översta figuren på nästa OH (hämtad från NVIDIAs promotionsmaterial). Kan vi få det med OpenGL? I första ansatsen räknar man ut ljusvärden i väggens fyra hörn. Även om dessa skulle bli de korrekta, så ger en efterföljande interpolering (Gouraud) ett föga realistiskt resultat. För att få något bättre måste vi dela in väggen i flera mindre fyrkanter och göra ljusberäkning för var och en av dessa. Finns något annat sätt?

Inspirerade av framgångarna med texturer införde man i spelet Quake **ljuskartor** (eng. light map), som är texturkartor med ljusmönster. Längst ned på nästa OH hittar vi en sådan. Ljuskartan byggs upp i förväg (kan vara handtillverkad eller framräknad med stor omsorg). Vid ritningen kombineras ljuskartan med tegeltexturen (texel för texel och därmed pixel för pixel), se andra figuren, innan ritning sker. Med denna teknik kan övertygande effekter åstadkommas. I figurens fall har kartvärdena multiplicerats med varandra, men man kan också tänka sig operationer som + eller någon allmän blandning.

Detta har i sin tur lett till ett begrepp som **multi-texturering**, som innebär att den slutliga texturen bestäms av ett antal olika. För att få snabbhet har de senare grafikkorten allt bättre hårdvara för arbete med sådan texturering. Man har också infört ytterligare möjligheter till automatisk texturgenerering som gör att man med få ljuskartor även kan hantera dynamiska lampor. NVIDIA talar nu om per-pixel-lighting i ställer per-vertex-lighting som är det vanliga i OpenGL. Och nog har man lyckats skapa många imponerande ting. Men det krävs att man har ett grafikkort som stöder dessa utvidgningar av OpenGL.

Belysning i spel 2(3)



Belysning i spel 3(3)

Har man sagt A och B, skall man kanske även säga C även om jag går händelserna i förväg. Här följer det gamla sättet att lägga på ljuskartor.

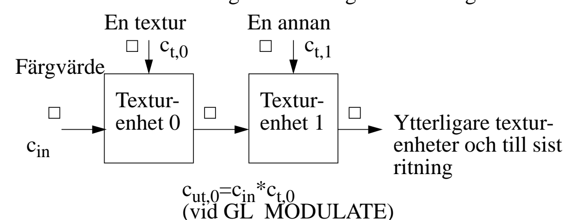
```
// Först ritar vi kvadraten med tegel utan ljus
// Talet 5 ger repetition av texturen
glBindTexture(GL_TEXTURE_2D, texName[tegelnr]);
glColor3f(1.0, 1.0, 1.0); //Bara om vi använder GL_MODULATE,
                          //lägrevärden skulle förmörka
glBegin(GL_QUADS);
     glTexCoord2f(0, 0); glVertex2f(-1, -1);
     glTexCoord2f(5, 0); glVertex2f(1, -1);
     glTexCoord2f(5, 5); glVertex2f(1,  1);
     glTexCoordf(0, 5);  glVertex2f(-1,  1);
glEnd();

// Sedan är det dags att byta till ljuskartan och se till att
// värdena blandas på lämpligt sätt med det redan ritade
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO, GL_SRC_COLOR); // Ger tegel*ljuskarta
glDepthFunc(GL_LEQUAL); // Ser till att det ritas även om
     // vi redan ritat på just det avståndet; finns
     // andra sätt
glBindTexture(GL_TEXTURE_2D, texName[ljusnr]);
// Plats för extranummer
//Och så ritar vi kvadraten med enbart ljuskartan
glColor3f(1.0, 1.0, 1.0); // Onödig upprepning
glBegin(GL_QUADS);
     glTexCoord2f(0, 0); glVertex2f(-1, -1);
     glTexCoord2f(1, 0); glVertex2f(1, -1);
     glTexCoord2f(1, 1); glVertex2f(1,  1);
     glTexCoord2f(0, 1); glVertex2f(-1,  1);
glEnd();
```

Extranummer(låter oss flytta ljustexturen):

```
glMatrixMode(GL_TEXTURE);glPushMatrix();glTranslatef(tx, ty, tz);
... och efter glEnd(): glPopMatrix(); glMatrixMode(GL_MODEL_VIEW);
```


Multitexturering 1(2)

Detta är ett exempel på en utvidgning som kommit in i specifikationen för OpenGL 1.3 (aug 2001). Från våren 2004 kan vi använda det på våra SUN-ar. Syftet är att kunna använda flera texturer samtidigt utan att som vi tidigare (OH om belysning i spel) gjorde skicka samma geometri flera gånger. Man har infört ett antal texturenheter (minst två) numrerade 0 och uppåt, som har sina egna egenskaper (eget påläggningssätt, egen texturmatris, eget igång-tillstånd etc). Dessa verkar i följd, vilket med modern teknologi och lämpligt matningssätt innebär att de inte kostar något tidsmässigt. Så här fungerar det:



Den tidigare koden (se OH 163) förenklas. Vi placerar geometrihanteringen i en separat procedur `wall` (suffixen `ARB` och `_ARB` kan tas bort).

```
void wall() {
    glBegin(GL_QUADS);
      // Första hörnet, ett texturkoordinatpar för
      // tegelväggen (texturenhet 0), ett annat för
      // ljuskartan (texturenhet 1)
      glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0, 0);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0, 0);
      glVertex2f(-1, -1);
```

Multitexturering 2(2)

```
        // Övriga hörn, 5 ger upprepning av tegelmönstret
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,5, 0);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1, 0);
        glVertex2f(1, -1);
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,5, 5);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1, 1);
        glVertex2f(1,  1);
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0, 5);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0, 1);
        glVertex2f(-1,  1);
    glEnd();

}
```

Och i omritningsproceduren `display()`:

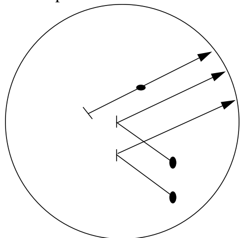
```
  // Med glActiveTexture bestämmer vi vilken texturenhet
  // vars tillstånd skall förändras
  // Först texturenhet 0 som får ha hand om tegelmönstret
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glBindTexture(GL_TEXTURE_2D, texName[tegelnummer]);
  glEnable(GL_TEXTURE_2D);
  // Sedan texturenhet 1 som hanterar ljuskartan
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
  glBindTexture(GL_TEXTURE_2D, texName[ljusnr]);
  glEnable(GL_TEXTURE_2D);
  // Extranumret
  glMatrixMode(GL_TEXTURE);
  glPushMatrix();
  glTranslatef(tx, ty, tz);
  wall();
  glPopMatrix(); // texturmatrisen
  glDisable(GL_TEXTURE_2D);
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glDisable(GL_TEXTURE_2D);
```


Environment mapping (omgivningsavbildning)

Syfte: Återge omgivningens reflektion i ett reflekterande föremål utan att ta till strålföljning.

Metod: Skapa en projicerad platt (2D) bild av omgivningen. Lagra den som en textur. När vi vill rita objektet så låter vi reflektionsvektorn ge upphov till texturkoordinater och applicerar texturen (som vanligt bildpunkt för bildpunkt).

Tanken bakom: Om det reflekterande objektet är litet och långt ifrån omgivningen så bestämmer reflektionsvektorn vad som syns i en punkt. Objektet får inte heller reflektera sig självt, dvs det måste vara konvext. Även om förutsättningarna inte är uppfyllda kan resultatet bli sådant att det accepteras.



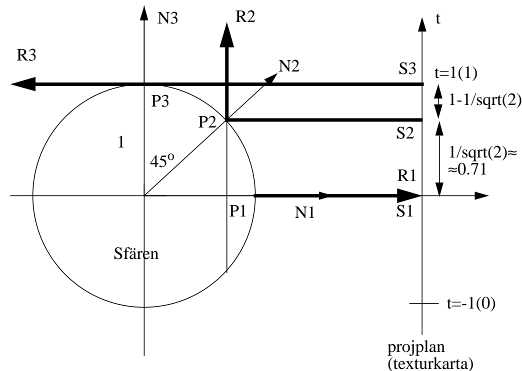
Det finns ett antal olika sätt att göra projiceringen.
1. OpenGL (och hårdvaran) ger stöd för omgivningsavbildning med sfär. Några OH beskriver detta sätt, men dessa kan till största delen överhoppas eftersom vi från hösten 2004 kommer åt omgivningsavbildning med kub (se nästa punkt).
2. OpenGL 1.3 (och någorlunda moderna grafikkort) ger stöd för dito med kub, som är en betydligt bättre metod.
3. En 5-6 år gammal metod med paraboloider är lovande och kanske ges hårdvarustöd (finns delvis). Vi tar ej upp den mera här.

Man kan ha ett antal invändningar mot såväl metoden i sig som de de enskilda projiceringssätten.

Demo: $DG/DEMOS/CUBEMAP/cubemapMB -i eller äldre `glut-3.7/progs/spheremap/glsmap` eller `glut-3.7/progs/advanced97/usespheremap` eller`ADV97_usespheremapMB2` eller `ADV99_spheremapMB`+data

Omgivningsavbildning med sfär 1(3)

Det finns flera sätt att plana ut en sfärisk yta. T ex kan man utgå från en vanlig parameterframställning av en sfär. Ett mindre känt sätt är det som beskrivs nedan och har stöd i OpenGL (och av grafikkort).

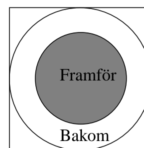


Den sfäriska texturen som är en enhetscirkel i en texturkarta utgörs helt enkelt av den bild vi ser då vi tittar på en högreflekterande enhetssfär ortografiskt. Sfären tänkes placerad med mittpunkten mitt i det reflekterande objektet.

Betrakta figuren ovan. Sfären finns till vänster och betraktaren till höger. Allt utspelar sig i vykoordinat/ögon-koordinatsystemet, dvs y-axeln är uppåtriktad och z-axeln är riktad mot betraktaren. En tänkt stråle S1 träffar sfären i punkten P1 och reflekteras tillbaka. Det som syns i P1 placeras alltså mitt i texturkartan (projektionsplanet). Strålen S2 motsvaras av en reflektionsvektor som R2 som går rakt upp. I texturkartan vid S2:s utgångspunkt placeras alltså det som finns rakt ovanför sfären. Strålen S3 går precis förbi sfären och bör alltså motsvaras av det som finns bakom sfären. Mellan utgångspunkterna

Omgivningsavbildning med sfär 2(3)

för strålarna S1 och S2 lagrar vi den delen av världen som finns till höger om sfären. Mellan S2 och S3 det som finns "bakom" sfären (till vänster om den). De båda delarna får väsentligt olika utrymme. Vi ser att olika reflektionsriktningar motsvarar olika punkter i texturen.



I figuren till vänster visas den sfäriska texturkartan symboliskt. Till höger visas en riktig (kafé Verona i Palo Alto, Calif).

Hur bildas den sfäriska kartan? Det finns minst tre sätt.
1. Genom fotografering av en sfär placerad i en verklig miljö.
2. Genom strålföljning.
3. Genom att göra lämpliga transformationer av kubavbildning.

Vid uppritningen genererar OpenGL texturkoordinater per hörnpunkt (motsv) anpassade för den sfäriska texturen förutsatt att vi gjort följande anrop

```
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);
```

Detta görs dock någorlunda korrekt bara för samma betraktningsrikning som användes då texturen bildades. Perspektiv får gärna vara påslaget. OpenGL har tillräcklig information för att beräkna synstrålen mellan betraktaren och hörnet (ögat är ju givet). Eftersom

Omgivningsavbildning med sfär 3(3)

hörnet måste vara försett med normal (som automatiskt omräknas till vykoordinater), kan systemet även beräkna en normaliserad reflektionsvektor R = (R_x,R_y,R_z) i vykoordinatsystemet. Texturkoordinaterna (s,t) beräknas slutligen med

$$L = \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

$$s = 0.5\left(\frac{R_x}{L} + 1\right)$$

$$t = 0.5\left(\frac{R_y}{L} + 1\right)$$

Utan additionerna +1 och muliplikationerna med 0.5 hamnar (s,t) på en cirkel med radien

$$\frac{R_x^2 + R_y^2}{R_x^2 + R_y^2 + (R_z + 1)^2} \le 1$$

och mittpunkt i origo. Termen $(R_z+1)^2$ i uttrycket för L gör således att reflektionsvektorer med olika z-komponent får olika texturkoordinater. De nämnda operationerna gör att den i stället hamnar inom det normala texturområdet 0≤s,t≤1.

Vektorstyrd texturkoordinatgenerering

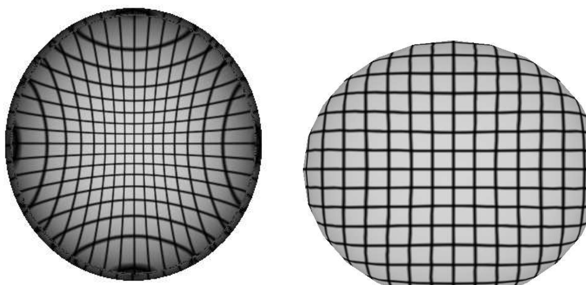
Det är vad som åstadkommes med parametervärdena (suffixet `_ARB` behövs inte)

`GL_SPHERE_MAP`

`GL_NORMAL_MAP_ARB`

`GL_REFLECTION_MAP_ARB`

Vi har redan avhandlat `GL_SPHERE_MAP`. Här kommer dock två bilder till. Vi har lagt vårt kvadratiska rutnät på en sfär. Högra bilden är gjord med perspektiv och återger mest korrekt den givna texturen. Den vänstra är med ortografisk projektion och förvränger givetvis på grund av texturkoordinatgenereringen mönstret.

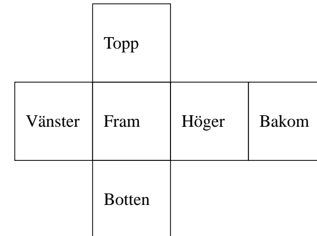


De två andra fallen styr genereringen av tre texturkoordinater (s,t,r) med hörnets normal respektive reflektionsvektor (samma som med `GL_SPHERE_MAP`), dvs även glEnable(GL_TEXTURE_GEN_R) och `glTexGeni(GL_R, GL_TEXTURE_GEN_MODE,...)` måste vara med. En tillämpning kommer på nästa OH.

Omgivningsavbildning med kub 1(2)

Sfärtexturen måste förnyas om observatören byter plats (rörelse i synriktningen må passera). Det vore önskvärt att ha ett sätt som är oberoende av var vi finns. Ett sådant sätt är kubavbildning.

Man placerar en enhetskub med mittpunkt i det reflekterande objektet. Man projicerar den omgivande världen på vart och en av de sex begränsningsplanen. Det kan ske genom att vi ritar upp scenen med sex olika 90°-iga synpyramider. Detta görs en gång för alla.

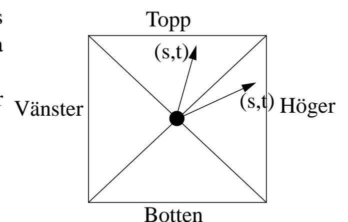


Vid uppritningen gäller det återigen att givet reflektionsvektorn i ett hörn beräkna vilken av de sex texturerna som berörs och koordinater inom den. NVIDIA introducerade hårdvarustöd för detta hösten 1999, vilket gör att sådan avbildning kan göras i realtid. På våra SUN-ar görs det i mjukvara och blir därmed långsamt (går t o m något snabbare med Mesa). I rum 6220 går det från hösten 2004 undan!

Man använder alltså moden `GL_REFLECTION_MAP` för generering av s,t,r-koordinater. Vektorn (s,t,r) är reflektionsvektorn i vykoordinatsystemet. Denna vektor används för val av en av de 6 texturerna och

Omgivningsavbildning med kub 2(2)

sedan indexering i rätt. Låt oss bara antydningsvis belysa detta för 2D-fallet då vektorn är (s,t). Om s och t positiva, väljs höger sida om s>t, etc.



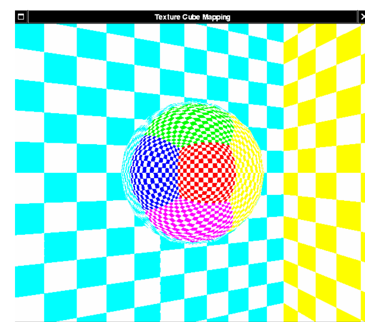
Rent praktiskt i OpenGL

```
glEnable(
  GL_TEXTURE_CUBE_MAP_ARB);
```

och för varje kubsida enligt modellen

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB,...
```

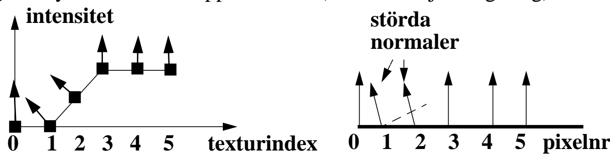
Exempel: `cubemap.c` (i Mesa-distributionen). En reflekterande sfär



i ett rum med rutiga väggar. **Demo**: Bubbles från NVIDIA om jag bara hade kunnat visa det.

Bump-mapping 1-2(2)

Syfte: Med texturering ökar vi realismen. Men det finns mer att önska sig. Ett äpple är rätt slätt, men det gäller inte en apelsin som har små gropar (eng. bump). Även tegelmurar och gräsmattor är "gropiga". Vi vill alltså ge intryck av gropighet eller reliefstruktur utan att detaljmodellera.
Metod: Normalerna störs med utgångspunkt från t ex en höjdkarta lagrad som en textur. Man får på det viset en störd normal N' för varje bildpunkt och räknar med hjälp av den ut ljusvärdet per bildpunkt i stället för som normalt per hörnpunkt. Förfaringssättet är kostsamt. Men med nyare processorer går det som en dans.
På nästa OH finns tre bilder kring bumpmapping som fungerar trycktekniskt (OBS! Ej avsedda som reklam).
Överst syns resultatet. Nedtill till höger finns den grundläggande texturen. Till vänster finns den textur, en höjdkarta (vitt=hög höjd, svart = låg), som styr bump-mappingen. Låt oss försöka beskriva idén i en endimensionell variant. Givet höjdkartan (vänstra fig) kan vi beräkna normaler i varje texel. När vi sedan lägger den grundläggande texturen på vår yta (här för enkelhets skull ett plan och 1-1 pixel/texel, högra fig) stör vi vid belysningsberäkningen ytans normaler proportionellt mot den horisontella komponenten av normalen i vänstra fig. Detta gör att ytan illusoriskt upplevs brant (streckad linje i högra fig).



Ofta lagras inte höjdkartan utan i stället motsvarande beräknade normalkarta. En komplikation är att om ytan är krökt eller ligger snett, måste texturnormalerna transformeras till ytans tangentplan.

3D-texturer

Våra föremål är i allmänhet 3-dimensionella. Ibland är texturerna på angränsande ytor av litet olika typ men ändå måste passa ihop. Ett typiskt exempel är en träkloss, där vi på en sida kan se ringar, men på en annan bara den längsgående ådringen.

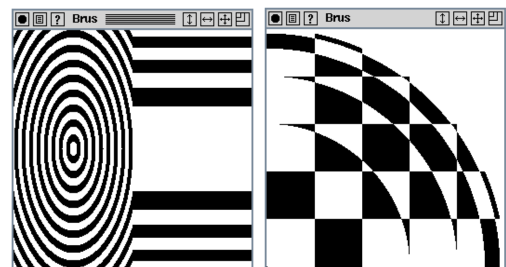
Lösningen på detta problem är 3D-texturer. Man talar även om solid texturering. Man kan tänka sig två varianter:

1. Ett antal lager av 2D-texturer. Om varje 2D-textur är på 256x256 skulle man kanske ha 256 lager. Utrymmesbehovet växer således krafigt och knappast något grafikkort skulle kunna hålla allt hos sig. Visst kan man minska på texturupplösningen, men då missar man ju detaljer. T ex passar tekniken inte för träklossar. Inte heller eventuell komprimering förbättrar situationen tillräckligt. Men för enklare texturer går det ju bra.
   OpenGL 1.2 och 1.3 (som finns på våra SUN'ar) har stöd för denna teknik. I princip behöver man bara byta ut konstanten `GL_TEXTURE_2D` mot `GL_TEXTURE_3D` genomgående och anropa `glTexImage3D` i stället för `glTexImage2D`. Härvid anger man förutom texturbredd och texturhöjd även texturdjup . Dessutom måste man naturligtvis nu ange tre texturkoordinater för varje hörn, vilket sker med `glTexCoord3f(...)`.
   OpenGL-stödet kan också användas för s k volymvisualisering, dvs visualisering av 3D-mätdata.
2. En annat sätt är att generera erforderlig texturinformationen vid behov. Kort sagt när vi behöver texturen för en texturkoordinat (s,t,u) anropar vi en funktion med punkten som parameter. Funktionen räknar ut aktuellt värde. Detta kan innebära ett omfattande räknearbete och metoden passar inte realtidskrav. Man brukar tala om **procedurtexturer** (eng procedural textures). OpenGL ger inget som helst stöd. Används av alla avancerade strålföljare. I framtiden kanske den här typen av beräkningar kan utföras av grafikprocessorn (eller någon sidoprocessor till den).
   Detta sätt kan användas även för 2D-texturer. Ibland kallar man texturer framställda så här för **syntetiska texturer**.

Procedurtexturer

Ex. 1: Trätextur. Vi tittar mot en kant på en kub. Vänstra figuren. Här liksom i nästa exempel ortografisk projektion.

Ex. 2: Blocktextur. Vi tittar mot en sfär. Högra figuren.



För trätexturen användes proceduren (funktionen)

```
float wood(int frekvens, float x, float y, float z) {
   float r;
   r = sqrt((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5));
   return ((int)(frekvens*r))%2;
}
```

med frekvensen = 32. Den ger 0 eller 1, som får styra vitt/svart. För blocktexturen

```
float block(int frekvens, float x, float y, float z) {
   int f = frekvens;
   return ((int)(f*x)+(int)(f*y)+(int)(f*z)) % 2;
}
```

med frekvensen = 5. Även nu 0 eller 1 som resultat. Vi är inte riktigt nöjda. Vi behöver litet slump för att det skall bli mera verklighetsnära.

Texturer i Art Of Illusion: Allmänt

Vi vill **belysa** hur man arbetar med texturer i ett modelleringsprogram. För detta väljs i år *Art Of Illusion* (i fortsättningen förkortat *AoI*).

Det verkar som om version 2.1 av *AoI* är instabil i både vår Linux- och Windowsmiljö när man vill göra något utöver det enklaste. Förmodligen har det med det just införda JOGL-stödet att göra. Jag har i de kommande exemplen använt närmast föregående version 2.0 utan problem. Den nås med Linux-kommandot `artof2`. I Windows kan man klicka på
`/users/course/TDA360/AOI2.0/ArtOfIllusion2.0/ArtOfIllusion.jar`

Jag redovisar på dessa OH varje steg tydligt för att den som vill prova inte skall behöva fördjupa sig i annat material, men kommer nog att vara översiktligare vid genomgången. Detta leder till att det blir många OH. Bilderna blir tyvärr väl små i det tryckta materialet.

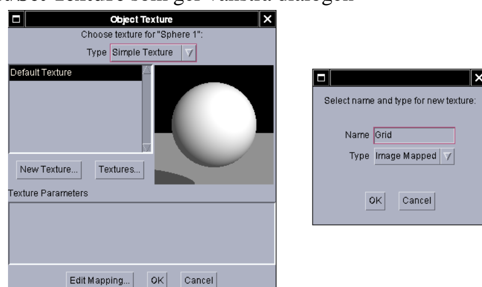
I *AoI* har inte materialegenskaper samma betydelse som i OpenGL. Man använder i stället texturer för detta. När materialegenskapen är jämn över ytan, används "uniform" texturer. Vi hoppar över dessa helt. I *AoI* dräller det av inställningsmöjligheter. Vi ordar inte om dem. Den som vill veta detaljer får läsa den trevliga manualen.

Vi tar upp (filerna i `$DG/AOI`)

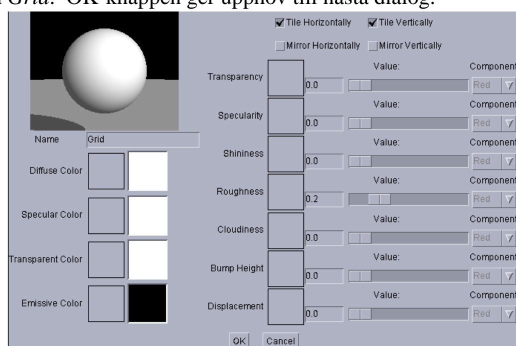
- Bildtexturer med standardavbildning (`DEMO_TEXTUR1.aoi`)
- Bildtexturer med UV-avbildning (`DEMO_TEXTUR2.aoi`)
- Procedurtexturer (`DEMO_TEXTUR3.aoi`)

Texturer i Art Of Illusion. Bildtextur. 1(4)

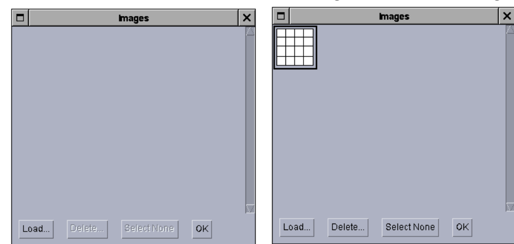
Börjar med att lägga in en sfär, till vilken vi avser att koppla en textur. Ser till att sfären är vald (t ex via objektlistan till höger). Använder **Object**/**Set Texture** som ger vänstra dialogen



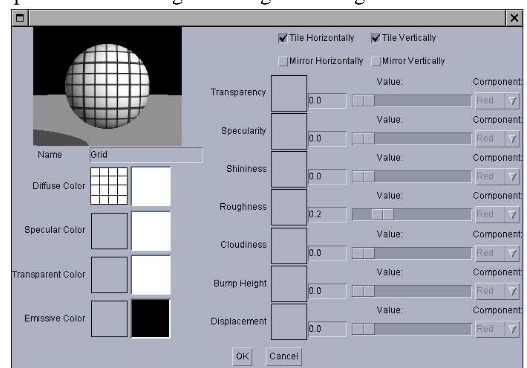
Trycker på **New Texture**, som leder till högra dialogen ovan, där jag ersätter typen **Uniform** med **Image Mapped** och ger texturen ett namn *Grid*. OK-knappen ger upphov till nästa dialog.



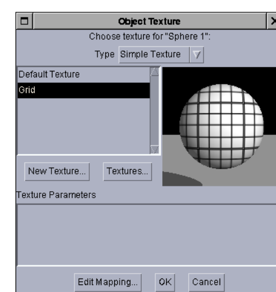
Roughness-värdet 0.2 är troligen förinställt. Jag låter det vara liksom alla övriga värden. Trycker på fyrkanten omedelbart till höger om texten **Diffuse Color**, varefter vänstra dialogen nedan visar sig.



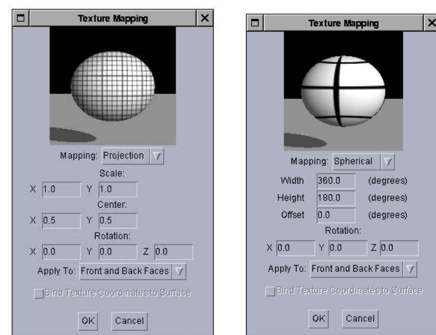
Load-knappen ger oss en traditionell fildalog. Jag letar upp mappen `/users/course/TDA360/AOI` och väljer bildfilen `R.png`. Då kommer den bilden in i bildförrådet och det ser ut som i den högra figuren. Tryck på OK och en tidigare dialog ändrar sig till



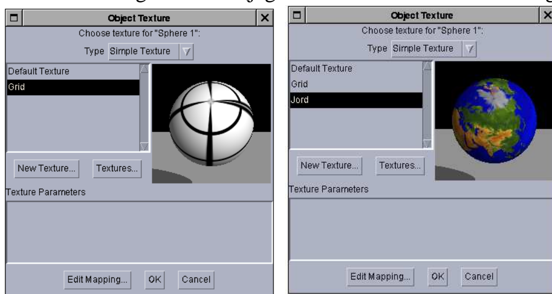
Tryck på **OK**. Nu har vi bara kvar dialogrutan (markera *Grid*)



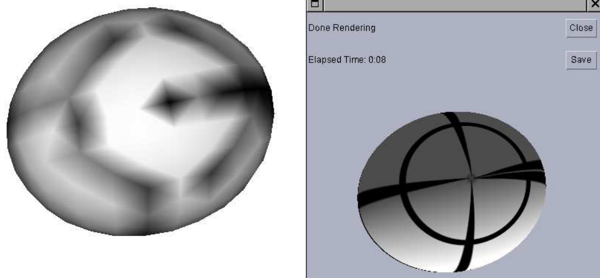
I bildrutan ser vi texturen projicerad på en sfär (kan ändras till bl a kub via MK3-meny). Man kan också med MK2 rotera objektet. Just nu används en texturpåläggning som i *AoI* kallas **Projection**. Den är i allmänhet bara lämplig för plana objekt. Vi vill byta till **Spherical** och trycker därför på **Edit Mapping**, som ger vänstra figuren. Efter byte



av **Mapping** ser det ut som i högra figuren ovan. Vi bryr oss inte om alla andra ifyllnadsmöjligheter utan trycker på **OK** och har nu bara kvar vänstra dialogen nedan (jag har här roterat sfären en aning). Vi



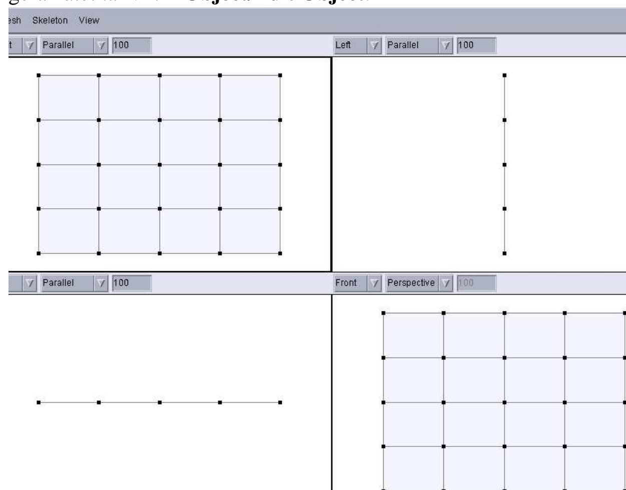
ser till att **Grid** är markerad och avslutar med **OK**. Högra figuren visar hur det ser ut om en till textur fanns. Byter till **Scene/Display Mode/ Textured** (från det normala **Smooth**). Då ser det ut så här i AoI:s huvudfönster. En lågupplöst version av texturen ligger pålagd. Med **Scene/Render Scene** får vi den korrektare högra bilden (jag såg till att bakgrunden blev genomskinlig med **Scene/Render Scene/Output**)



Texturer i Art Of Illusion: UV-avbildning. 1(4)

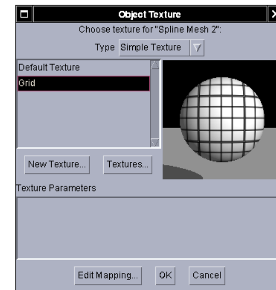
UV-avbildning (eng. UV-mapping) är den etablerade termen i modelleringsprogram för motsvarigheten till texturkoordinater i OpenGL. Fungerar i *AoI* enbart för nät (eng. mesh) (splinesnät, som skapas med verktyget till höger, eller triangelnät, som skapas ur annat objekt med **Object/Create Triangle Mesh**).

Vi skapar först ett nät med detta verktyg. Som standard får man ett 5x5-nät, men genom att dubbelklicka på verktyget kan man ändra till annat. Vi accepterar det förinställda. Fånigt nog visas inte nätet som ett 5x5 i AoI-fönstret, utan det blir fler rutor. För att eventuellt redigera nätet tar vi till **Object/Edit Object**.

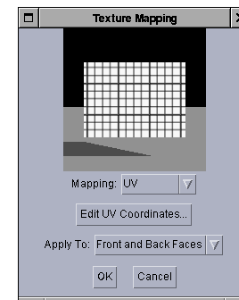


Vi ser att det är ett nät med 5x5-styrpunkter, dvs 4x4-rutor. Vi bryr oss inte om att bukta på nätet nu, så lämna med **OK** (under den del av fönstret som fick plats här).

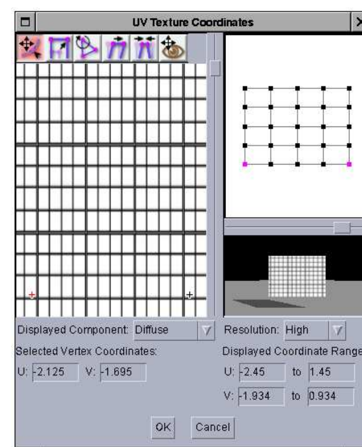
Vi kopplar i stället objektet - nätet - till en bild på vanligt sätt och kallar texturen *Grid*. Vi använder samma bild `R.png` som tidigare.



Tryck på **Edit Mapping** och ändra **Projection** till **UV**



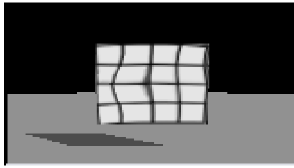
och sedan **Edit UV Coordinates**, som ger (varvid jag ändrat **Resolution: Low** till **High** och zoomat in på nätet).



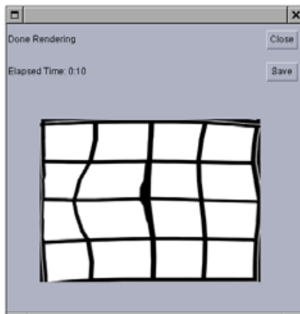
Upp till höger visas nätet. Till vänster finns texturen periodiskt upprepad. *AoI* har automatiskt tilldelat varje styrpunkt texturkoordinater och dessa ligger i de intervall som anges till höger under **Displayed Coordinate Range**. Dessa motsvarar dessutom gränserna för texturområdet till vänster. Kanske noterar vi att U-intervallet har längden 3.9, medan V-intervallets längd är 2.868. Under nätet finns en "preview" av texturen applicerad på nätytan. Man kan markera styrpunkter i nätet. Motsvarande punkt visas då i texturen. Genom att markera punkten i texturen visas koordinaterna. T ex avläser vi i figuren att nedre vänstra styrpunkten har texturkoordinaten (-2.125, -1.695).

Motsvarande högra har (1.125, -1.695) och övre vänstra (-2.125, 0.695), dvs avståndet i U-led är 3.25 och i V-led 2.39. Verkar som om programmakaren begett ett tankefel.

Vi kan nu enkelt ändra texturkoordinater genom att flytta markerade punkter i texturen. Detta görs i allmänhet för att få små korrigeringar i vissa områden. Men vi försöker i stället se till att den ursprungliga texturen brer ut sig över hela ytan. Efter det tålamodsprövande flyttandet av 25 punkter ser "preview"-delen ut så här:

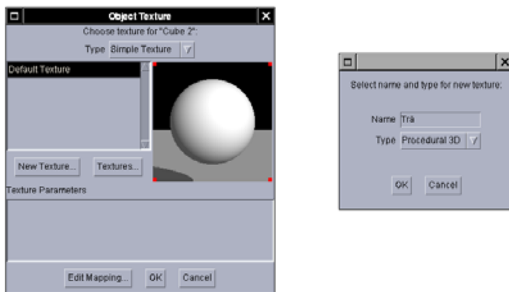


Tryck på minst två OK-knappar och vi är tillbaka i AoI-fönstret. Rendering ger

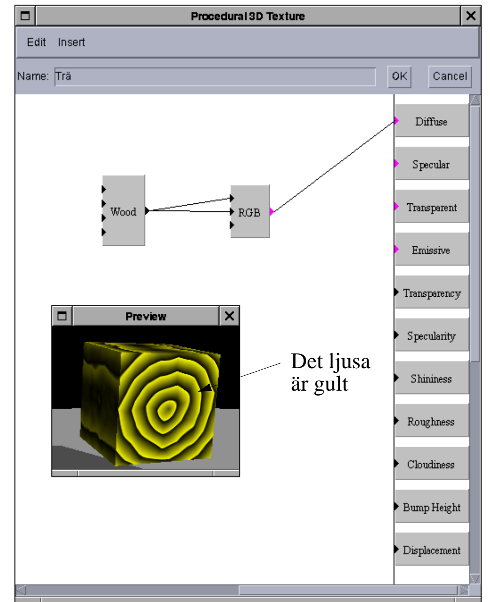


Texturer i Art Of Illusion: Procedurtextur. 1(4)

Man arbetar med bl a svarta lådor på ett sätt som är vanligt och som du kanske mött i något annat program. Vi skall nu knyta en trästruktur till en kub. Vi skapar först en kub och ser till att den är vald. Därefter **Object/Set Texture** (vänstra figuren). Med knappen **New Texture** får vi en ny dialogruta (högra figuren), där vi namngivit texturen med *Trä* och bytt från **Uniform** till **Procedural 3D**.



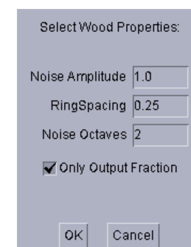
Efter **OK**, ser det ut som på nästa sida, bortsett från att arbetsytan från början är tom. Jag har med **Insert/Color Functions/RGB** och **Insert/ Patterns/Wood** lagt in två boxen RGB och Wood. Utgången på Wood har kopplats ihop med R-ingången och G-ingången på RGB och RGB-utgången har sammanbundits med ingången på Diffuse. Man ser hela tiden ett Preview-fönster av samma typ som tidigare (med MK3 har jag bytt från Sphere till Cube).



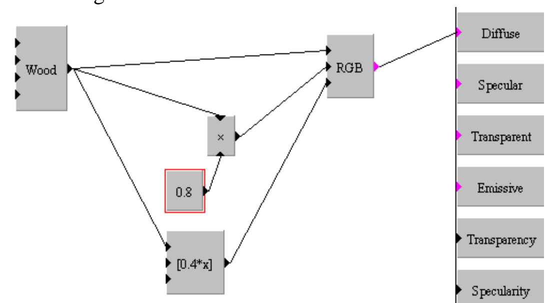
En box har ett antal inportar och normalt en utport. Vad dessa står för kan få reda på genom med godtycklig musknapp markera porten ifråga. Man får då också reda på standardvärdet. T ex ger utporten på RGB-boxen som väntat Color och den översta inporten R(0), som betyder att porten avser rödvärdet och att detta som standard är 0. För

Wood-boxens översta inport får man X(X), som betyder att värdet normalt är x-värdet i modellkoordinatsystemet. Understa blir Noise(0.5), vars tolkning inte är lika uppenbar.

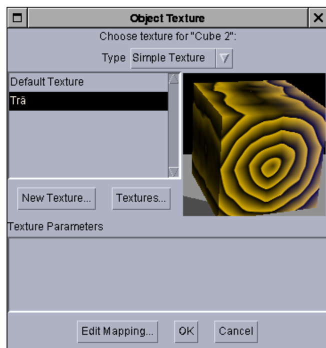
Vissa boxar är redigerbara. Klicka i så fall på boxen. T ex för Wood-boxen.



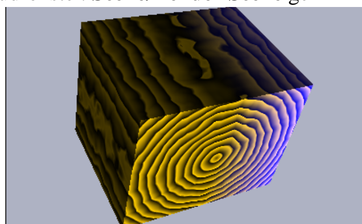
Om man är missnöjd med färgen i trätexturen, får man blanda färgerna annorlunda. Jag hade fått för mig att färgerna RGB i proportion 1:0.8:0.4 skulle ge en brun ton. Ändrade därför till



men nådde inte riktigt önskad brun nyans. Som synes finns det boxar för multiplikation, tal och allmänna uttryck. Dessa hämtas med hjälp av **Insert**-menyns undermenyer.
När vi är nöjda trycker vi på **OK** och återvänder därmed till (se till att Trä är markerad).



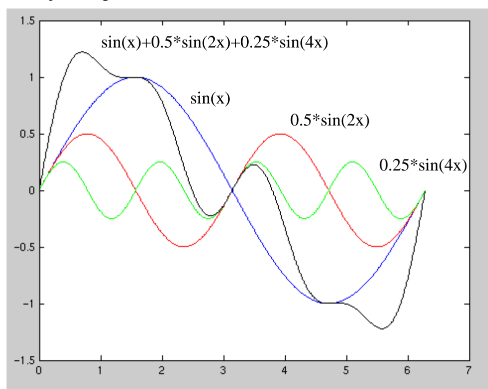
Om vi tar till **Edit Mapping** upptäcker vi att nu finns bara **Linear**-mapping (inte t ex **Spherical**), vilket beror på att en procedurtextur ju beskriver varje voxel direkt. Tryck slutligen på OK och vi är tillbaka hos AoI:s huvudfönster. **Scene/Render Scene** ger



Perlin-brus 1 (10)

Vi skall nu beskriva en teknik som introducerades av Ken Perlin 1985 och som kan användas för tillverkning av texturer och effekter som är en aning påverkade av slumpen. Den har använts mycket i filmindustrin och Perlin har tilldelats pris av denna industri. Han har senare modifierat tekniken på olika sätt (se OH efter dessa 10) och presenterade 2001 en ny metod kallad “simplex brus”, som är lättare att implementera i hårdvara. Jag (liksom Hills bok) gör litet våld på hans algoritm.

Men först en utflykt till MATLAB. Förmodligen vet du att signaler (ljud etc) från verkligheten kan delas upp i olika frekvenser. Ofta finns det en dominerande frekvens (eller frekvensband). I följande figur illustrerar vi detta.

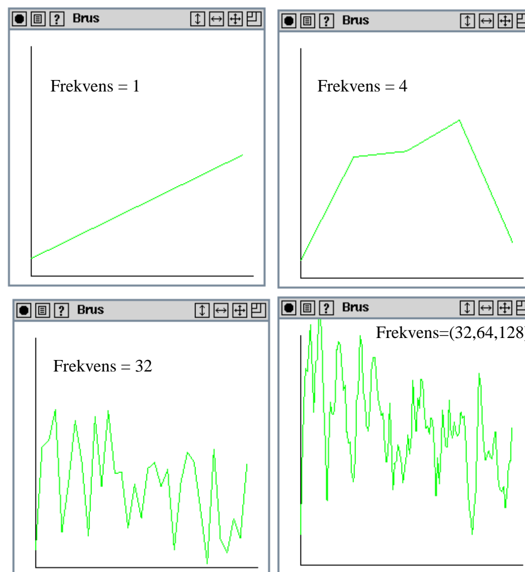



```
>> x = 0:0.01:2*pi;       % Bilda samplingsvektor
>> plot(x,sin(x),'b')     % Rita sin(x) med blått
>> hold on          % Se till att gammalt finns kvar
>> plot(x,0.5*sin(2*x),'r')   % Rita 0.5*sin(2x)
>> plot(x,0.25*sin(4*x),'g')  % Rita 0.25*sin(4x)
>> plot(x,sin(x)+0.5*sin(2*x)+0.25*sin(4*x),'k')
                         % Rita summan
```


Perlin-brus 2 (10)

Vi använder nu analogitänkande för att konstruera olika slag av brus. Vi börjar med endimensionellt kontinuerligt brus för intervallet [0,1]. Först måste vi hitta en motsvarighet till det vanliga frekvensbegreppet.

Låt oss dela in [0,1] i N lika stora delintervall med hjälp av punkterna
$x_i = i/N$, i=0,1,...,N. I var och en av dessa punkter skapar vi ett slumptal y_i (likformig fördelning på [0,1)). Vi förbinder sedan punkterna (x_i,y_i) i tur och ordning med räta linjer. Detta innebär att vi linjärinterpolerar de genererade slumpvärdena för att få en funktion definierad för alla x i intervallet. Resultat för N=1, N=4 och N=32 ser du i följande figur. Låt oss helt enkelt kalla antalet delintervall, dvs N, för **frekvensen**. I den fjärde figuren har

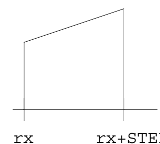


Perlin-brus 3 (10)

vi blandat brus med frekvenserna 32, 64 och 128 på samma sätt som när vi blandade sinuskurvorna, dvs amplituderna är 1, 0.5 respektive 0.25. Vitsen med detta är att vi får detaljvariation utan de kraftiga variationer vi skulle fått med frekvensen 128.

Utdrag ur kod (ligger i omritningsproceduren *display*; vi beräknar brusvärdet i brytpunkterna och ritar):

```
glOrtho (-0.1, 1.1, -0.05, 1.5, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
axlar();   // Koordinataxlarna
STEP = 1.0/frekvens;
glBegin(GL_LINE_STRIP);
   for (i=0; i<=frekvens; i++) {
       rx = i*STEP; b = brus(frekvens,rx);
       glVertex2f(rx,b);
   }
glEnd();
```



I koden anropas en funktion *brus* som givet en frekvens och ett x-värde räknar ut brusvärdet. Den returnerar alltid samma värde för givna parametrar. Hur det går till återkommer vi till.

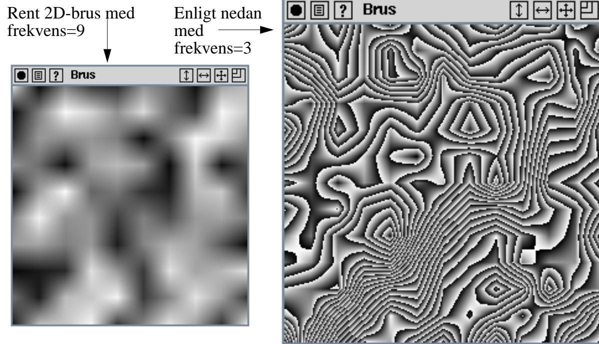
När man blandar flera frekvenser brukar man tala om **turbulens**. Speciellt vanlig är en blandning av typen (vi skriver den för 3D-fallet)

$$turbulens(frekvens,x,y,z) = \sum_i^M \frac{1}{2^i} brus(2^i frekvens,x,y,z)$$

Tanken är att s k 1/f-brus skall approximeras, dvs brus där amplituden avtar med 1/frekvensen.

Perlin-brus 4 (10)

Vi kan även arbeta med 2D-brus, dvs brus definierat över en kvadrat $0 \le x,y \le 1$ (för enkelhets skull). **Ett exempel:**

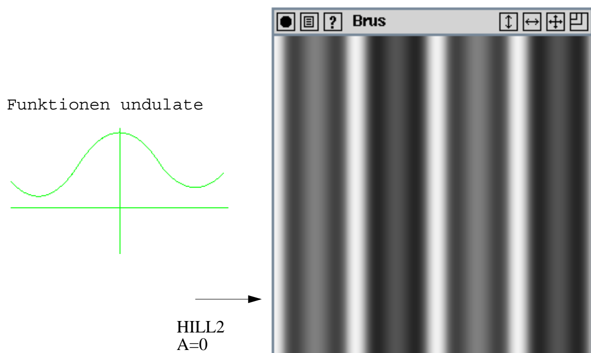


Vi ritar nu punkt för punkt. **Utdrag ur kod** (uppritningsproceduren)

```
glOrtho (0.0, width, 0.0, width, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); STEP = 1.0/width;
for (i=0; i<width; i++) {
    for (j=0; j<width; j++) {
        rx = i*STEP; ry = j*STEP;
        b = brus2(frekvens,rx,ry)+
            0.5*brus2(2*frekvens,rx,ry)+
            0.25*brus2(4*frekvens,rx,ry);
        b = 20*b; b = b - (int)b;
        glColor3f(b, b, b);
        glBegin(GL_POINTS);
             glVertex2f(i,j);
        glEnd();
    }
}
```


Perlin-brus 5 (10)

Ytterligare ett 2D-exempel, där vi försöker syntetisera marmor. Man startar med att identifiera något grunddrag i den önskade texturen. I marmorfallet kanske som här ett antal vertikala ådringar.



Vi skulle kunna få ett mönster av denna typ genom att i förra programmet ha

```
b = sin(12.28*rx);
glColor3f(b, b, b);
```

Dvs än så länge ingen slump. Bilden och de två följande är dock gjorda med ytterligare en rad mellan de två angivna

```
b = undulate(b);
```

Se längre fram. Resultatet blir något sämre utan den.

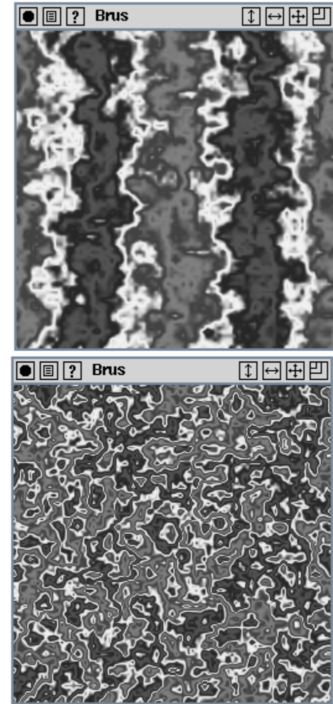
Sedan stör vi det först bildade b-värdet slumpmässigt med variation både i x och y, vilket resulterar i bilderna på nästa sida..

```
 b = brus2(frekvens,rx,ry)+
     0.5*brus2(2*frekvens,rx,ry)+
     0.25*brus2(4*frekvens,rx,ry);
 b = undulate(sin(12.28*rx+A*b));
 glColor3f(b, b, b);
```


Perlin-brus 6 (10)

HILL2
frekvens=20
A=2

HILL2
frekvens=20
A=8



Funktionen *undulate* (andragrads-spline på [-1,1]) är knyckt från Hills bok:

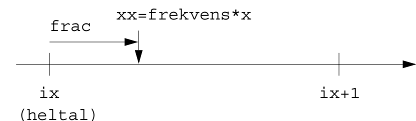
```
float undulate(float x) { if (x<-0.4) return 0.15+2.857*(x+0.75)*(x+0.75);
                else if (x<0.4) return 0.95-2.8125*x*x;
                 else return 0.26+2.666*(x-0.7)*(x-0.7);}
```


Perlin-brus 7 (10)

Hur konstrueras brus-funktionen `brus(frekvens, x)` respektive den tvådimensionella `brus2(frekvens, x, y)`? Vi vill kunna beräkna värdet för godtyckliga reella tal $0 \le x,y \le 1$ och (heltals)frekvens<256. Det är viktigt att ett visst par (x,y) - för given frekvens - alltid ger samma resultat (annars skulle texturen förändras), men samtidigt skall värdena sinsemellan te sig slumpmässiga. I 2D-fallet skulle man kunna tänka sig att en gång för alla räkna ut värdena i ett lagom tätt gitter och interpolera för övrigt. Med en frekvens om 255 skulle det betyda cirka 256x256 värden, vilket vore acceptabelt. Men eftersom vi vill överföra idén till 3D, där motsvarande behov i så fall skulle vara 256x256x256 värden, vilket vi bedömer som orimligt stort, skall vi göra på ett annat - approximativt - sätt. En stund framöver antar vi ändå att vi har en brusvektor med komponenter b_i, i=0,1,...,255 resp en brusmatris med komponenter b_{ij}, i,j=0,1,2,...,255.

Då kan vi i **1D-fallet** skriva

```
float brus(int frekvens, float x) {
    float xx = frekvens*x;  // dvs max frekvens
    int ix = (int)xx;   float frac = xx - ix;
    return b[ix] + frac*(b[ix+1]-b[ix]);
}
```



Om vi inför en funktion för linjärinterpolation

```
float LIP(float t, float a, float b) {
     return a + t*(b-a);
}
```

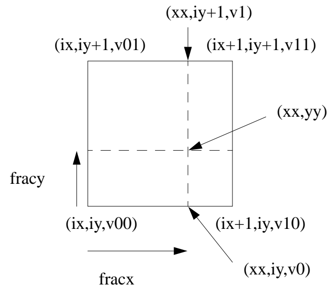
kan return-raden ersättas av

```
return LIP(frac, b[ix], b[ix+1]);
```

(i engelsk litteratur LIP -> lerp = Linear intERPolation)

Perlin-brus 8 (10)

I **2D-fallet** bildar vi på motsvarande sätt xx=frekvens*x och yy=frekvens*y och interpolerar sedan enligt figuren



vilket ger koden

```
float brus2(int frekvens, float x, float y) {
   float xx = frekvens*x, yy = frekvens*y;
   int ix = (int)xx, iy = (int)yy;
   float fracx = xx - ix, fracy = yy - iy;
   float v00, v10, v11, v01;
   float v1, v0;
   v00 = b[ix][iy]; v10 = b[ix+1][iy];
   v11 = b[ix+1][iy+1]; v01 = b[ix][iy+1];
   v1 = LIP(fracx,v01,v11);
   v0 = LIP(fracx,v00,v10);
   return LIP(fracy,v0,v1);
}
```


Perlin-brus 9 (10)

3D-fallet, som var vårt egentliga mål, hanteras på liknande sätt. Nu måste vi förutsätta att vi har slumpvärden b_{ijk}, i,j,k=0,1,...,255. Och antalet interpolationer växer.

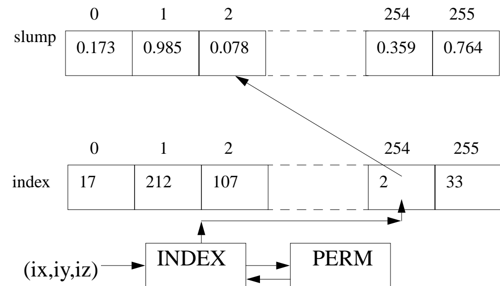
```
float brus3(int frekvens, float x, float y, float z) {
   float xx = frekvens*x, yy = frekvens*y;
         zz = frekvens*z;
   int ix = (int)xx, iy = (int)yy, iz = (int)zz;
   float fracx = xx - ix, fracy = yy - iy,
         fracz = zz -iz;
   float v000, v100, v110, v010, v001, v101,
         v111, v011;
   float v00, v10, v11, v01;
   float y1, y0;
   // Hörnvärdena; snurror bättre men trasslar till
   // figuren
   v000 = b[ix][iy][iz]; v100 = b[ix+1][iy][iz];
   v110 = b[ix+1}[iy+1][iz];
   v010 = b[ix][iy+1][iz)];
   v001 = b[ix][iy][iz+];
   v101 = b[ix+1][iy][iz+1)];
   v111 = b[ix+1][iy+1][iz+1];
   v011 = b[ix][iy+1][iz+1];
   // Interpolera till planet iz+fracz
   v00 = LIP(fracz, v000, v001);
   v10 = LIP(fracz,v100, v101);
   v11 = LIP(fracz,v110, v111);
   v01 = LIP(fracz,v010, v011);
   // Interpolera till linje
   y1 = LIP(fracx,v01,v11);
   y0 = LIP(fracx,v00,v10);
   // Interpolera till punkt
   return LIP(fracy,y0,y1);
}
```


Perlin-brus 10 (10)

Till sist hur vi undviker matrisen [b_{ij}] och aggregatet [b_{ijk}].

Vi bildar en enda slumpvektor (större längd möjlig) `float slump[256]` med 256 tal mellan 0 och 1. För det kan vi använda `rand(...)/(RAND_MAX-1.0)`.
C-funktionen `rand()` producerar i vår miljö heltalsslumptal i intervallet [0,32767] och RAND_MAX=32767, dvs vi får reella tal i intervallet [0,1).

Vi bildar dessutom en heltalsvektor `int index[256]` med heltalen 0-255 väl blandade.



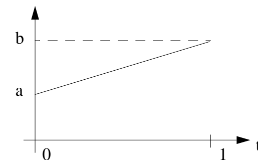
Under processen behöver vi beräkna slumptal för trippler (ix,iy,iz) med heltal. I princip skulle vi kunna använda slumpvärdet slump[(ix+iy+iz) % 256]. Men för att reducera risken för oönskade regelbundenheter görs detta på ett något omständligare sätt via funktioner INDEX och PERM. Dessa kan se ut så här (för den C-bildade: dessa kan med fördel vara makron):

```
int PERM(int x) {
 return index[x & 255];//Snabbare än mod-beräkning med %
}
int INDEX(int i,int j, int k) {
   return PERM(i+PERM(j+PERM(k)));
}
```

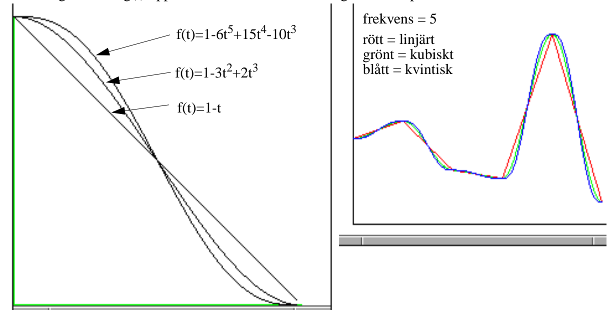
Vi skriver sedan slump[INDEX(ix,iy,iz)] i st f b[ix][iy][iz] i koden ovan.

Mera Perlin-brus

Bruset som vi hittills beskrivet det blir hackigt (ej kontinuerlig derivata). Det beror naturligtvis på att vi interpolerar linjärt



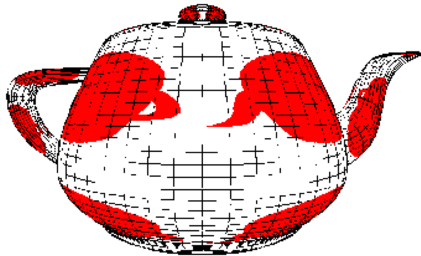
vilket kan skrivas
$F(t) = a + t(b - a) = (1 - t)a + tb = (1 - g(t))a + g(t)b$ med $g(t) = t$.
Enda sättet att åstadkomma kontinuerlig lutning är att se till att $F'(0) = F'(1) = 0$. För det måste vi välja g(t) annorlunda. Eftersom $F'(t) = (b-a)g'(t)$, måste $g'(0) = g'(1) = 0$ och självklart $g(0) = 0$ och $g(1) = 1$. Dvs g(t) Hermite-interpolerar. Man finner lätt att
$g(t) = 3t^2 - 2t^3$ uppfyller kravet. Perlin använde motsvarande F(t) i sin ursprungliga metod. Senare ville han även kontinuerlig andraderivata, vilket uppfylles med femtegradspolynomet
$g(t) = 6t^5 - 15t^4 -10t^3$.
I vänstra figuren är 1-g(t) uppritad för de tre fallen. Till höger en slumpkurva också för de tre fallen.



Rendering till textur

Vid t ex kub-mappning vore det bekvämt att kunna rita direkt till en textur och senare använda den som sådan. Det finns en ARB-utvidgning (begreppet beskrivs senare) *WGL_ARB_render_to_texture* som tillåter detta, som har stöd i Windows-miljö, men inte i Linux-miljö (jag gissar att det har med GLX att göra). Ev kan den senaste utvidgningen *ARB_pixel_buffer_object* (Dec 2004) hjälpa.

I figuren visas ändå en sådan effekt. Först har jag ritat upp en röd tekanna mot en textur. Texturen har sedan använts som textur för samma tekanna. För att vi verkligen skall se att föremålet är en tekanna har jag ritat kannan även som en trådmodell (p g a djupbuffertens arbetssätt syns inte alla delarna av trådarna (en lösning på det problemet beskrivs senare).



I Linux-miljö (RENDER_TEXTUR.c) kan effekten uppnås genom att vi ritar den röda tekannan i det osynliga bildminnet och rasterkopierar till en variabel, som sedan används som sista parameter i anropet av *glTexImage2D*. Anropen av *glReadPixels* och *glTexImage2D* kan kombineras till ett enda anrop `glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0, X,Y, 0)`, där *X* och *Y* anger fönsterbredd och -höjd.

Animering

Syfte med dessa sidor: Att ge en liten inblick i ett jätteområde som dessutom är hårt knutet till kommersiell eller intern programvara. Hearn/Baker: kap 16, 584-597, Angel: 347-349(gamla), 435-437 (nya), Hill har mycket lite: 170-172. Möller: Inget.

Vad är animering?: Levandegöra. Förknippades ursprungligen med tecknad film. Eller dockfilm. Eller lerfilm. Nu datorframställd (Toy Story, A Bugs Life etc). Animeringen kan utspela sig i realtid eller vara inspelad. Vetenskaplig animering: någon process simuleras och visas grafiskt (såväl MATLAB som Mathematica klarar sådant). Animeringen kan ha interaktionsmöjligheter.

Många utmaningar: Fotorealism, stelkroppsrörelse, mänsklig rörelse, tygs interaktion med omgivningen, kollision, ansiktsuttryck, synkronisering av tal och munrörelse, natureffekter (eld, rök, moln etc).

Milstolpar: Disney (Musse Pigg 1928, Snövit 1933), Fleischer (Betty Boop 1930), östeuropeisk dockfilm, leranimering, datoranimering (Luxo jr 1986, Tin Toy 1988, Jurassic Park 1993, Toy Story 1995/Antz 1998/A Bugs Life 1998), BBC Dinosaurs 2000.

Datoranimering

Marknad: Reklamfilm, spelfilm, barnprogram.

Producenter av datorgenererad film: Pixar (Toy Story, Toy Story II), Pacific Data Images, Industrial Light & Magic (Jurassic Park), Aaardman (Wallace & Gromit).

Exempel på programvara för animering: SoftImage, 3D Studio MAX, LightWave, Alias/Wavefront Maya, TrueSpace. Alla program för fotorealism brukar klara viss animering, gäller t ex POVRay (men modelleraren Moray ger inget eget stöd).

Vinst med datoranimering: 3D, kameraåkning (inkl korrekt perspektiv) lätt, bildritningen automatiseras, mellanbilderna kan genereras direkt, snabbare, billigare (?), mera detaljer kan tillåtas (texturer, massor av ljuskällor), mass-scener, effekter.

Resursbehov: 24 bilder per sekund, dvs 100 minuter film kräver 100x60x24 = 144000 bilder. Vid inspelad animering kan man lägga ner mycket tid per bild. Pixar uppgav 1996 1-10 tim. Med 1 tim/bild blir det en total datortid om 144000/24 = 6000 dygn, dvs 16.5 år. Inte konstigt att produktionen av Toy Story gjordes med ett kluster av ett hundratal datorer. Även minnesbehovet stort. Med 1 MB/bild (för litet för filmkvalitet men i överkant för video) blir det 144 GB.

Traditionell animering (tecknad film)

1. Skissat bildmanus (eng. story-board), en bild per scen, typiskt 2-10 sekunder.
2. Ljud
3. Bakgrund
4. De rörliga objekten ritas bild för bild på plastark (celluid), eng. cel animation[1]
5. Stel rörelse: Arket rörs över bakgrunden och avfotograferas.
   Normal rörelse: Arken byts successivt ut.

Man brukar säga att resultatet blir 2 1/2D (personerna är platta vid rörelse men kan finnas på olika djup). Med datoranimering blir det lättare att arbeta i 3D.

Bilderna var av två slag - huvudbilder (eller nyckelbilder; eng. key-frame) och mellanbilder (eng. in-betweens). De tillverkades av i huvudsak två grupper av animatörer med olika lön: chefsanimatörer och "inbetweeners". Sedan fanns det en ännu lägre grupp som fyllde i färger etc. Huvudbilderna görs för ytterlighetslägen (t ex sänkt arm resp höjd).

Stop-motion animering

I vanlig film vill man ibland ha med levande varelser som inte finns (jätteapor, rymdvarelser, dinosaurier). En teknik som använts sedan 1900-talets början är stop-motion animering. Den innebär att en konstgjord ledad modell i naturlig eller förminskad storlek byggs och förses med passande kroppsvolym. Man tar sedan en bild i taget och ändrar på ett eller annat sätt i modellen mellan tagningarna. Ett känt exempel är King Kong (1933). Ett mindre känt Lost World (1925), som hade dinosauriescener. Dinosaurieinslagen i Jurassic Park (1993) var tänkta att göras med denna teknik och man byggde mekaniska fullskalemodeller. Men under arbetets gång fick datoranimering ersätta flera av scenerna. Dock användes rörelsen (styrd av stop-motion-animatörer) hos en del mekaniska modeller som indata till animeringsprogrammet

1. Cell animation är något annat. Se t ex `http://vcell.ndsu/nodak.edu/animations/home.htm`.

Disneys animeringsregler

Som belysning av att animering är en konst snarare än en teknik.

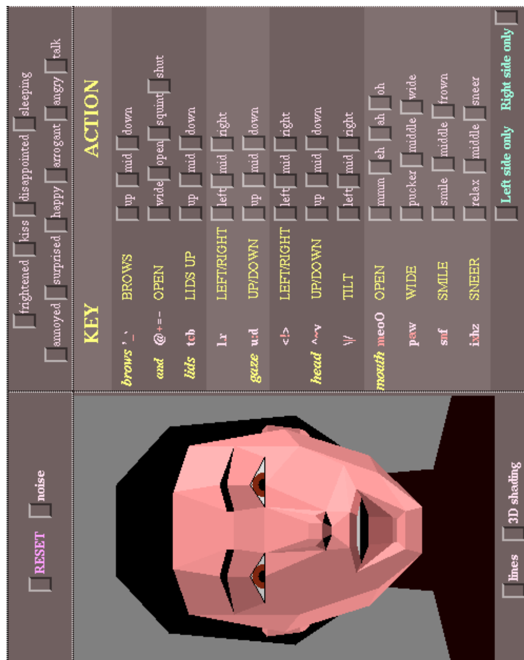
Hämtade från John Lasseter: Principles of Traditional Animation Applied to 3D Computer Animation, SIGGRAPH'87, där han exemplifierar med bl a sin egen film Luxo jr.

John Lasseter var animatör hos Walt Disney och gick senare till Pixar och gjorde en serie omtalade datorgenererade kortfilmer (bl a Luxo jr och Tin Toy). Regisserat Toy Story.

1. **Squash and Stretch** ("tryck ihop och sträck ut"): De flesta objekt deformeras under rörelse.
2. **Timing** ("timing"): Se till att saker och ting tar lagom tid, varken mer eller mindre.
3. **Anticipation** ("förväntan"): Varje handling föregås av en inledande fas. T ex om man skall ta ett föremål börjar man med att rikta blicken mot det.
4. **Staging** ("scensättning"): Se till att åskådaren har blicken där handlingen sker. Koncentrera handlingen till en sak i taget.
5. **Follow Through and Overlapping Action**: Handlingar har alltid en avslutande fas, t ex svänger en ben litet efter det att man sparkat iväg en boll.
6. **Straight Ahead Action and Pose-To-Pose Action**:
7. **Slow In and Out**:
8. **Arcs**: Rörelse längs krökta kurvor snarare än rätlinjigt.
9. **Exaggeration**: Överdriv utan att det blir orealistiskt så fattar åskådaren lättare.
10. **Appeal**: Gör det trevligt och nöjsamt.

Ansiktsanimering

Ett intressant område (eng. facial animation) med aktivitet. Även av intresse för enklare bildtelefoni (ansiktstextur överförs en gång!)

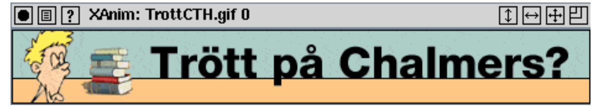


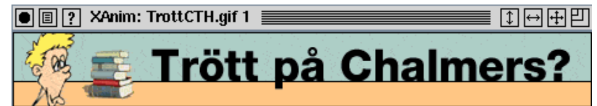
Bilden hämtad från `http://mrl.nyu.edu/~perlin/facedemo/`. Systemet skrivet helt i JavaScript och Java.

GIF-animering

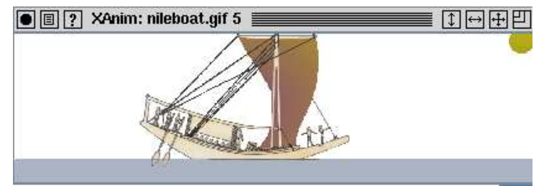
GIF är ett format för lagring av bilder. Det tillåter att man staplar ett antal bilder på varandra som återges en efter en. Används ofta för enkel webb-animering. Webbläsarna, t ex Mozilla, har inbyggt stöd. Fungerar bra för mindre bilder. Enkla medel kan ge påtaglig effekt.

Exempel: Blinkande person. Två inledningsbilder ur en sekvens.





Exempel: Seglande nilbåt



Flash-animering

Flash är också ett format för enklare grafikanimering via nätet. Läsaren Flash Player (insticksprogram till t ex Mozilla) är fri, medan produktionsprogrammet Flash kostar. Används på vissa av Chalmers officiella sidor. Borde finnas installerat tillsammans med Mozilla hos oss enligt StuDAT-specifikationen, men tycks inte göra det. Ute finns åtminstone version 7, men vi har vad jag kan se bara version 5 (Sun-miljön) respektive version 6 (2004 års Linux-miljön). I version 7 kan bl a ett C/Java-liknande skriptspråk användas, vilket gör att riktig 3D-grafik kan åstadkommas. SVG (Scalable Vector Graphics) är ett format, som jag tror har ambitionen att tävla med Flash. Kommer från WWW-konsortiet W3C. Vår Mozilla har stöd för SVG.

Exempel: Hoppande gubbe som rör sig mot en rörlig bakgrund.

Prova själv: Flytta dig till mappen `$DG/BILDER/FLASH`. Skriv `flashp6` (kollad 2005) och öppna någon av filerna `Cyber-Calculator-weltin.swf` och `Xmas-Elf-steele.swf` (ovanstående). Man får en del felutskrifter som beror på att jag inte installerat flashplayer enligt alla konstens regler. Filerna borde - men det går inte nu - också kunna betittas med en webbläsare.

Tidslinjeanimering

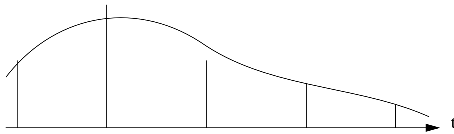
Låt oss anta att en scen består av N st objekt. Ett vanligt användarsnitt ser ut ungefär så här:

Objekt	Bild 1	Bild 2	Bild 3	
1				
2				
N				

I princip skulle rutorna innehålla aktuella data (positioner och ev hastigheter m m) för objekten och fyllas i manuellt. Men i praktiken manipuleras objekten i stället med t ex musens hjälp. Det är svårt att få naturlig rörelse.

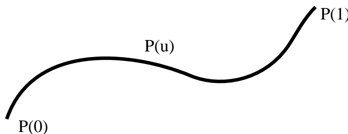
I de avancerade programmen finns mer sofistikerade sätt att styra rörelserna. Liksom vid traditionell animering används huvudbilder (borde väl nu heta huvudscener) för vilka fullständiga data finns (inkl hastigheter direkt eller indirekt). Variationerna modelleras med kurvor (B-splines eller motsv).

Tidslinje



Stelkroppsrörelse

I enklaste fallen handlar det om att flytta ett föremål längs en kurva eller rotera det kring någon punkt eller axel. Om kurvan är given på parameterform
$P = P(u)$, $0 \le u \le 1$, med P(0) och P(1) givna

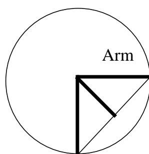


är det frestande att generera t ex 9 st mellanbilder baserade på positionerna P(0.1), P(0.2), ..., P(0.9) med tanken att rörelsen då blir jämn. Men det är inte självklart att parametern u har med kurvlängd att göra, varför någon form av reparametrisering kan behövas. Dessutom skall rörelsen i allmänhet inte vara jämn. Rörelsen skall kanske först accelerera från vila och på slutet bromsas. Detta kan naturligtvis i vissa fall vara inbyggt i parameterframställningen genom att u helt enkelt står för tiden.

Enkla fall: Fritt fall, en kanonkulas rörelse, en studsande boll.

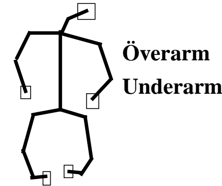
Objekt som deformeras: Om vi har en parameterframställning för deformation, så kan detta hanteras på samma sätt. T ex en boll som som krymper eller förvandlas till en ellipsoid (låt radien resp halvaxlarna vara parametrar).

Linjär interpolation: Är sällan det rätta sättet. Kan ge oönskade effekter, t ex en arm som sänks från horisontellt till vertikalt läge.

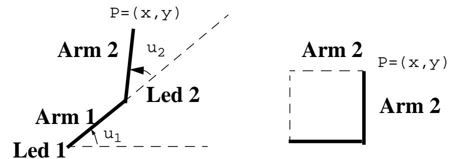


Framåtkinematik

Objektskelett: Levande varelser är inte stela objekt. Förenklat kan man se dem som uppbyggda av ben (kallade armar i figurerna), som är hopkopplade med leder och senare kläs med (ev deformerbara) volymer. För studiet och beskrivningen av rörelsen kan skelettet duga.



För att belysa problemets karaktär låt oss se på ett objekt med bara två leder och ben och med enbart två frihetsgrader (i verkligheten upp till sex frihetsgrader per led i 3D; tre rotationer och tre translationer). Totalt sägs en människokropp behöva 200 frihetsgrader.



Vid framåtkinematik utgår man från förändringar i den interna strukturen - i vårt fall de två vinklarna u_1 och u_2 - och ser hur dessa påverkar ett ändläge P=(x,y). Vi ser av högra figuren att det kan finnas flera arrangemang som ger samma ändläge. Om vi för enkelhets skull låter armarna ha samma längd:

$$x = L(\cos(u_1) + \cos(u_1 + u_2))$$
$$y = L(\sin(u_1) + \sin(u_1 + u_2))$$

Kinematik, invers kinematik

Vid invers kinematik utgår man i stället från en eller flera ändpunkter och vill bestämma de vinklar (etc) som behövs för att nå dessa. Det blir alltså fråga om att lösa ett högeligen olinjärt ekvationssystem F(**u**)=**x**, där nu **u** och **x** är vektorer. Det kan finnas 0, 1 eller flera lösningar.

I vårt exempel kan man med litet möda lösa ut u_1 och u_2 uttryckta i x och y, t ex

$$u_2 = \arccos\left(\frac{x^2 + y^2 - 2L^2}{2L^2}\right), u_1 = \arctan\left(\frac{-x\sin u_2 + y(1 + \cos u_2)}{y\sin u_2 + x(1 + \cos u_2)}\right)$$

men detta är givetvis inte möjligt i allmännare fall. I praktiken måste man använda någon approximativ metod, t ex Newtons metod (eller ta till något annat trick/approximation)

$$J(\mathbf{u}^{(k)})\mathbf{u}^{(k+1)} = J(\mathbf{u}^{(k)})\mathbf{u}^{(k)} + \mathbf{x} - F(\mathbf{u}^{(k)})$$

där J är Jacobimatrisen (även kallad funktionalmatrisen). För varje iterationssteg har man alltså att lösa ett linjärt ekvationssystem. Men det kan uppstå en del problem.

Invers kinematik är en stor sak även inom robotteknik.

Fysikbaserad kinematik

Kinematik är en ofullständig modell. Man kan göra en matematisk modell för objektet, som tar hänsyn till massan hos de olika delarna och friktion och tröghet i lederna, liksom ev pålagda krafter. Rörelsen styrs av lagen kraften=massanxaccelerationen. En fysikalisk princip säger att rörelsen sker så att arbetet är minimalt. Man förvandlar alltså rörelseproblemet till ett matematiskt optimeringsproblem innehållande en differentialekvation
mx”(t)=F(t).
Det är hanterbart.

Verklighetsbaserad kinematik

En tidig ide vid animering var att hämta rörelsedata från verkliga varelser. I början av 1900-talet uppfanns rotoskopet, som innebar att man ritade av förinspelat material. Denna teknik skall ha använts för animeringen av Snövit (men inte dvärgarna) i Disneys Snövit och de sju dvärgarna (1933). Senare har man använt sensorer placerade på ett antal strategiska ställen på en kropp. I Jurassic Park användes denna teknik i ett par fall, men man utgick från de mekaniska dockor som från början skulle använts genomgående. Dockorna användes alltså för rörelsen. Hud och liknande var även i dessa fall datorgenererad.

Datorgenererade objekt i reell värld eller vice versa

Handlar om att placera ett datorgenererat objekt i en en filmad värld eller en filmad varelse i en datorgenererad värld. Detta är huvudsakligen en filmteknisk fråga.

Exempel (`BILDER/GENEVA.mov`): En artificiell Marilyn Monroe vandrande utmed Geneve-sjön.

Morfing

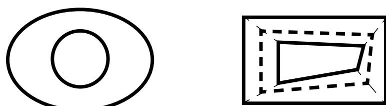
Problem: Låt ett objekt övergå i ett annat (formförändring). Effekten blir mest intressant när de två objekten är väsentligt olika, t ex en bil blir en tiger (se Hearn/Baker). Man kan tänka sig processen utspelad i 2D eller 3D. Effekten gjordes populär med filmer som Terminator II (199x) och Indiana Jones (199x). Kan ses som en form av animering.

Parametrisk morfing: Om objekten kan beskrivas med en och samma ekvation men med olika parametervärden kan vi lätt åstadkomma övergången. T ex om en ellips

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

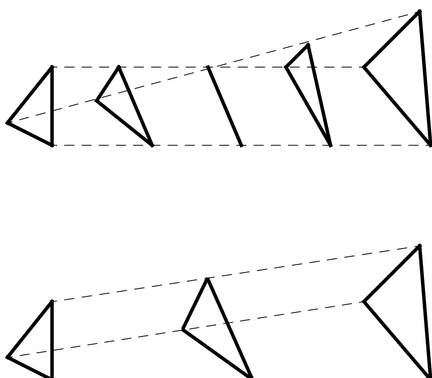
Morfing, forts

med a=3 och b=2 skall omformas till en cirkel med radien 1, så kan vi bilda en följd av ellipser genom linjär interpolation mellan a=3 och a=1, b=2 och b=1.



Liknande om en rektangel skall omformas till en fyrhörning

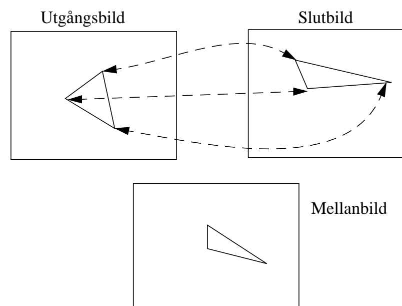
För att parametrisk morfing med linjär interpolation skall ha en chans att lyckas måste hörnen paras ihop lämpligt. Jfr följande två figurer med identiska start- och slutobjekt. I första figuren urartar förloppet, vilket är mindre lyckat.



Bildmorfing

Låt oss se på ett specialfall där det gäller att få en bild att övergå i en annan under viss tidsperiod. Det enklaste sättet är att bara tona över från den ena bilden till den andra (eng. cross-dissolve, sv. övertona). Detta är ett gammalt trick i filmindustrin (jag kommer ihåg det från filmatiseringen av R.L. Stevensons Dr Jekyll och Mr Hyde med en ung Spencer Tracey). Men då framgår inte en eventuell formförändringen så väl.

Vi beskriver ett möjligt sätt. Om det gäller två ansikten så är idén att även om näsorna är olika placerade och stora så skall näsa övergå i näsa mjukt.



En mellanbild skapas så här:

1. Trianglarna interpoleras från motsvarande i utgångs- och slutbilderna.
2. Fyll i en punkt (x,y) = (u,v) i barycentriska koordinater (se kommande avsnitt om Beräkningsgeometri) enligt $I(u, v) = (1-t)I_{start}(u, v) + tI_{slut}(u, v)$

Vi får kontinuitet längs kanterna men inte mer. Professionella program arbetar annorlunda.

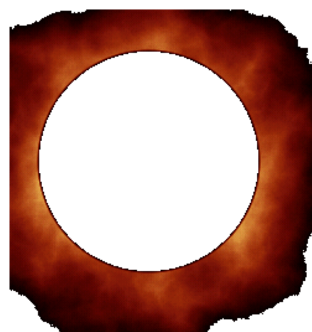
Tekniker vid återgivning i realtid

1. Dubbelbuffring är nödvändig.
2. Tidsstyrning. Förloppet skall gå lika fort oberoende av dator. Se avsnitt 17 i OpenGL-häftet.
3. Vid resursbrist hoppa över bilder eller sänk bildkvalitén.
4. Rasterkopiera i stället för att rita om en komplicerad bakgrund.
5. Använd sprite-teknik, dvs lägg små objekt ovanpå annat.

Effekter, naturfenomen

För detta finns ett stort antal tekniker. Här kommer början på en uppräkning.

1. Partikelsystem. Mer om detta separat.
2. Lösning med brus. I fallet Perlin-brus utökar man bara med en dimension för tiden, dvs 2D-fallet övergår i 3D och 3D i 4D. Knappast realtidsmetod. Bilden visar eldsflammor kring ett klot (svart i ursprungsfilen). Finns som `Flames500.gif` och körs med t ex någon webbläsare Hämtad från Perlins `www.noisemachine.com`..

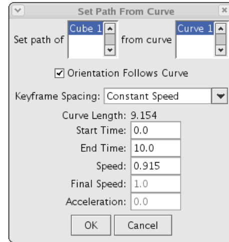


3. Plakat (bill-boarding). Tas upp senare.
4. Visa förloppet som en förinspelad GIF-animering (eller motsv).

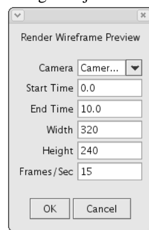
Animering i Art of Illusion 1(6)

Vi vill belysa hur datoranimering kan gå till praktiskt och väljer i år att arbeta i *Art Of Illusion*. För detaljer hänvisas du som vanligt till manual och handledningar (tutorials). I *AoI* finns tre huvudsätt: rörelsen definierad av bana, rörelsen definierad av ett antal situationer och rörelsen definierad av formler.

Exempel 1 (`OH_AOI1.aoi`): Vi börjar med att rita kurvan med något av de två kurvverktygen (avsluta kurvan med ENTER). Sedan skapar vi en kub med kubverktyget (kan placeras var som helst; det är mittpunkten som kommer att följa kurvan). Det är praktiskt att reducera storleken på AOI-fönstret, så att ingen del döljs av annat, t ex systemmenyn. Vi ser till att både kurva och kub är valda och väljer **Animation/Set Path from Curve**, som ger en dialog.

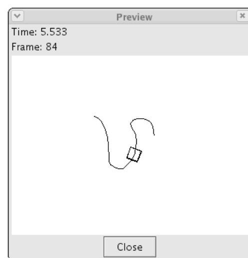


Vi ändrar **End Time** till 10.0. Trycker på **OK** och begär en förhandsvisning med **Animation/Preview Animation**, som ger följande dialog.

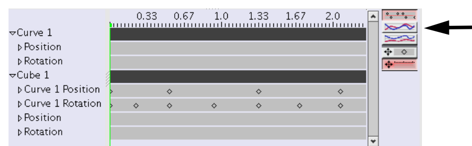


Vi trycker på **OK** efter ha granskat värdena. Detta gör att det dyker upp ett fönster som visar animationen.

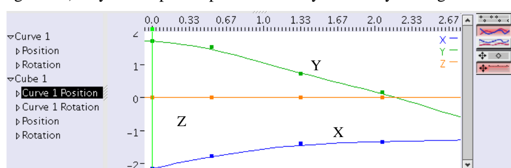
Animering i Art of Illusion 2(6)



Vi tar fram animeringens partitur (eng. score) med **Animation/Show Score**.



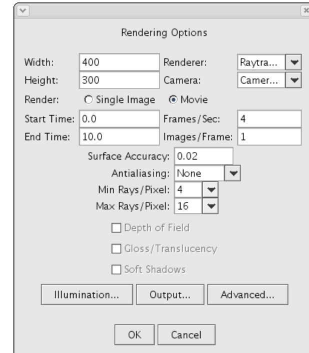
Överst finns en tidslinje med sekunder som enhet. Nedtill markerar små romber (eng. diamonds) vid vilka tidpunkter som nyckelbilder skapas (dessa motsvarar styrpunkter hos vår utgångskurva). Trycker vi på den pilmarkerade symbolen byter diagrammet utseende.



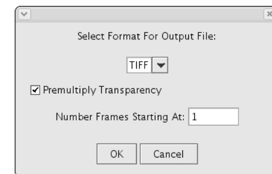
Kurvorna visar hur kubens position varierar med tiden (tidslinjen överst). Det finns en kurva för var och en av de tre koordinaterna X, Y och Z. Vi ser att kurvan för Z-koordinaten är z = 0, vilket naturligtvis beror på att vår bankurva är 2-dimensionell. Y avtar som väntat inledningsvis, medan X ökar svagt. Med den understa symbolen till höger kan vi flytta diagrammet. De fyrkantiga punkterna (som motsvarar styrpunkterna för den

Animering i Art of Illusion 3(6)

ursprungliga kurvan) fungerar som nyckelbilder. Dessa kan flyttas genom att vi väljer den näst understa symbolen till höger och sedan använder MK1 och drar. De kan flyttas i både tidsled (horisontellt) och rumsled (vertikalt). En flyttning kan i det här fallet leda till att vi hamnar utanför den avsedda banan. En punkt kan också redigeras genom att den markeras och man väljer **Animation/Edit Keyframe**. MK3 flyttar diagrammet. Kurvorna är lokala polynom (splines), dvs ändringar brer inte ut sig.
Nästa steg är att göra den slutliga animeringen, för vilket vi använder
Scene/Render Scene.

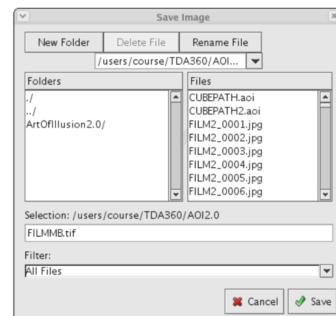


Vi markerar **Movie** och fyller i sluttid samt önskat antal bilder per sekund och får efter OK-et en ny fråga



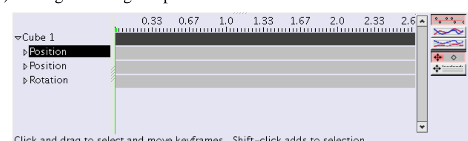
Animering i Art of Illusion 4(6)

Jag ändrar bildformatet till TIFF (PNG hade också gått bra). Sedan dyker det upp en fildialog.



Javas fildialoger brukar vara långsamma vid musklick. Själv använder jag ENTER-tangenten efter att gjort en markering.
Nu skapas ett antal bilder kallade `FILMMB0001.tif` till och med `FILMMB0040.tif` (alla med svart bakgrund även om jag väljer transparens). De får sedan fogas ihop till en film med något lämpligt program (se nedan).

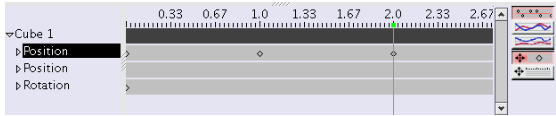
Exempel 2 (`OH_AOI1.aoi`): Återigen animerar vi en kub i rörelse. Denna gång genom att vi anger tre punkter för banan. Vi skapar en liten kub med kubverktyget och placerar den (med MK1) i övre vänstra delen av Front-fönstret. Ser till att kuben är vald. Med **Animation/Add Track To Selected Objects/Position/XYZ** (**One Track**) skapar vi ett spår (eng. track). Vi begär visning av spåret med **Animation/Show Score**.



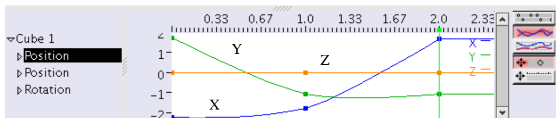
Överst syns tidslinjen. Tidslinjemarkören (ett grönt streck med en klump upptill) finns vid tiden 0. Vi vill stoppa in en nyckelbild vid denna tid, vilket görs med

Animering i Art of Illusion 5(6)

Animation/Keyframe Modified Tracks of Selected Objects. Denna nyckelbild utgörs av kuben i sitt utgångsläge. Vi vill också skapa nyckelbilder vid t = 1 och t = 2. Med MK1 flyttas markörhandtaget (det gäller att pricka det rätt) till t = 1. Vi väljer flyttningsverktyget högst upp i AoI-fönstret. Sedan flyttar vi med MK1 kuben nedåt i Front-fönstret. Sedan **Animation/Keyframe Modified** .... igen. Vi gör motsvarande för t = 2, men flyttar denna gång kuben åt höger.



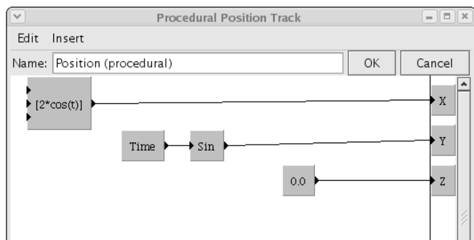
Vi ser nyckelbildssymbolerna. Genom att klicka på den andra av symbolerna till höger får vi förloppet i form av kurvor.



I Front-fönstret är x-axeln horisontell och y-axeln vertikal, som betyder att den inledande vertikala rörelsen minskar y och lämnar x (i stort sett) oförändrad, vilket bekräftas av kurvorna. Med **Animation/Preview Animation** kan vi som tidigare se hur animeringen tar sig ut: kuben faller under den första sekunden och drar sig sedan åt höger. Rörelsen avslutas vid t = 2, men animeringen fortsätter ytterligare 8 sekunder.

Animering i Art of Illusion 6(6)

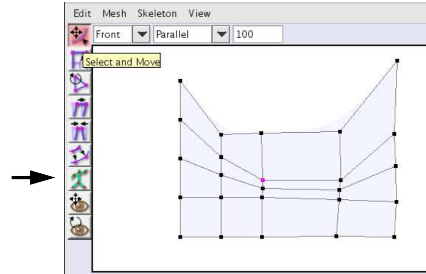
Exempel 3 (`OH_AOI3.aoi`): Det finns ett tredje sätt i *AoI* att beskriva en animering. Det utgår ifrån att man har en formel för rörelsen, vilket är vanligt i t ex simuleringssammanhang. Om vi vill att en sfär ska snurra i en elliptisk bana med halvaxlarna 2 resp 1 runt origo i xy-planet, kan vi göra som följer. Skapa en sfär och se till att den är vald. Välj **Animation/Add Track To Selected Objects**. Tag fram partituret med **Animation/Show Score**. I spåret se till att **Position (Procedural)** är vald. Vi vill redigera den valda egenskapen och tar till **Animation/Edit Track**. Redigeringen görs med samma blockteknik som vi använde för procedurtexturer. Efter redigeringen ser det ut så här:



Och vi kan betitta animeringen. Några kurvor i parturet tycks inte genereras när man arbetar med formler.
Vi har avsiktligt valt enkla objekt i dessa tre exempel och valt att arbeta med positioner. Man kan dessutom beskriva bl a rotationer.

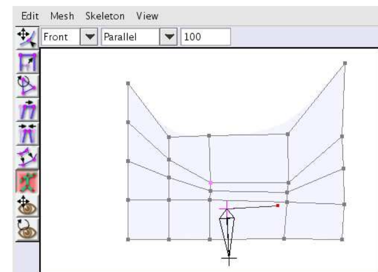
Kinematik i Art of Illusion 1(3)

Även för detta ett mycket enkelt exempel (`OH_AOI4.aoi`): som belyser arbetssättet. Realistiska exempel blir pillriga. Beträffande viss terminologi se tidigare OH. I *AoI* utgår man från ett splinenät (eng. spline mesh), till vilket man kopplar ett skelett (eng. skeleton). Skelettet består - oberoende av vilka benämningar som vi använt tidigare - av leder (eng. joint) (ibland kanske kallade knutar) och ben (eng. bone). Vi använder splineverktyget (mellersta i vänstra kolumnen) och skapar med MK1 ett standardnät i xy-planet. Påbörjar redigering av nätet med **Object/Edit Object**.
Jag flyttar med MK1 om styrpunkterna i Front-delen så att blir något i stil med



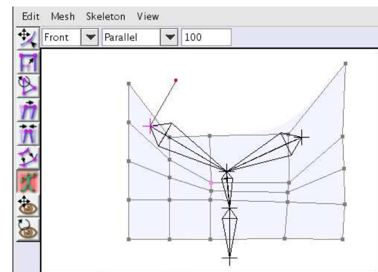
Nu skall skelettet in, vilket fixas med skelettverktyget på plats 3 nedifrån. Jag skapar ett ben genom att först markera nedre krysset (första leden) i figuren med CTRL-MK1 och sedan det övre skära krysset (andra leden) med CTRL-MK1. Nu ser det ut så här:

Kinematik i Art of Illusion 2(3)



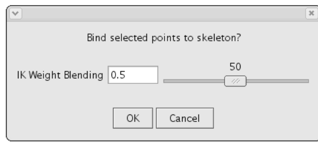
Vi ser ett rött handtag kopplat till den övre leden. Med det (MK1) kan vi vrida benet; än så länge utan att nätet påverkas. Den första punkten kommer att fungera som rot för vårt skelett.

Vi vill lägga till ytterligare ben. Ett ben läggs alltid till den skära punkten (man gör en punkt skär genom att klicka på den med MK1). Jag lägger en knut något högre upp med CTRL-MK1 och en till höger med CTRL-MK1. För att kunna lägga till den vänstra leden i figuren nedan gör jag först utgångspunkten skär med MK1 (om den inte redan är det).

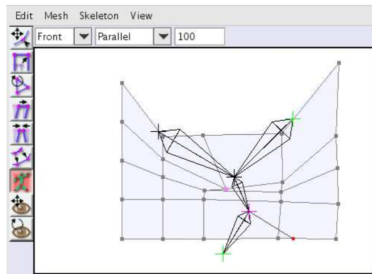


Kinematik i Art of Illusion 3(3)

Vi kan rotera det vänstra benet med handtaget (MK1) i yttersta leden. Ett ben rör sig alltid runt sin moderled. Nu skall vi koppla ihop skelettet med nätet, vilket görs med **Skeleton/ Bind Points to Skeleton**.



Vi funderar inte närmre på det föreslagna värdet utan trycker bara på **OK**.
Nu kan vi röra på ben på det vis som beskrivits och nätet påverkas (i just detta exempel dock inte så tydligt som vore önskvärt). Vi kan markera förgreningsleden (med MK1) och sedan röra hela övre delen av skelettet med den ledens handtag. Detta är **framåtkinematik**.
Vi kan också begära att visa leder skall hållas fixerade. Detta görs med SHIFT-MK1, som gör krysset grönt. I figuren nedan är roten och övre högra leden grönmarkerade (fixerade). Därefter kan vi röra skelettet genom att med MK1 trycka på en led (inte på handtaget) och dra den.
I figuren nedan har jag rört på förgreningsleden. Detta är **invers kinematik**.



Film från en bildföljd

Vi har sett att en del program producerar animeringen i form av en bildföljd. Denna måste i så fall göras om till en film av ett fristående program. Det finns många sådana. Utformatet kan vara bl a MPEG, AVI (ev med DivX) eller Microsofts WMV.
Själv använder jag ett uråldrigt UNIX-program *mpeg_encode* (åtkomligt direkt under Linux om man gjort `setup_course TDA360`), som tillverkar en MPEG-film utifrån bl a TIFF-, PNG- eller XWD-bilder. Bränns den sedan på en CD eller DVD kan filmen betittas i en TV kopplad till en DVD-läsare eller spelas upp med *gmplayer*/*mplayer* (se separat). Det programmet används så här:
`mpeg_encode P.par`
där `P.par` är en parameterfil som innehåller information om indata och utdata m m. Det är inte alldeles självklart vilka värden man skall ange för de olika parametrarna. Jag kan lägga upp en mall om någon är intresserad. Ett annat program *ppmtompeg* fungerar på ett likartat sätt.
Rasmus Anthin tipsade om ett fritt PC-program *Virtual Dub*, som kan användas för detta ändamål. Det finns många kommersiella.

Uppspelning av animeringar

Programmet *mplayer*/*gmplayer* finns installerat under Linux (dock inte via menyerna). Man kan spela t ex en enstaka .mpeg eller .vob-fil (ej kopieringsskyddade) enligt modellen
`mplayer dvd://N VTS_01_1.VOB`
(`dvd://N` enbart om filen på DVD-skiva) eller starta
`gmplayer`
som ger GUI. Tryck på MK3 och välj i menyn, varefter man får ett för oss alla numera välkänd typ av panel.

Programmet klarar en massa andra format, t ex AVI och MOV (Quicktime). Programmet finns även i vår Windows-miljö om man letar bland kursmapparna.
I mappen `$DG/BILDER` finns en del animationer.

Animering från OpenGL

Om vi vill göra en film från OpenGL kan vi låta uppdateringsmetoden producera en bild på t ex PNG-format vid varje anrop och sedan bilda en MPEG-fil som ovan antytts. Ett krux är möjligen att hitta kod som tillverkar bilden. Ett värre krux är förmodligen att bilderna inte tillverkas i helt jämn takt och därför borde tidsmärkas.

Ett enkelt sätt att få fram bilderna under Linux är att använda xwd-kommandot inifrån programmet. Detta X-kommando tar en s k fönsterdump och lagrar den på XWD-format. Man kan i parameterfilen till *mpeg_encode*/*ppmtompeg* tala om att .xwd-filerna skall omvandlas till .ppm-filer som dessa program utgår ifrån. Som vanligt kommando skriver man:
`xwd -name fönsternamn -out bildfilnamn`
I ett OpenGL-program gör man så här:

1. Inför två globala variabler
```
static int antal = 0;
static char command_string[] = "/usr/X11R6/bin/xwd -name fffffffff
-out nnnn";
```
2. I uppdateringsmetoden (klarar bilder 01 till 99)
```
antal = antal + 1;
if (antal < 10 ) {
      sprintf(command_string,"/usr/X11R6/bin/xwd -name MyWind
                             -out   bb0%d",antal);
} else {
     sprintf(command_string,"/usr/X11R6/bin/xwd -name MyWind
                            -out bb%2d",antal);
}
system(command_string);
```

Med denna kod får vi bilder *bb01* till *bb99*.

Animering i MATLAB

Borde man säga något om men utrymmet är knapert, så det får räcka att man kan dels skapa interna animationer, dels göra om en sådan till AVI-format (Audio/Video Interleaved).

Animering i Blender och PovRay

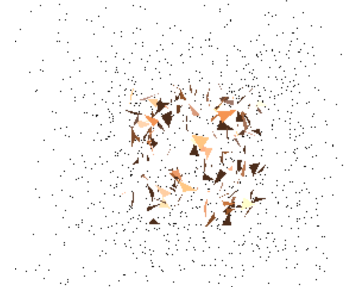
Går utmärkt. Blender ungefär som AoI.

Partikelsystem

Ett partikelsystem utgörs av ett antal partiklar och regler för dessas rörelser och andra beteenden. Partiklarna kan i enklaste fallet vara punkter, men också andra mindre objekt är tänkbara. Det kan t ex finnas föreskrifter om en viss hastighet eller acceleration. Objekt kan eventuellt skapas och dö bort successivt. Eventuellt pågår fenomenet bara så länge "bränslet" (i enklaste fall en varvräknare) räcker. Vill man att det skall gå långsammare kan man låta förloppet styras av fysikaliska lagar.

Naturliga företeelser som kan modelleras med partikelsystem är fyrverkerier och andra explosioner. Men även t ex eld, rök och damm.

Exempel: Gustav Taxéns program `DEMOS/EXPLOSION.c`. När man trycker på mellanslagstangenten sker en explosion som yttrar sig i att dels ett tusental punkter far iväg radiellt från origo i en slumpmässig riktning och med en slumpmässig hastighet, dels att ett antal slumpmässigt skalade trianglar gör likaledes samtidigt som de snurrar. Bilden gör inte programmet rättvisa.



I programmet är trianglarna belysta medan punkterna bara ritas med sin färg. Programmet är uppbyggt som ett typiskt sådant program.

1. Skapa objekten.
2. Låt en idle-procedur räkna ut nya positioner etc och anropa sedan via `glutPostRedisplay` uppdateringsproceduren, som ritar den nya situationen. I just detta fall beräknas nya position genom att man adderar partikelns hastighet.

Exempel: Skärmlåsningsprogrammet *xlock* med `xlockMB -mode pyro`

Modifierat utdrag ur kod för EXPLOSION.c

```
/* Uppdateringsproceduren som ritar om scenen */
void display (void) {
  int    i;
  // ... Sudda och placera kamera
  if (fuel > 0)    {
    glPushMatrix ();
    glDisable (GL_LIGHTING); glDisable (GL_DEPTH_TEST);
    glBegin (GL_POINTS);      // PUNKTERNA
         for (i = 0; i < NUM_PARTICLES; i++) {
            glColor3fv (particles[i].color);
            glVertex3fv (particles[i].position);
         }
    glEnd ();
    glPopMatrix ();
    glEnable (GL_LIGHTING); glEnable (GL_LIGHT0);
    glEnable (GL_DEPTH_TEST);
    glNormal3f (0.0, 0.0, 1.0);
    for (i = 0; i < NUM_DEBRIS; i++) {  // TRIANGLARNA
      glPushMatrix ();
      glTranslatef (debris[i].position[0],
                    debris[i].position[1],
                    debris[i].position[2]);
      glRotatef(debris[i].orientation[0],1.0,0,0);
      glRotatef(debris[i].orientation[1],0.0,1,0);
      glRotatef(debris[i].orientation[2],0.0,0,1);
      glScalef (debris[i].scale[0], debris[i].scale[1],
                debris[i].scale[2]);
      glBegin (GL_TRIANGLES);
          glVertex3f (0.0, 0.5, 0.0);
          glVertex3f (-0.25, 0.0, 0.0);
          glVertex3f (0.25, 0.0, 0.0);
      glEnd ();
      glPopMatrix ();
    }
  }
  glutSwapBuffers ();
}
```


Flockar, boids

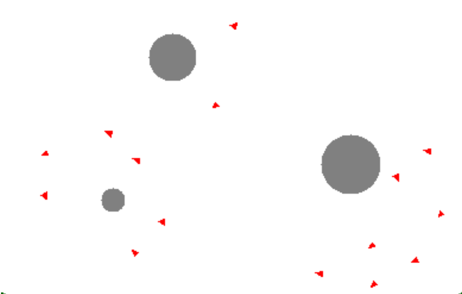
I en del situationer har man ett flockbeteende. T ex fåglars flykt och fiskstim. För detta finns det engelska begreppet boids (av bird-oid), som infördes av Craig Reynolds 1987. Ingår i något man kallar **Artificial Life**.

Han upptäckte att man kan få naturlig flockning med hjälp av tre enkla regler:

1. Separation: Se till att inte komma för alltför nära andra boids eller hinder i scenen.
2. Gemensamt mål (eng alignment): Eftersträva samma hastighet och färdriktning som de boids som är nära.
3. Grupp (eng cohesion): Styr mot masscentrum för de boids som finns i närheten.

Varje boid beter sig individuellt, dvs det finns ingen flockregel. Med N boids blir problemet av komplexiteten $O(N^2)$, men med lagom approximationer kan man få flockar att röra sig i realtid. Man kan fortfarande tala om ett partikelsystem men med tilläggsregler.

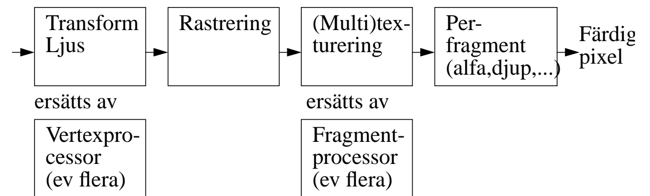
Här visas en (något tryckmodifierad) av många appletar. http://www.taygete.demon.co.uk/java/flock/ (ur funktion 2003/4)



Referenser med många länkar: `http://www.red3d.com/cwr/boids/` och `http://www.red3d.com/cwr/boids/applet/`

Vertex- och fragmentprogrammering, allmänt 1(2)

är begrepp som stöds av NVIDIA, ATI och Microsoft (DirectX) och nu även OpenGL i något olika varianter. Vi belyser det en aning, dels för nyhetsvärdets skull, dels för att vi kan få ökad förståelse. För detaljer hänvisas till teknisk dokumentation. Engelska benämningar vertex programming eller vertex shading, respektive pixel programming, pixel shading eller fragment programming/fragment shading. Beskrivningen görs enbart från ett OpenGL-perspektiv. Följande bild visar OpenGL:s rörledning kraftigt förenklad. Den matas från vänster med bl a hörnkoordinater (i modellkoordinatsystemet). Dessa transformeras i första steget till hörn i homogena projektionskoordinater med bl a tillhörande uträknade ljusvärden. Senare kommer textureringssteget, som vi vet kan modifieras en hel del.



Dessa två steg är i nyare grafikprocessorer programmerbara. Användaren har hittills förväntats skriva program för dem i ett maskinkodsspråk. Nivån höjs en hel del med C-liknande språk som Microsofts HLSL (High Level Shading Language) och NVIDIAS Cg, och OpenGL Shading Language (GLSL). För att öka prestanda har en del grafikprocessorer flera vertex- och fragmentprocessorer som arbetar parallellt. ARB införde först begreppen vertexprogram och frag-

Vertex- och fragmentprogrammering, allmänt 2(2)

mentprogram (utvidgning 2002) som skrivs i ett språk ligger mellan maskinspråken och de nämnda högnivåspråken. Senare - 2003 - kom ARB med utvidgningarna “vertex shader” och “fragment shader”, vilka skrivs i väsentligen GLSL. Hösten 2004 fördes dessa in i OpenGL 2.0 med några mindre namnförändringar. Fr o m hösten 2005 kan vi här använda de riktiga 2.0-namnen. Jag har svårt att hitta någon bra översättning av “shader” och kommer därför att använda benämningarna vertexprogram och fragmentprogram i stället för de korrekta.

Som namnen antyder arbetar ett vertexprogram på ett hörn (vertex), medan ett fragmentprogram arbetar på de fragment (pixlar) som kommer ut från rastreringsetappen. För **varje** hörn respektive fragment kommer motsvarande program att genomgås.

Fragmentprogrammering gör att t ex Phong-toning (se belysningsavsnittet) kan åstadkommas. Varje pixel kan påverkas genom kombination av en mängd texturer. Man har använt fragmentprogram för avancerade ljusberäkningar, procedurtexturer, fraktalbilder, etc.

Innan de två stegen blev programmerbara utvidgades de med ett stort antal speciallösningar för diverse problem, vilket ledde till oöverblickbarhet. Förmodligen var stegen internt “programmerbara” redan då. Nu blir det tydligare för grafikprogrammeraren vad som är möjligt.

Det som sägs här bygger på bl a specifikationstexterna för OpenGL 2.0 (c:a 400 sid) och The OpenGL Shading Language (c:a 100 sid). I den senare beskrivs språket för vertex- och fragmentprogram, medan kopplingen mellan OpenGL och dessa program behandlas i specifikationen för OpenGL 2.0. Länkar till mer dokumentation på kurssidan.

OpenGL Shading Language (GLSL) 1(3)

är det språk i vilket vi skriver våra vertex- och fragmentprogram.

Språket är avsiktligt C-liknande med ett antal nya datatyper, bl a för beskrivning av vektorer. Vi kan bygga upp programmen modulärt genom att skriva egna funktioner. Programmen startar alltid sin exekvering med funktionen *main*. I funktionerna kan vi införa egna lokala variabler. Det finns ett stort antal fördefinierade namn, dels funktioner, dels variabler som är globala i programmen (vissa av dessa är bara läsbara eller bara skrivbara). Dessutom kan vi införa egna globala variabler.

GLSL innehåller de normala styrkonstruktionerna som if, for och while, return samt discard (avbryter exekveringen av ett program och allt blir ogjort). Hårdvaran/drivrutinerna för våra 6600-processorer har stöd för hela GLSL (utom noise-funktionerna). I praktiken finns en del andra restriktioner (begränsad programlängd, begränsat variabelutrymme), som gör att en del programidéer stöter på patrull. Men utvecklingen går vidare.

Ett program utgörs initialt av en sträng som sedan kompileras till en intern maskinkod. Vi kan skriva strängen direkt i vårt vanliga OpenGL-program, men vanligen läser man in vertexprogrammet från en fil och fragmentprogrammet från en annan (härigenom kan man få sitt program att bete sig annorlunda genom att bara ändra i filen). I frånvaro av endera typen av program utförs de normala stegen. T ex ett vertexprogram kan delas upp på flera filer, men det är ju något som rimligen bara är intressant för riktigt stora program, så det går vi inte in på.

OpenGL Shading Language (GLSL) 2(3)

Exempel på datatyper: `float`, `int`, `vec3`, `vec4`, `mat3`, `mat4`, ...

Vid **initieringar** används en **konstruktor**, t ex

```
vec3 color = vec3(1.0, 1.0, 0.0);
```

För vektorer kan utöver den vanliga notationen med [i] **.xyzw-notation** (rumskoordinater) användas liksom .rgba- (färger) och .stpq-notation (texturer). T ex ändrar `color.rg = vec2(1.0, 0.0)` eller `color.g = 0.0` eller `color[1]=0.0` den tidigare gula färgen till röd.

I C används **kvalifikatorn** `const` vid sidan av datatypen för att definiera en konstant. Samma gäller GLSL, t ex

```
const vec3 redcolor = vec3(1.0, 0.0, 0.0);
```

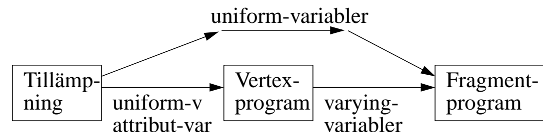
Dessutom finns ytterligare **kvalifikatorer** `uniform`, `attribute` och `varying`, vars uppgift framgår av nästa OH.

De olika matematiska operationerna skrivs som i C, med den skillnaden att de kan verka elementvis på vektorer. T ex `color1+color2`, `sin(color)`.
För skalärprodukt och vektoriell produkt finns funktionerna `dot` respektive `cross`. Ytterligare en mängd funktioner finns. Ett urval: `cos`, `tan`, `pow`, `exp2`, `log2`, `abs`, `inversesqrt` (ger 1/kvadratroten(...)), `length`, `min`, `max`, `normalize`, `reflect` (ger reflektionsvektorn givet ljusvektorn och normalen i vykoordinater), `noise1/2/3/4`. Vissa är i praktiken approximativa. Nvidia har ännu inte lyckats implementera `noise`-funktionerna.

För matriser `m` avser `m[i]` den i:te kolumnen!!!

OpenGL Shading Language (GLSL) 3(3)

Vår tillämpning, vertexprogrammet och fragmentprogrammet kan kommunicera. Hur visas schematiskt i följande bild.

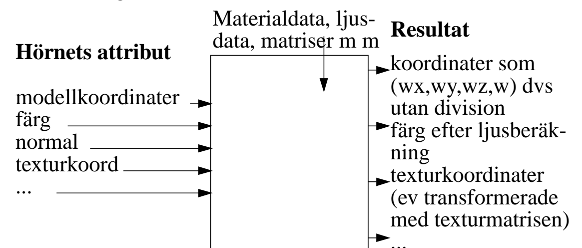


Vertexprogrammet		Fragmentprogrammet	
In	**Ut**	**In**	**Ut**
gl_Vertex	gl_Position	gl_Color	gl_FragColor
gl_ModelViewMatrix	gl_TexCoord[i]	gl_SecondaryColor	
gl_ModelViewProjectionMatrix	gl_FrontColor	gl_TexCoord[i]	
gl_LightSource[i]	gl_BackColor	gl_FrontMaterial	
gl_MultiTexCoord0-7		gl_BackMaterial	
gl_Normal		gl_LightSource[i]	
gl_NormalMatrix		...	
gl_Color			

En uniform-variabel deklareras i vertex/fragmentprogrammet och ges värde i tillämpningsprogrammet. Den är bara läsbar för övrigt. Attribut-variabler används på ett likartat sätt men är tänkta för situationer där värdet ändras från vertex till vertex. En varying-variabel deklareras i vertex- och fragmentprogrammet. Den är skriv- och läsbar i vertexprogrammet men enbart läsbar i fragmentprogrammet. Värdet i fragmentprogrammet interpoleras fram. Det finns fördefinerade variabler som följer de tillstånd (attribut) vi sätter med t ex anropet *glVertex3d*. Några av dessa nämns i kolumnerna **In** i bilden. I kolumnerna **Ut** anges några variabler som måste (bör) ges värden. De används i stegen efter vertex- respektive fragmentprogrammet.

Vertexprogrammering 1(9)

Normalt transformeras all geometrisk information för ett hörn automatiskt från modellkoordinatsystemet till vykoordinatsystemet och projektionskoordinater och ljusvärde beräknas (om så begärts) för hörnet. Detta sker i princip när `glVertex` anropas, dvs övriga intressanta tillstånd (som färg och normaler) skall vara satta dessförinnan.

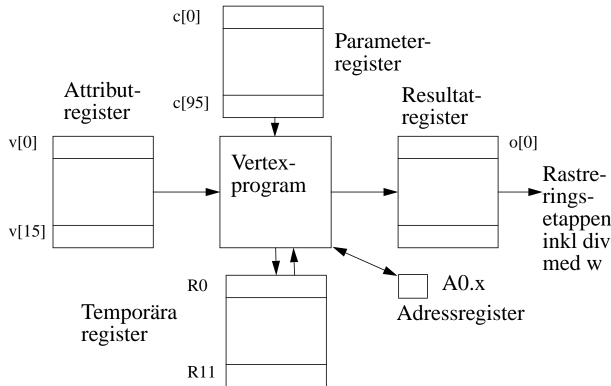


I de enklaste fallen handlar det om att beräkna positionen i projektionskoordinater (exkl divisionen med fjärde komponenten som görs i ett senare steg).

Denna etapp i den grafiska rörledningen är från och med NVIDIAs GeForce3 programmerbar. Men då måste man göra allt själv! Vitsen är att nya effekter kan uppnås och att arbete kan flyttas till grafikprocessorn.

Vertexprogrammering 2(9)

Kanske kan det ha något intresse att se på arkitekturen.



Vertexprogrammet arbetar mot ett antal maskinregister (dvs snabba minnesplatser). Vart och ett av dessa register innehåller fyra flyttal betecknade xyzw (punkter, vektorer) eller strq (texturkoordinater). Adressregistret dock bara ett tal. Antalet register liksom tillåten längd på vertexprogrammet varierar. I **attributregistren** hamnar hörnets attribut - storheter som ofta ändras per hörn. T ex position, färg och texturkoordinater. I **parameterregistren** storheter som ändras mindre ofta. T ex material- och ljusdata liksom diverse transformationsmatriser. Temporärregistren används naturligtvis för mellanräkningar och är unika på så sätt att de är både skriv- och läsbara. Slutligen placerar vertexprogrammet resultaten i **resultatregistren**. Tidigare refererade man till de olika registren med nummer, men GLSL låter oss referera till dem med namn som `vertex.position` och `resultat.color`.

Vertexprogrammering 3(9)

Det enklast tänkbara vertexprogrammet ser ut så här:

```
void main(void) {
    // Transformation till projektionskoordinater
    // Vi multiplicerar MVP-matrisen med positionen
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
    gl_FrontColor = gl_Color;
}
```

Det finns en inbyggd funktion *ftransform*, som garanterar att transformationen utförs exakt som om vi inte skrev ett eget vertexprogram. Med den blir det i stället

```
void main(void) {
    // Den normala (fixa) transformation
    gl_Position = ftransform();
    gl_FrontColor = gl_Color;
}
```

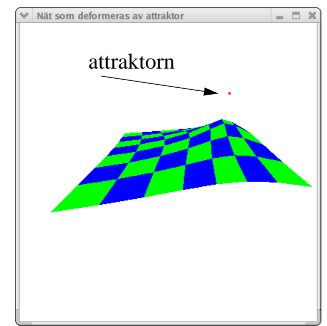
Om man vill krångla till det för sig kan man även skriva

```
void main(void) {
    // OBS! Eftersom m[i] avser en kolumn måste vi
    // använda den transponerade matrisen
    mat4 mvp = gl_ModelViewProjectionMatrixTranspose;
    //En rad i taget. dot = skalärprodukt.
    gl_Position.x  = dot(mvp[0],gl_Vertex);
    gl_Position.y  = dot(mvp[1],gl_Vertex);
    gl_Position.z  = dot(mvp[2],gl_Vertex);
    gl_Position.w  = dot(mvp[3],gl_Vertex);
    gl_FrontColor = gl_Color;
}
```

Vi kan alternativt skriva `gl_FrontColor = vec4(1.0, 0.0, 0.0, 0.0);` så blir allt ritat rött och vi ser hur man kan skriva en konstantvektor. Vill vi att färgläggningen skall göras utifrån modellkoordinaterna, kan det ske med (värden under 0 ger svart, över 1 ger vitt): `gl_FrontColor = sin(gl_Vertex);` Vi får alltså en färgsättning som beror på modellkoordinaterna utan att använda något *glColor*.

Vertexprogrammering 4(9)

Ett exempel hämtat från Mesa 4.1 modifierat för GLSL. Ett rutnät deformeras av en roterande attraktorn ovanför rutnätet. Attraktorns dragningskraft avtar omvänt proportionellt mot avståndet mellan dess position och aktuellt hörn. Vi flyttar en del beräkningar som skulle ha kunnat utföras av värddatorn till grafikprocessorn.



Vertexprogrammet (filen `$DG/DEMOS/Warp.vert;` kan köras med *VertexWarp2005* i samma mapp).

```
// Dessa får alltså värden i vår tillämpning
uniform vec4 Strength; // Attraktorns styrka
uniform vec4 Pos;      //  Dess position
void main(void) {
     vec4 r1, nytt_vertex; float r2;
     // Avståndet från attraktor till hörn beräknas
     // Vektor från hörn till attraktor
     r1 = Pos - gl_Vertex;
     // Avståndet i kvadrat
     r2 = dot(r1,r1);
     // Verkan skall vara omvänt proportionell mot avståndet
     r2 = inversesqrt(r2);
     r2 = r2*Strength.x;
     nytt_vertex = gl_Vertex + r2*r1;
     gl_Position = gl_ModelViewProjectionMatrix*nytt_vertex;
     gl_FrontColor = gl_Color;
}
```

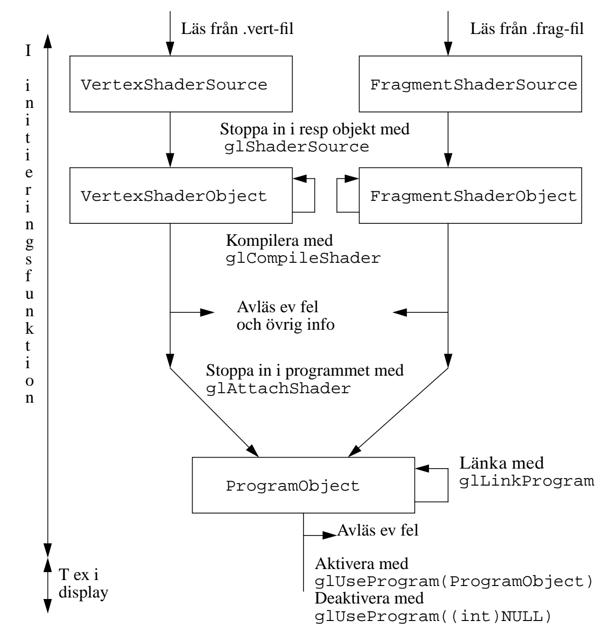

Vertexprogrammering 5(9)

Följande bild visar översiktligt vad som görs i OpenGL-programmet. Vi arbetar med tre globala s k objekt

`GLuint ProgramObject, VertexShaderObject, FragmentShaderObject`

samt två lokala (i *main*) strängar

`GLchar *VertexShaderSource, *FragmentShaderSource.`



Vertexprogrammering 6-9(9)

Över till OpenGL-programmet mest för fullständighetens skull (en del konstigheter förklaras av att man annars får kompileringsfel). Se `$DG/DEMOS/VertexWarp2005.c` för alla detaljer. PC-koden skiljer sig; se sen ...

```
#define GL_GLEXT_PROTOTYPES
#include <GL/glut.h>
GLuint ProgramObject;
GLuint VertexShaderObject;
GLuint FragmentShaderObject;

GLboolean glversion2() {...}
int shaderSize(char *fileName) {...}
int readShader(char *fileName, char *shaderText, int size)
{...}
int  readShaderSource(char  *fileName,  GLchar  **ourShader)
{..}
GLboolean initGLSL(const GLchar *vertexShader,
                   const GLchar *fragmentShader ) {
    GLchar    *pInfoLog;
    GLint compiled = GL_FALSE; //GLboolean ger typfel senare
    GLint linked   = GL_FALSE;
    GLint length, maxLength;
    // Skapa programobjekten.
    ProgramObject      = glCreateProgram();
    VertexShaderObject =
            glCreateShader(GL_VERTEX_SHADER);
    FragmentShaderObject =
            glCreateShader(GL_FRAGMENT_SHADER);
    length = strlen(vertexShader);
    if (vertexShader) {
      // Lägg in vertexprogramsträngen i objektet
      glShaderSource(VertexShaderObject, 1,
                  &vertexShader, NULL); //&length);
      // Man kan frisläppa utrymmet för strängen
      free((char*)vertexShader);
      // Kompilera vertexprogrammet och skriv ut ev info
      glCompileShader(VertexShaderObject);
```


felutskrit

```
      glGetProgramiv(VertexShaderObject,
                GL_COMPILE_STATUS, &compiled);
      glGetProgramiv(VertexShaderObject,
               GL_INFO_LOG_LENGTH, &maxLength);
      pInfoLog=(GLchar *)malloc(
                 maxLength*sizeof(GLchar));
      glGetProgramInfoLog(VertexShaderObject, maxLength,
                 &length, pInfoLog);
      printf("%s", pInfoLog);
      free(pInfoLog);
      if (!compiled) {
         printf("Misslyckad kompilering av VertexShader\n");
         return GL_FALSE;
      }
      printf("Lyckad kompilering av VertexShader\n");
      // Stoppa in i programobjektet
      glAttachShader(ProgramObject,VertexShaderObject);
      // Det är bra att ta bort shaderobjektet nu        /
      glDeleteShader(VertexShaderObject);
    } // Slut på vertexshaderdelen
    // Motsv för fragmentshader
    // Länka och skriv ut ev info
    glLinkProgram(ProgramObject);
    glGetProgramiv(ProgramObject,
               GL_LINK_STATUS, &linked);
    glGetProgramiv(ProgramObject,
               GL_INFO_LOG_LENGTH, &maxLength);
    pInfoLog=(GLchar *)malloc(
               maxLength*sizeof(GLchar));
    glGetProgramInfoLog(ProgramObject, maxLength, NULL,
               pInfoLog);
    printf("%s\n", pInfoLog);
    free(pInfoLog);
    // Aktivera programobjektet
    if (linked) {
        glUseProgram(ProgramObject);
        return GL_TRUE;
    } else { return GL_FALSE; }
}
```



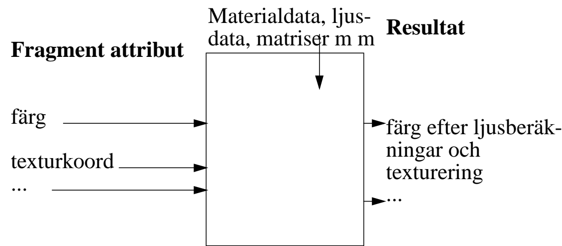
```
void DrawMesh( int rows, int cols ) {
   static const GLfloat colorA[3] = { 0, 1, 0 };
   static const GLfloat colorB[3] = { 0, 0, 1 };
   const float dx = 2.0 / (cols - 1);
   const float dy = 2.0 / (rows - 1);
   float x, y;
   int i, j;
   y = -1.0;
   for (i = 0; i < rows - 1; i++) {
      glBegin(GL_QUAD_STRIP);
         x = -1.0;
         for (j = 0; j < cols; j++) {
            if ((i + j) & 1)  glColor3fv(colorA);
            else glColor3fv(colorB);
            glVertex2f(x, y); glVertex2f(x, y + dy);
            x = x + dx;
         }
      glEnd();
      y = y + dy;
   }
}
GLfloat Phi = 0.0, X=1.5;
// Snurrar på attraktorn
void idle( void ) {
   Phi += 0.01;  glutPostRedisplay();
}
void reshape( int width, int height ) {
   glViewport( 0, 0, width, height );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective(90,1.0,0.1,25.0);
}
void display( void ) {
   GLfloat x, y, z, r = 0.5;
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   gluLookAt(0,0,X, 0,0,0, 0,1,0);
   glRotatef(-60.0, 1, 0, 0);
```



```
   // Position the gravity source
   x = r * cos(Phi); y = r * sin(Phi); z = 0.7;
   glUniform4f(glGetUniformLocation(
                  ProgramObject,"Pos"), x, y, z, 1);
   // Rita ljuskällan röd
   glUseProgram(NULL);//0 i st f NULL hindrar varning
   glBegin(GL_POINTS);
        glColor3f(1,0,0); glVertex3f(x, y, z);
   glEnd();
   glUseProgram(ProgramObject);
   DrawMesh(8, 8);
   printf("glGetError=%s\n",
        gluErrorString((int)glGetError()));
   glutSwapBuffers();
}

int main(int argc, char **argv) {
    GLchar *VertexShaderSource;
    glutInitDisplayMode(GLUT_RGBA|GLUT_DEPTH|GLUT_DOUBLE );
    glutInitWindowSize(400, 400);
    glutCreateWindow("Nät som deformeras av attraktor");
    if (!glversion2()) {
       printf("GLSL stöds inte av denna dator.\n");
       exit(1);
    }
    glutDisplayFunc(display);
    glutIdleFunc(idle);
    glutReshapeFunc(reshape);
    readShaderSource("Warp.vert",&VertexShaderSource);
    initGLSL(VertexShaderSource, NULL);
    glUniform4f(glGetUniformLocation(
               ProgramObject,"Strength"), 0.5,0,0,0);
    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0, 1.0, 1.0, 1);
    glShadeModel(GL_FLAT);
    glPointSize(3);
    glutMainLoop();
    return 0;
}
```

Fragmentprogrammering 1(4)

I den vanliga rörledningen behandlas fragmentet enligt figuren nedan. Ljusberäkningarna är redan gjorda på vertexnivå (om vi inte använt ett vertexprogram). Väsentligen handlar det därför om texturering i ett eller flera steg (och dimma som vi inte tagit upp).



Denna etapp i den grafiska rörledningen är från och med NVIDIAs GeForce3 programmerbar. Det viktigaste användningsområdet är mer realistiska ljusberäkningaar, t ex Phong-toning, som ju måste göras per bildpunkt. Det praktiska maskineriet är mycket likt det vid vertexprogrammering. I själva OpenGL-programmet läser man in och kompilerar m m fragmentprogrammet på samma sätt som när det gällde vertexprogrammet. Det är bara när man med *glCreateShader* skapar ett fragmentshader-objekt som parametern skall vara annorlunda.

Fragmentprogrammering 2(4)

När det gäller fragmentprogrammets variabler (se tidigare tabell) avser nu

- `gl_Color` det färgvärde som givits `gl_FrontColor` resp `gl_BackColor` i vertexprogrammet om sådant används, annars det värde som beräknats av standardmaskineriet. Fragmentet kommer i förekommande fall både som ett framsides- och ett baksidesfragment.
- `gl_TexCoord[i]` det värde som tilldelats `gl_TexCoord[i]` i vertexprogrammet eller om sådant inte ingår det värde som hör ihop med den i:te texturenheten. Värdet är med interpolation framräknat från hörnvärdena. Detta gäller även t ex `gl_Color`.
- `gl_FrontMaterial` och `gl_BackMaterial` med komponenterna `.emission`, `.ambient`, `.diffuse`, `.specular` och `.shininess` materialvärden som satts med *glMaterial* i OpenGL-programmet. Den sista komponenten är `float` medan övriga är `vec4`.
- `gl_LightSource[i]` med bl a komponenterna `.ambient`, `.diffuse`, `.specular`, `.position` och `.halfVector` ljusvärden som satts med *glLight* (den sista är automatiskt framräknad). De nämnda komponenterna är alla av typen `vec4`.
- `gl_FragColor` utgående färg. Denna variabel kan även läsas och kan ges värde flera gånger, vilket ibland är bekvämt.

Fragmentprogrammering 3(4)

Två enkla fragmentprogram ser ut så här:

```
void main(void) {
    gl_FragColor = gl_Color;
    //gl_FragColor = vec4(1.0, 1.0, 0.0, 0.0);
}
```

Om vi flyttar kommentartecknen en rad uppåt blir fragmentet gult oberoende av vad som hänt tidigare.
Ett något intressantare fragmentprogram är

```
uniform sampler2D enhetsnr;
void main(void) {
    gl_FragColor = texture2D(enhetsnr, gl_TexCoord[3].st);
}
```

som sätter utgångsfärgen till det värde i texturkartan hörande till texturenheten *enhetsnr* som pekas ut av texturkoordinaterna angivna med `glMultiTexCoord2f(GL_TEXTURE3,...)`. Variabeln *enhetsnr* måste ha tilldelats ett värde med

```
glUniformi(glGetUniformLocation(
ProgramObject,"enhetsnr"),heltal);
```

i OpenGL-programmet. I det måste också texturen bindas till en plats i texturregistret (och blir då ett texturobjekt), men `glEnable(GL_TEXTURE_2D)` och *glTexEnvf*-anropen behövs nu inte.

I ett glUniform-anrop tar vi först med `glGetUniformLocation(ProgramObject,"variabelnamn")` reda på platsen (registernummer, heltal 0 och uppåt) för variabeln och ger den sedan ett värde. Detta kan verka primitivt, men tanken är att man effektivt skall kunna ta reda på platsen en gång för alla.

Utan fragmentprogram utförs ju ett antal operationer per fragment. Med utökar vi bara antalet operationer, vilket gör att korta fragmentprogram inte kostar märkbart i tid.

Fragmentprogrammering 4(4)

Nu en kombination av vertex- och fragmentprogram som gör Phongtoning (enbart av det diffusa ljuset här och vi låtsas som om ljuskälla och betraktare är långt bort, vilket gör att vektorn L mot ljuskällan är konstant över objektet). Vi antar att ljuset är maximalt vitt, dvs {1.0, 1.0, 1.0, 1.0}. Först `LIGHT.vert`:

```
void main(void) {
    // Transformation från modellkoord till projkoord
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // Transformera normalen till vykoordinater, se "Från .."
    // Lägg som texturkoordinat 0 för interpolatio
    // gl_Normal är vec3 och gl_NormalMatrix mat3
    gl_TexCoord[0].stp = gl_NormalMatrix * gl_Normal;
    // Borde vara likvärdigt men namnet finns ej hos oss
    //gl_TexCoord[0] = gl_ModelViewMatrixInverseTranspose
                       * (vec4(gl_Normal,0.0));
    // Nu kommer vi åt den interpolerade normalen
    // i fragmentprogrammet
}
```

Över till fragmentprogrammet `LIGHT.frag`:

```
uniform vec3 LightPos;
void main(void) {
   vec4 diffuse = gl_FrontMaterial.diffuse;
   vec4 specular = gl_FrontMaterial.specular;
   vec3 L, N; float dotProd;
   // Vi har en ljusriktning, denna har transformerats och
   // avser vykoordinater. Normalisera den
   // Tycks inte uppdateras
   // L = normalize(gl_LightSource[0].position.xyz);
   L = normalize(LightPos);
   // Normalisera den interpolerade normalen
   N = normalize(gl_TexCoord[0].stp);
   // Beräkna skalärprodukten LN
   dotProd = dot(L,N); gl_FragColor = diffuse * dotProd;
}
```

Fragmentprogrammering: Felhantering

Se OH-242 för hur felutskrifter från t ex kompilering av ett vertex- eller fragmentprogram kodas.

De flesta fel upptäcks vid körningen av OpenGL-programmet. Några exempel (talet inom parentes anger radnummer):

Rad utan avslutande semikolon
```
(9) : error C0000: syntax error, unexpected identifier, expecting ';'
```
Deklaration utan typangivelse
```
(9) : error C0501: type name expected at token "r1"
```
Odeklarerad variabel ges värde
```
(9) : error C1054: initialization of non-variable "r1"
```
Odeklarerad (felstavad) variabel
```
(13) : error C1008: undefined variable "gl_vertex"
```
Typkonflikt vid addition
```
(13) : error C1022: operands to "add" must be numeric
```

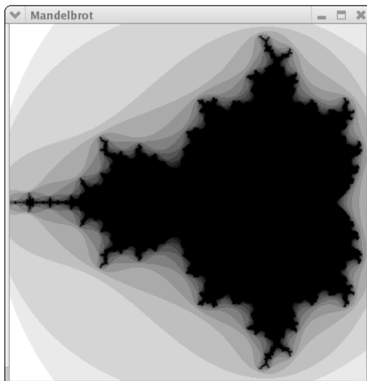
Sedan kan naturligtvis OpenGL-programmet gå fel med den tråkiga utskriften
`Segmentation fault`
Då bör man tillkalla handledare om man inte är bevandrad i avlusaren *gdb*. Denna avlusare fungerar ev dåligt i Linux-miljö mot program länkade med de dynamiska grafikbiblioteken, vilket `OGL_COMPILE` gör. Man kan i stället använda `OGL_COMPILE_STAT`, som länkar mot ett statiskt GLUT-bibliotek.

Fragmentprogrammering: Mandelbrot i GPU 1(3)

I OpenGL-programmet (`$DG/DEMOS/MANDELFRAG2005.c`; ej omskrivet för OpenGL 2.0) ritar vi en kvadrat motsvarande den del av komplexa talplanet vi är intresserade av.

Vertexprogrammet (`$DG/DEMOS/Mand.vert`)
```
varying vec2 pos;// för kommunikation med fragmentprogrammet
void main(void) {
     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
     pos = vec2(gl_Vertex.x, gl_Vertex.y);
}
```

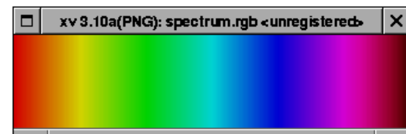
Fragmentprogrammet (`$DG/DEMOS/Mand.frag`) på nästa sida.



Fragmentprogrammering: Mandelbrot i GPU 2(3)

```
varying vec2 pos;
uniform sampler2D enhetsnr;
void main(void) {
     vec2 c = pos; // pos is read-only
     float size; int n = 0;
     vec2 z = vec2(0.0, 0.0);
     vec2 znew;
     float del;
     float iter;
     float max = 12.0;
     for (iter = 0.0; iter < max; iter++) {
          znew.x = z.x*z.x -z.y*z.y + c.x;
          znew.y = 2*z.x*z.y+c.y;
          z = znew;
          size = dot(z,z);
          if (size < 4.0) n = n + 1;
     }
     del = 1-n/max;
     // gl_FragColor = texture2D(enhetsnr, del,0.5);
     gl_FragColor = vec4(del,del,del,0);
     // gl_FragColor=noise1(z.x);
     // ger alltid 0 enligt info april 2005
     // gl_FragColor=vec4(sin(z.x),sin(z.y),0,0); snygg!!!
}
```

Den först bortkommenterade raden ger färg om vi använder texturen



Fragmentprogrammering: Mandelbrot i GPU 3(3)

Texturen är egentligen 1-dimensionell men läst som 2-dimensionell, varför jag använt `sampler2D` och `texture2D` i programmet. texture2D(...) använder den textur som f n är bunden till den texturenhet som anges av enhetsnr. Texturkoordinaterna anges som de två sista parametrarna.

Inläsning etc av textur kan vi i detta fall göra i *main* i OpenGL-programmet

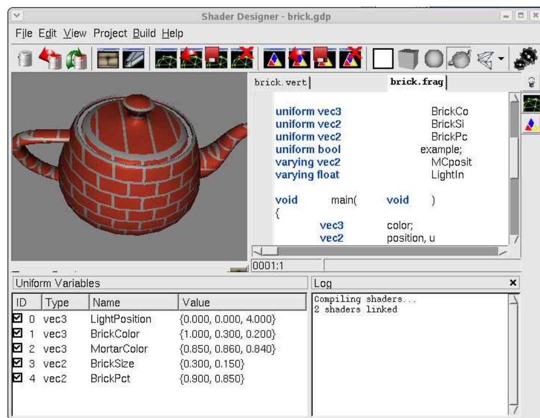
```
// texture global vektor
glGenTextures(1,texture);
// Vi lägger texturen i texturenhet 0
glActiveTexture(GL_TEXTURE0);
// Läsning av texturen och glBindTexture och glTexEnvf
//samt glTexParameteri
...
// Meddela fragmentprogrammet texturenhetens nummer
glUniform1i(glGetUniformLocation(
                         ProgramObject,"enhetsnr"),0);
```

Vertex- och fragmentprogrammering: Övrigt

Den kraft som grafikprocessorerna erbjuder kanske kan användas för allmänna beräkningar? Det finns grupper som sysslar med just detta, *General-Purpose computation on GPUs* (förkortat GPGPU). Till och med flera webbsidor, bl a `www.gpgpu.org`.
Det finns en OpenGL-emulator Mesa, som går att köra på en godtycklig dator oberoende av grafikkort. Den motsvarar OpenGL 1.5 och stöder därmed f n inte GLSL och inte heller de OpenGL-funktioner som används då. Däremot klarar den lågnivå-varianten av vertex/fragmentprogrammering. Arbete sägs pågå för att höja Mesa till OpenGL 2.0-nivån.

Utvecklingsmiljöer för vertex- och fragmentprogram 1(3)

Med detta begrepp avses program där man kan experimentera med vertex- och fragmentprogram och se hur de påverkar olika typer av objekt. Sådana finns för Microsofts HLSL och GLSL (och föregångare).
Vi intresserar oss här bara för GLSL. Jag har hittat en - kallad *ShaderDesigner* (`http://www.typhoonlabs.com`) - som finns för både Linux (dock kommersiell numera) och Windows. Min Linux-version av den är dålig (textvisning och -redigering är opålitlig), men belyser ideerna. När det gäller Windows-versionen tycks mina rättigheter inte räcka till för en korrekt installation under StuDAT. Så här ser det i alla fall ut under Linux:

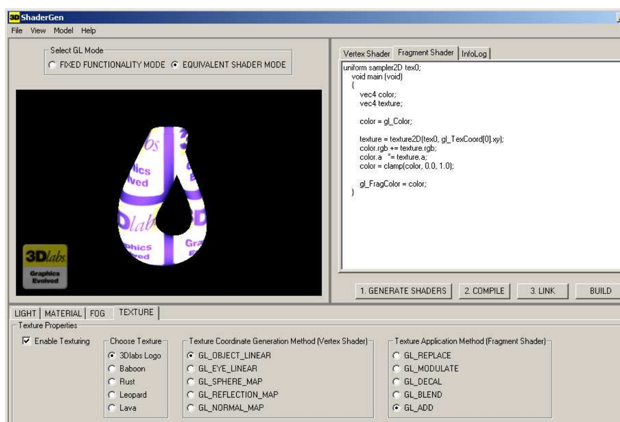


Utvecklingsmiljöer för vertex- och fragmentprogram 2(3)

I figurdelen kan man rotera och förflytta objektet liksom zooma med musknapparnas hjälp. Med verktygen till höger upptill kan vi växla mellan några olika objekt. I figuren är tekannan vald. Därunder kan vi inspektera antingen vertex- eller fragmentprogrammet och även i princip ändra i dem. Under figurdelen hittar vi en variabellista. Genom att klicka på en rad kan man med viss svårighet ändra värden.

Starta programmet med *shaderdesigner*. Med **File**/**Open Shader Project** öppnar du `$DG/SHADERDESIGNER/shaders/brick.gdb`.

För Windows har vi i stället ett liknande program kallat *ShaderGen* från 3DLabs (`http://www.3dlabs.com`).



Utvecklingsmiljöer för vertex- och fragmentprogram 3(3)

ShaderGen är tänkt att visa hur ett standardbeteende kan kodas med vertex- och fragmentprogram. Välj inställningar som i figuren och tryck på knappen **Build**. Du får då upp en textur applicerad på något objekt. Objekt kan bytas med **Model**-menyn (jag har valt Klein). Fragmentprogrammet kan studeras liksom vertexprogrammet. Man kan också ändra i dessa, vilket får genomslag efter **Compile** + **Link**. Ändringar av inställningarna i nedre halvan får ofta omedelbar effekt.

Gå till mappen `$DG/PC/SHADERGEN/bin` och starta `ShaderGen.exe`. Experimentera litet. Ändra t ex till gl_FragColor = vec4(1,0,0,0).

En Mac-ägabde kursdeltagare nämnde *OpenGL Shader Builder* för MacOS X. Ngot oklart om den klarar GLSL.

Det finns andra "shading"-språk. Det mest kända är **Renderman** (`http://www.renderman.org`), som har använts för filmframställning och är oberoende av grafikprocessor. Något fritt program för det språket känner jag just nu inte till.

Sh (http://libsh.org) "is a library that acts as a language embedded in C++, allowing you to program GPUs (Graphics Processing Units) and CPUs for graphical and general-purpose computations in novel ways."

Uppgift 7, laboration 3

a) **Linux-användare**: Starta programmet med *shaderdesigner*. Med **File**/**Open Shader Project** öppnar du `$DG/SHADERDESIGNER/shaders/brick.gdb`. Experimentera men försök inte ändra i programtexten.
Windows-användare: Gå till mappen `$DG/PC/SHADERGEN/bin` och starta `ShaderGen.exe`. Experimentera litet. Ändra t ex till `gl_FragColor = vec4(1,0,0,0)`.
b) Kopiera filerna `LIGHT.vert` och `LIGHT.frag` i `$DG/DEMOS` till din egen katalog. Kör programmet `$DG/DEMOS/LABVERTEXFRAG`. Det arbetar i Phong-moden med vertex- och fragmentprogrammet. Använd menyerna knutna till MK3. `LIGHT.frag` gör fragmentvis ljusberäkning men tar bara hänsyn till den diffusa reflektionen. Ändra i `LIGHT.frag` så att hänsyn till spegelreflektionen tas. Låt vektorn V mot betraktaren vara (0,0,1). Använd som exponent n talet 20.0. Beräkna det spekulära ljuset som $(V \bullet R)^n$, där $R=2(L \bullet N)N-L$ normerad (vår modells sätt). R kan beräknas med `reflect(-L,N)`. Alla vektorer skall vara normerade och alla beräkningar görs i vykoordinatsystemet. Använd `pow(float,n)` för upphöjt till.
OBS! Du skall bara utöka fragmentprogrammet, inte ändra i OpenGL-programmet. Uppgiften är därmed oberoende av om du hittills arbetat i C, C++ eller Java. **Redovisa b-delen av uppgiften**.
Anm. Källkoden `LABVERTEXPROG.c` finns i samma mapp, utan att jag gjort någon slutlig tillsnyggning av koden. Den har inte heller anpassats till OpenGL 2.0. Jag hade förra året märkliga problem med de knyckta läsfunktionerna och tvingats kommentera bort ett *fclose* och ett *free*. Därefter har allt gått bra.
Uppgift 8, lab 3 utgår men det hindrar ju inte att du försöker animera.

Utvidgningar av OpenGL 1(4)

OpenGL är standardiserat. Men samtidigt utvecklas grafikkorten i mycket rask takt och blir allt kraftfullare. Därför inför olika tillverkare utökad funktionalitet. Flera sådana äldre utvidgningar har tagit sig in i standarden (ett tidigt exempel är *glBindTexture*, som gör det lätt att arbeta med mer än en texturkarta). Man kan lätt undersöka vad som finns tillgängligt av hårdvara och utvidgningar. Se avsnitt 14 i OpenGL-häftet. Partiella utskrifter från programmet `GL_INVESTIGATE2005.c` på ett par OH fram redovisas nedan. Man skiljer på utvidgningar som ARB (Architecture Review Board, som har hand om OpenGL) accepterat och sådana som bara några leverantörer kommit överens om (EXT) eller bara en infört (t ex NV från NVIDIA eller SGI från Silicon Graphics).

Datorerna i 6220 (PC med Linux, grafikkort NVIDIA FX6600)
Version: 2.0.0 NVIDIA 76.64
Leverantör: NVIDIA Corporation
Grafiksyst: GeForce 6600/PCI/SSE2/3DNOW!
GLSL-VERSION: 1.10 NVIDIA via Cg 1.3 compiler
Utvidgningar: ... **GL_ARB_fragment_program** GL_ARB_fragment_program_shadow **GL_ARB_fragment_shader** GL_ARB_occlusion_query GL_ARB_point_parameters GL_ARB_point_sprite **GL_ARB_shader_objects GL_ARB_shading_language_100** **GL_ARB_vertex_shader** GL_EXT_Cg_shader GL_EXT_point_parameters GL_EXT_secondary_color GL_EXT_separate_specular_color GL_EXT_texture3D GL_EXT_texture_compression_s3tc GL_EXT_texture_cube_map GL_HP_occlusion_test GL_NV_fragment_program GL_NV_occlusion_query GL_NV_point_sprite GL_NV_texture_compression_vtc ...
Antal aux-buffertar: 4
Antal bitar per pixel i djupbuffert: 24
Antal röda bitar per pixel: 8
Antal alfabitar per pixel: 0
Antal stencil bitar per pixel: 8
Antal röda ackumuleringsbitar per pixel: 16
Stödjer stereo?0
Stödjer dubbel-buffring?1
GLUT_WINDOW_BUFFER_SIZE= 32

Vissa av nollorna beror på att vi inte i programmet begärde motsvarande resurs.

Utvidgningar av OpenGL 2(4)

Ett grävande program (GL_INVESTIGATE2005.c)

```
#include <GL/glut.h>
int main (int  argc,char *argv[]) {
    GLint a[5];  GLboolean b[5];  const char *v;
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_DEPTH |
                 GLUT_RGBA| GLUT_STENCIL );
    // Måste vara med
    glutCreateWindow ("Undersökning: Se terminalfönstret");
    v = glGetString(GL_VERSION);printf("Version: %s\n", v);
    v=glGetString(GL_VENDOR);printf("Leverantör: %s\n", v);
    v=glGetString(GL_RENDERER);printf("Grafiksystem: %s\n",v);
    v=glGetString(GL_SHADING_LANGUAGE_VERSION);
         printf("GLSL-VERSION: %s\n",v);
→   v=glGetString(GL_EXTENSIONS);printf("Utvidgningar: %s\n", v);
    glGetIntegerv(GL_AUX_BUFFERS,a);
    printf("Antal aux-buffertar: %d\n", a[0]);
    glGetIntegerv(GL_DEPTH_BITS,a);
    printf("Antal bitar per pixel i djupbuffert: %d\n",a[0]);
    glGetIntegerv(GL_RED_BITS,a);
    printf("Antal röda bitar per pixel:  %d\n", a[0]);
    glGetIntegerv(GL_ALPHA_BITS,a);
    printf("Antal alfabitar per pixel:  %d\n", a[0]);
    glGetIntegerv(GL_STENCIL_BITS,a);
    printf("Antal stencil bitar per pixel:  %d\n", a[0]);
    glGetIntegerv(GL_ACCUM_RED_BITS,a);
    printf("Antal röda ackumuleringsbitar per pixel:  %d\n",
           a[0]);
    glGetBooleanv(GL_STEREO,b);
    printf("Stödjer stereo?%d\n", b[0]);
    glGetBooleanv(GL_DOUBLEBUFFER,b);
    printf("Stödjer dubbel-buffring?%d\n", b[0]);
    if (glutExtensionSupported("GL_EXT_polygon_offset"))
         printf("Visst stödjer jag det du frågade om!\n");
    //GLUT kan också ge besked, t ex antal bitar i bildminne
    printf(" GLUT_WINDOW_BUFFER_SIZE= %d\n",
        glutGet(GLUT_WINDOW_BUFFER_SIZE));
}
```

Programmet finns i `$DG/DEMOS`.

Utvidgningar av OpenGL 3(4)

En hel del utvidgningar har tagit sig in i senaste standard (specifikation) OpenGL 2.0 (september 2004; 1.5 kom oktober 2003). Det finns några problem.

- Dokumentationen finns inte alltid fullt ut i specifikationen (hittas via `www.opengl.org`) , utan i lösa dokument (`http://oss.sgi.com/projects/ogl-sample/registry/`).
- Standarden avviker i vissa avseenden från tidigare NV- eller ATI-versioner.

Här följer en lista över **några** utvidgningar som tagits in i OpenGL:s kärna (det gör att ARB i namnen i princip kan uteslutas):

- 2.0 GLSL (OpenGL Shading Language) och shading objekt, texturer utan 2^n-kravet, ...
- 1.5 Occlusion query, fragmentprogram (enbart utvidgning), ...
- 1.4 Vertex-program (enbart utvidgning), shadow, ...
- 1.3 Multitexturering, texturkomprimering, kubtexturer, alla matriser kan fås på transponerad form vilket är naturligare och förenklar, ...
- 1.2 3D-texturer, bildbehandlingsdel, ...
- 1.1 Hörnvektorer, texturobjekt, polygonoffset, ...

Utvidgningar av OpenGL 4(4)

De senaste av ARB accepterade utvidgningarna är
26-27 (Juni 2002/Sept 2002):*GL_ARB_vertex_program*, *GL_ARB_fragment_program* (**Anm**. Microsoft claims to own intellectual property related to this extension), som har sina ursprung i ATI- och NV-utvidgningar.
28. *GL_ARB_vertex_buffer_object*
29. *GL_ARB_occlusion_query*
30-32 (Juni 2003): *GL_ARB_shader_objects*,*GL_ARB_vertex_shader*, *GL_ARB_fragment_shader*
33 (Juni 2003): *GL_ARB_shading_language_100*
34 (Juni 2003): *GL_ARB_texture_non_power_of_two*
35. *GL_ARB_point_sprite*
36. *GL_ARB_fragment_program_shadow*
37 (Juli 2004): *GL_ARB_draw_buffers* (standard i 2.0)
38 (Juni 2004): *GL_ARB_texture_rectangle*
39. *GL_ARB_color_buffer_float*
40. *GL_ARB_half_float_pixel*
41. (Okt 2004) *GL_ARB_texture_float*
42. (Dec 2004) GL_ARB_pixel_buffer_object

Flera av dessa är inaktuella i och med att motsvarigheter tagit sig in i OpenGL 2.0.

Utvidgningar i egen kod på PC 1(2)

Vill man på en PC (med någon medlem i familjen Windows) dra nytta av utvidgningar i OpenGL (inkl OpenGL 2.0) måste man i minst ett avseende förfara litet annorlunda (än i andra system) vid egen kodning.

- Självklart måste du ha grafikkort (och tillhörande programvara) som stöder utvidgningen. Det bör observeras att Microsofts ursprungliga dynamiska bibliotek `opengl32.dll` inte förändras, vilket nog hade varit naturligt. I stället tillhandahåller korttillverkaren en ICD (installable client driver), som anropas av `opengl32.dll`.
- Du måste också ha (finns t ex via OpenGLs webbsida `www.opengl.org`) `glext.h` och placera den tillsammans med `gl.h`. Under Linux inkluderas den från `gl.h`, men vanligen inte under Windows. Skriv därför `#include <GL/glext.h>` efter motsvarande glut-rad. Härigenom får man tillgång till de flesta nya namn. Rätt filer följer rimligen med vid dator/kort-köp, men de kan behöva placeras lämpligt.
- I koden bör du kontrollera att den aktuella utvidgningen är tillgänglig, t ex enligt modellen

  ```
  if (glutExtensionSupported("GL_ARB_multitexture")) { ... }
  else ...
  ```

  Detta kan göras tidigast efter `glutCreateWindow ("...")`.
- Om du vill använda ett nytt funktionsnamn (som ej hör till kärnan) måste du förfara enligt modellen

  ```
  glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
                wglGetProcAddress("glMultiTexCoord2fARB");
  ```

  Samma placering som i förra punkten. Detta krångliga sätt, som inte alls behövs under Linux eller Solaris, hänger ihop med Microsofts åsikt att `opengl32.dll` inte får ändras.

Utvidgningar i egen kod på PC 2(2)

Sista punkten på föregående sida vållar mycket besvär och gör att programmen inte utan vidare är transportabla. På nätet kan man hitta färdiga initieringsprocedurer, som underlättar. Just nu är nog GLEW - OpenGL Extension Wrangler[1] (se `http://glew.sourceforge.net`) mest inne).

Man skriver då i st f

```
#include <GL/glut.h>
```

raderna

```
#define GLEW_STATIC 1
#include <GL/glew.h>
#include <GL/glu.h>
#include <GL/glut.h
```

och före glutMainLoop();

```
glewInit();
```

Kompilering under Linux:

```
$gcc -o Prog.out Prog.c  -I/users/course/TDA360/LINUX/GLEW/
glew/include  -L/users/course/TDA360/LINUX/GLEW/glew/lib  -
lGLEW -lglut -lGLU -lGL
```

och under Windows (hos oss; tillfälliga placeringar; och ger just nu fel)

```
gcc -o Prog.exe Prog.c -I../PC/GLEWPC/src/glew/include -L../
PC/GLEWPC/src/glew/lib -lglew32 -lglu32 -lopengl32
```

Före körning under Linux

```
setenv LD_LIBRARY_PATH /users/course/TDA360/LINUX/GLEW/glew/
lib:$LD_LIBRARY_PATH
```

Liknande för Windows

1.Boskapsskötare enl ordboken

Mesa och OpenGL 1.5 1(2)

Mesa (`www.mesa3d.org`) är en OpenGL-emulator, som i sin senaste version 6.0 (dec 2003) (6.2 okt 2004) uppges klara allt i OpenGL-specifikationen 1.5 (även GLSL tycks det). Utan att man har program/maskinvara för OpenGL! Mesa bygger bara vidare på datorns bottengrafiksystem, dvs t ex X eller Windows. En ansenlig prestation av främst upphovsmannen Brian Paul. Men snabbt går det inte. Motiven för vidareutveckling verkar något svaga, sedan SGI släppt källkod för OpenGL. Det förefaller dock som om Mesa i viss mån utnyttjar eventuellt installerad OpenGL. Inga specialåtgärder behövs under Windows. Man inkluderar bara som vanligt `GL/glut.h`.
För den som inte har tillgång till bra grafikkort kan det användas för att pröva t ex vertex- och fragmentprogrammering (uppgift 7 på laboration 3). En annan användning kan vara felsökning. Men långsamt.

Hur på kurskontot? (ej kontrollerat 2005)

- Kompilering och länkning med
  `MESALINK DittOpenGLProg`
- Körning med `DittOpenGLProg`
  eller `MESARUN DittOpenGLProg`
- I `$DG/MESA6.0/Mesa-6.0/progs` finns ett antal demoprogram (källkod och exekverbara program).

På PC?

Binärer för Visual C++ finns att hämta. Se kurssidan. För gcc har jag inte lyckats fullt ut, men något finns via kurssidan. OBS! Mesa har egna DLL-er för OpenGL.

Beräkningsgeometri: Allmänt

Vi skall här se på några geometriska problem som är av datorgrafiskt intresse. **Beräkningsgeometri** (eng computational geometry) anses handla om algoritmer (och datastrukturer) för geometriska problem. Det är alltså frågan om konstruktion och analys av metoder för konkret praktisk lösning av problem och inte den andra sidan av geometrin nämligen härledning av egenskaper. Vi presenterar inte fullständiga algoritmer och gör inte heller ordentliga prestandaanalyser. Och vi tar bara upp några få problem.

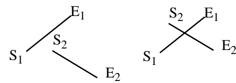
Vi förutsätter i allmänhet att ändpunkter etc har heltalskoordinater, dvs att vi arbetar i skärmkoordinatsystemet. Det gör att vi kan förutsätta att alla beräkningar görs exakt.

Vi har tidigare mött flera problem som skulle kunna sägas tillhöra området. Och som vi sa redan i början av kursen och alla vid det här laget märke till spelar matematik en betydande roll inom datorgrafik.

Referens: O'Rourke, J: Computational Geometry in C, Cambridge University Press, 1994.

Beräkningsgeometri: Linjers skärning i 2D

Vi har två linjesegment och det gäller att avgöra om de skär varandra och i så fall bestämma skärningspunktens koordinater. Ett segment



med startpunkt i S = (Sx,Sy) och slutpunkt i E = (Ex,Ey), kan matematiskt beskrivas dels på parameterform

P = S+t(E-S), $0 \le t \le 1$,

dels på tradionell form

F(x,y) ≡ Ax + By + C = 0 med begränsning av t ex x,

där A=(Ey-Sy), B=(Ex-Sx), C=ExSy-SxEy.

Algoritm 1: Beteckna parametern för de båda linjerna med *t* respektive *u*. Vi får då

$S_1 + t(E_1 - S_1) = S_2 + u(E_2 - S_2)$

som är ett ekvationssystem med två ekvationer för de två obekanta *t* och *u*. Om linjerna inte är parallella kan vi alltså lösa ut *t* och *u*. Skärning karakteriseras av att $0 \le t,u \le 1$. Annars är det segmentens förlängningar som skär varandra.

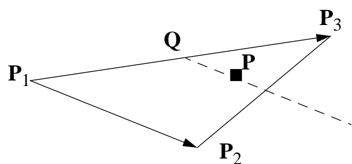
Om de flesta segment som vi vill undersöka skär varandra är detta en fullt acceptabel metod. Om däremot de flesta inte gör det, så lägger man ner onödigt mycket möda och följande metod är effektivare.

Algoritm 2: Vi undersöker först om ändpunkterna hos den andra linjen ligger på ena sidan om den första, dvs om $F_1(S_2)$ och $F_1(E_2)$ har samma tecken. I så fall finns ingen skärning. På samma sätt undersöks om $F_2(S_1)$ och $F_2(E_1)$ har samma tecken. I så fall finns heller ingen skärning. Annars skär segmenten varandra och skärningspunkten kan beräknas med linjär interpolation utifrån de beräknade värdena.

Beräkningsgeometri: Barycentriska koordinater

Ibland behöver man kunna beskriva punkter i en triangel. Vi såg ett sådant exempel i samband med morfing och möter ett annat på nästa OH.

Varje punkt **P** i en triangel



kan skrivas

$$P = P_1 + v(P_3 - P_1) + u(P_2 - P_1)$$

där $0 \le u$, $0 \le v$ och $u+v \le 1$, vilket framgår av figuren. Vi går ju från P_1 i vektorn P_3-P_1:s riktning till Q och därefter längs den streckade linjen, som har samma riktning som vektorn P_2-P_1. Kanten P_2 -P_3 motsvarar att u+v=1.

Alternativt utnyttjar vi att en triangel är konvex och att punkterna i den därför genereras med $P = \sum \alpha_j P_j$, med $\sum \alpha_j = 1$ och $0 \le \alpha_j \le 1$ eller omskrivet $P = (1 - \alpha_2 - \alpha_3) \cdot P_1 + \alpha_2 P_2 + \alpha_3 P_3$, som överensstämmer med den tidigare med $u = \alpha_2$ och $v = \alpha_3$. Man brukar kalla $(\alpha_1, \alpha_2, \alpha_3)$, där $\alpha_1 = 1 - \alpha_2 - \alpha_3$, för **barycentriska koordinater** (Möbius 1827 eller tidigare). Vi kanske kan kalla (u,v) för **triangelkoordinater**.

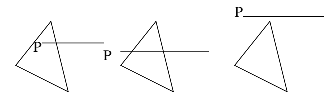
Beräkningsgeometri: Punkt i polygon 1(3)

Bl a vid pekning på objekt kan det vara av intresse att veta om en punkt ligger inuti en polygon eller inte. En del grafikbibliotek har en inbyggd funktion (typiskt namn *PointInPolygon*) för detta i skärmkoordinater. Specialfallet att polygonen är en rektangel med axelparallella sidor är simpelt. I det allmänna fallet kan det vara lämpligt att hålla reda på en omslutande rektangel, mot vilken grovtest sker.

Specialfallet triangel

Algoritm 1: Vi kan uttrycka punkten P i triangelkoordinater (u,v) genom att lösa ett litet ekvationssystem. Vi behöver därefter bara kontrollera om $0 \le u$, $0 \le v$ och $u+v \le 1$.

Algoritm 2: En helt annan algoritm räknar antalet skärningar mellan en högerriktad horisontell linje från den aktuella punkten och triangelsidorna.



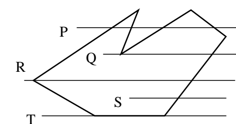
Är antalet skärningar 1, ligger punkten inuti, annars ligger den utanför. Test sker mot triangelsida efter triangelsida, dvs vi har ett problem av samma slag som i avsnittet om linjers skärning, som dock underlättas något av att den ena linjen är horisontell. Beträffande ett par komplikationer se nedan.

Beräkningsgeometri: Punkt i polygon 2(3)

Allmänt

Vi kan göra varianter av båda de tidigare algoritmerna. Låt oss börja med den sista.

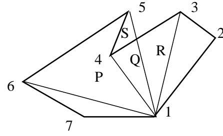
Algoritm 2: På samma sätt som förut räknar vi antalet skärningar mellan en horisonell linje och polygonsidorna. Är antalet udda ligger punkten inuti, annars ligger den utanför. Detta bygger på att när den horisontella linjen passerar en sida går man in i eller ut ur polygonen.



T ex ger punkten P i figuren fyra skärningar och ligger utanför medan S ger en skärning och ligger innanför. Om linjen går genom ett av polygonens hörn stämmer vårt resonemang inte riktigt. T ex ger R skärning med tre sidor, men ligger utanför. Förklaringen är att i polygonhörnet längst till vänster får vi bara en inpassage och bara en av de två skärningarna där skall räknas (ett annat sätt att inse det är att tänka sig R flyttad litet i höjdled vilket inte påverkar utanförskapet). I fallet Q skall däremot varje skärning räknas eftersom hörnet ger en utpassage följd av en inpassage. Detta kan uttryckas som att för spetsar i x-led tar man bara med en av skärningarna, medan för spetsar i y-led båda skall räknas. Horisontella kanter ger ju egentligen oändligt många skärningar, men ignoreras helt. T ex ger T två skärningar och ligger utanför. En likartad situation uppstår när det gällde rastrering av en allmän polygon.

Beräkningsgeometri: Punkt i polygon 3(3)

Algoritm 1: Vi väljer något hörn och tittar efter om punkten ligger i någon av de successiva trianglar som bildas, dvs i figurens fall 123, 134, 145, 156, 167.



Är polygonen konvex överlappar inte trianglarna och bildar tillsammans polygonen. Då räcker det att testa mot triangel efter triangel och att avbryta vid eventuell träff. Är den icke-konvex, som i figuren, måste samtliga trianglar undersökas och är antalet träffar udda så ligger punkten inuti polygonen, annars utanför. T ex ligger P och R i var sin triangel och ligger inuti polygonen. Q ligger i tre trianglar, 134 och 145 och 156, och innanför polygonen. S ligger i två trianglar, 145 och 156, och utanför.

Båda algoritmerna har en tidskomplexitet av O(N), där N är antalet sidor i polygonen. I allmänhet anses Algoritm 2 vara den snabbaste.

Beräkningsgeometri: Area och omkrets

En polygon i xy-planet med hörn $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$, ..., $\mathbf{P}_{n-1}$ och $\mathbf{P}_n$=$\mathbf{P}_0$, där $\mathbf{P}_k = (x_k,y_k)$ är given. Det är välbekant att för omkretsen gäller

$$Omkretsen = \sum_{k=1}^{n} \sqrt{(x_k - x_{k-1})^2 + (y_k - y_{k-1})^2}$$

Det är ju bara att summera de enskilda kanternas längder för vilka vi använder avståndsformeln.

Mindre känt är att arean är

$$Arean = \frac{1}{2} \cdot \left| \sum_{k=1}^{n-1} \langle x_k \cdot y_{k+1} - x_{k+1} \cdot y_k \rangle \right|.$$

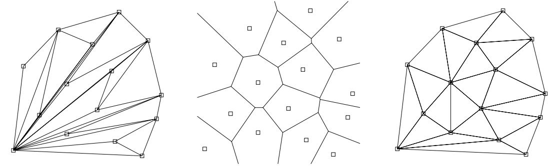
Förvånansvärt nog är denna formel “enklare” än den för omkretsen.

Det är däremot avsevärt svårare att bevisa ytformeln (man har nytta av begrepp som ytintegral och kurvintegral från den flerdimensionella analysen). Beviset är inget för oss. I enklare fall dock direkt ur motsvarande formel för en triangel (se t ex Beta).

Det kan anmärkas att OpenGL använder summan för att bestämma en polygons framsida. Och att summan också är en beståndsdel i det andra sättet att bestämma en normal till en polygonyta. Båda sakerna har vi varit inne på tidigare.

Beräkningsgeometri: Delaunaytriangulering

Problemet är att givet en N st punkter bilda ett triangelmönster med hörn i punkterna. En tillämpning kan vara att man har mätvärden i punkterna och vill interpolera fram värden i andra punkter. Linjär interpolation över trianglar gör man ju lätt. En annan är FEM (Finita Element Metoden). En tredje, s k morfing. I figuren nedan är de givna punkterna markerade med små fyrkanter. Längst till höger visas en lyckad triangulering.



Rakt på

Med N st punkter finns det $N^2/2$ möjliga kanter. En algoritm är att gå igenom dessa och rita kanter som inte skär redan ritade (återigen har vi nytta av linjeskärningsalgoritmer). Detta leder till ett resultat av typen i vänstra figuren ovan, dvs i allmänhet inte så bra.

En girig metod

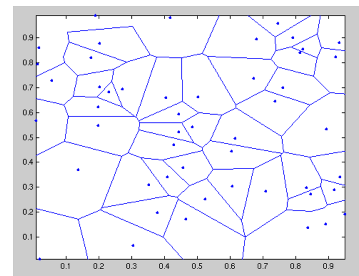
En bättre metod är att sortera de tänkbara $N^2/2$ kanterna i växande ordning och sedan gå igenom följden och rita kanter som inte skär redan ritade. Härigenom reduceras risken att trianglarna får långa sidor. Sorteringen (med t ex kvicksorteringen) är en $O(N^2 \log N)$ process. Resten av metoden kan (med knep) utformas så tidskomplexiteten totalt är sådan. Resultatet kan bli ungefär som i högra figuren ovan.

Delaunaytriangulering

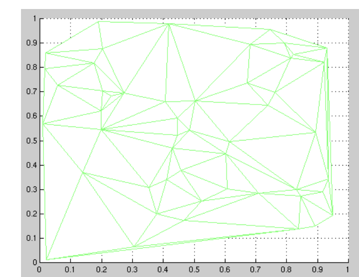
En bättre men mera komplicerad metod kallas Delaunaytriangulering. Det är en sådan som visas till höger ovan. En viktig egenskap är att metoden maximerar den minsta triangelvinkeln, dvs den ger mindre spetsiga trianglar än någon annan metod. Man kan konstruera algoritmen så att den får tidskomplexiteten O(N log N). Vi kan inte här gå in på några detaljer utan anger bara huvudstegen. Först bildas ett s k Voronoi-diagram, se mellersta figuren ovan. Till varje given punkt bildar man en Voronoi-polygon som utgörs av de punkter i planet som ligger närmre den givna punkten än någon annan. Att det rör sig om en polygon är uppenbart eftersom man successivt skär bort halvrymder. Nästa steg är att sammanbinda de ursprungliga punkter som har en gemensam polygonkant. Under rätt allmänna betingelser erhålles en triangulering. Det bör påpekas att spetsiga trianglar och långa triangelkanter inte alltid kan undvikas.

Voronoi/Delaunay med MATLAB

```
>> x = rand(1,50); y = rand(1,50);
>> voronoi(x,y)
```




```
>> tri = delaunay(x,y);
>> trimesh(tri,x,y, zeros(size(x)))
>> view(2)  % standardbetraktning i 2D
```




```
>>
```

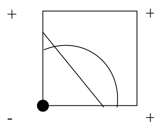
Marscherande kuber (marching cubes) 1(4)

Bakgrund:

- 2D
  1. Hur ritar man allmänna kurvor bestämda av en ekvation f(x,y)=0?
     Ex. $x^2 + y^2 - 1 = 0$ (cirkel)
     Ex. $x^{2/3} + y^{2/3} - 1 = 0$ (asteroid)
  2. Hur ritar man nivåkurvor till 2D-data givna över ett ekvidistant rutnät?
- 3D
  1. Hur ritar man allmänna ytor bestämda av en ekvation f(x,y,z)=0?
     Ex. $x^2 + y^2 + z^2 - 1 = 0$ (sfär)
     Ex. $x^4 + y^4 + z^4 - y^2z^2 - z^2x^2 - x^2y^2 - x^2 - y^2 - z^2 + 1 = 0$ (Kummers yta)
  2. Hur ritar man nivåytor till 3D-data givna över ett ekvidistant kubnät?

För speciella kurvor och ytor kan man hitta en parameterframställning och med dess hjälp lätt plocka fram approximerande trianglar (t ex). Det gäller ju cirkel- och sfärfallet men även för asteroiden ($x = \cos^3(t)$, $y = \sin^3(t)$). Men vi är intresserade av allmänna kurvor och ytor.

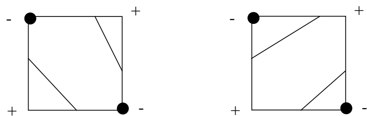
En idé är att bestämma en begränsande kvadrat till kurvan resp en kub till ytan och dela in kvadraten i mindre kvadrater resp kuben i mindre kuber och bestämma en approximerande linjär kurva resp yta för resp delobjekt. Låt oss hastigt se på en sådan situation i 2D. Funktionen antages beräknad i hörnen med värden vars tecken anges i figuren



Den verkliga funktionskurvan kan se ut som bågen i figuren. Men för små delkvadrater ligger den i närheten av den approximerande räta linje som ritats. Ändpunkterna till denna beräknas lätt med linjär interpolation.

Marscherande kuber (marching cubes) 2(4)

I figurens fall finns bara en tolkning, men det finns också situationer där mer än en tolkning är möjlig, t ex



Med tillräckligt små delkvadrater uppstår inte detta problem.

Låt oss övergå till det allmänna fallet i 3D. Det finns två huvudalternativ

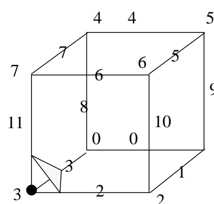
- Strålföljning
- Marscherande kuber

Vi vet att strålföljning ger ett strålande resultat (och man kan i allmänhet göra som för sfärer även om det blir besvärligare att lösa ekvationerna), men är en kostsam metod, så låt oss koncentrera oss på "marscherande kuber". Vi tänker oss att ytan är sluten (inte så viktigt; mest för formuleringarnas skull) och att

- $f(x,y,z) = 0$ på ytan, $f(x,y,z) < 0$ innanför och $f(x,y,z) > 0$ utanför

Förberedande steg: Bestäm ett rätblock inom vilket ytan (eller den intressanta delen av den finns). Dela in denna i t ex 100x100x100 delkuber.

Algoritmens steg per delkub:

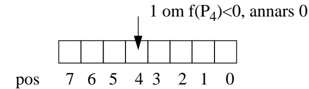


$f(P_i) \geq 0$ för alla hörnpunkter utom P_3.

Marscherande kuber (marching cubes) 3(4)

1. Bestäm vilka av hörnen som ligger inuti objektet. Leverera svaret i form av ett 8-siffrigt binärt tal

1 om $f(P_4)<0$, annars 0



pos 7 6 5 4 3 2 1 0

Ex: Om enbart hörn 3 inuti => 00001000 = 8
Ex: Om hörnen 0-3 inuti => 00001111 = 15

2. Slå nu i en förpreparerad tabell upp vilka kanter som skärs av ytan. Som ingångsnummer används det framräknade talet och raderna nedan är märkta med det. Därefter följer en uppgift om antalet trianglar och sedan för varje triangel motsvarande kantnummer.

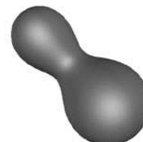
```
  0   0
......
  8   1 3 11 2
......
 15   2 8 11 10 8 10 9
......
255   0
```

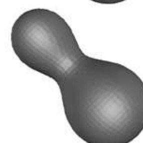
3. Bestäm nu med linjär interpolation varje skärningspunkt. Vi får därmed hörnpunkterna för samtliga trianglar som berör kuben. Maximalt behövs 5 trianglar per kub.
4. Rita upp trianglarna. För att resultatet skall bli bra behövs belysning.

Även i det tredimensionella fallet finns tvetydiga fall.

Referens: Paul Bourke (http://astronomy.swin.edu.au/~pbourke/), som gjort massor med bra datorgrafikmaterial. Jag har bearbetat hans program och läst en uppsats. Det finns andra utformningar av algoritmen. Metoden presenterades 1987 av W.E.Lorensen, men har senare bearbetats av många.

MK: Ett exempel från Paul Bourkes sidor 4(4)





Ytan är en s k blob (eller snarare två).

F ö från
Comp.Graphics.Algorithms
Frequently Asked Questions
http://www.exaflop.org/docs/cgafaq

Subject 5.11: What is the status of the patent on the "marching cubes" algorithm?

United States Patent Number: 4,710,876
Date of Patent: Dec. 1, 1987

Inventors: Harvey E. Cline, William E. Lorensen‘
Assignee: General Electric Company

Title: "System and Method for the Display of Surface Structures Contained Within the Interior Region of a Solid Body"

Filed: Jun. 5, 1985

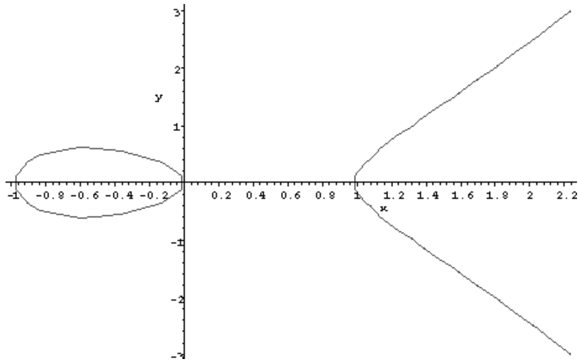
På andra håll:

"I believe the patent expires 17 years after issue."

"Not much we can do about that til around 2003 or so when the MC patent expires,"

Implicita kurvor/ytor i Maple

```
with(plots);
implicitplot(y^2-x^3+x,x=-3..3,y=-3..3);
```



Just detta fall är ju enkelt att analysera. För givet x har vi en andragradsekvation i y, $y^2=x^3-x$, dvs till varje x finns 0 eller 2 (ev sammanfallande) y-värden. För att lösning skall finnas måste $x^3-x>=0$, dvs $x(x^2-1)>=0$, dvs x>=1 eller -1<=x<=0, precis som figuren visar.

Maple använder en samplingsteknik för sin ritning, vilket förklarar den "fula" kurvan. Samplingstätheten kan visserligen justeras. Exakt hur Maple går tillväga kan man ta reda på genom att studera källkoden för `implicitplot` och de rutiner den använder sig av.

Maple har motsvarande kommando för ytor.

CSG-operationer 1(2)

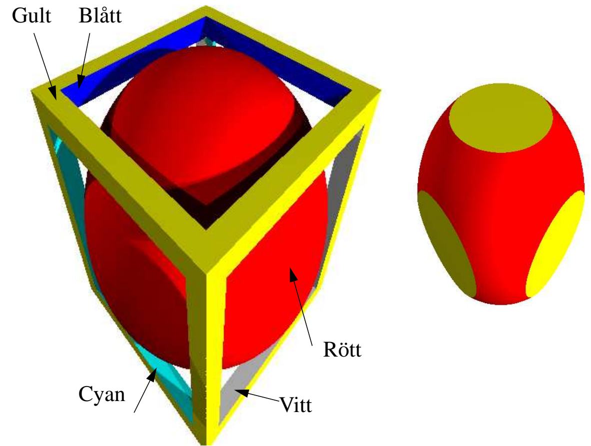
I modelleringsprogram brukar man kunna använda CSG-operationerna (CSG = Constructive Solid Geometry) + (union) och - (skillnad) samt skärning. Kallas även boolska operationer. Matematiken bakom går vi inte in på.

Exempel (POVRay): Som bl a visar hur lätt man kan tillverka “omöjliga” föremål.

```
union {
   // Ett rött klot
   sphere {<0, 0, 0>, 1.75
          pigment { Red }
   }
   // Från en kub subtraheras tre rätblock
   difference {
      object {UnitBox scale 1.5 pigment {Yellow}}
      object {UnitBox scale <1.51, 1.25, 1.25>
             pigment {White}
      }   // "clip" x
      object {UnitBox scale <1.25, 1.51, 1.25>
             pigment {Blue}
      }   // "clip" y
      object {UnitBox scale <1.25, 1.25, 1.51>
                     pigment {Cyan}
      }   // "clip" z
   }
   rotate y*45
}
```

Instoppat i kod med lampor kan resultatet bli det som syns på nästa sida. Väsentligen `granite.pov` men modifierad av trycktekniska skäl.

CSG-operationer 2(2)



Ytfärgen från de subtraherade rätblocken smiter tydligen av sig. Till höger skärningen mellan en sfär och en kub enligt koden

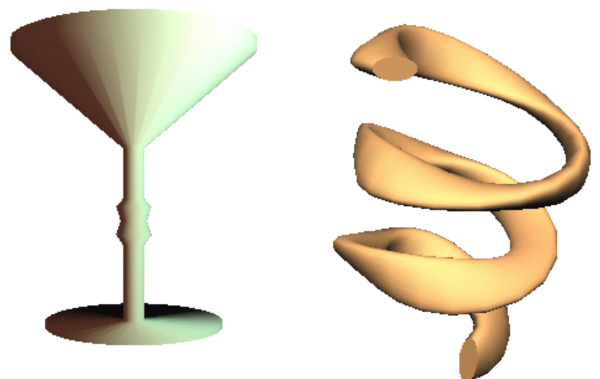
```
intersection {
   sphere {<0, 0, 0>, 1.75
             pigment { Red }
   }
   object {UnitBox scale 1.5 pigment {Yellow}
   }
   rotate y*45
}
```

OpenGL:s GLU har också CSG-operationer men de gäller bara polygoner.

Svep- och extrusionsbibliotek

Detta leder osökt in på två andra operationer vi mött tidigare under kursen.

Linas Vepstas (`http://linas.org/gle/index.html`) skapade 1991 till en föregångare till OpenGL ett sådant bibliotek kallat GLE (tube and extrusion library). I GLUT-distributionen ingår version 3.0.7 (Dec 2001). Via webbplatsen hittar man en senare 3.1.0 (Mars 2003).



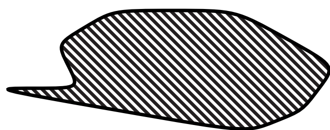
Prova: T ex `$DG/EXEMPEL_GLUT/gle/helix4`) och använd musen.

Egen kompilering (PC):

```
gcc helix4.c  mainsimple.c -lgle -lglut32 -lglu32 -lopengl32
                           -luser32 -lgdi32
```

Målning

Säg att vi har ett slutet område i 2D och vill fylla det med en viss färg eller ett visst mönster. Hur gör man då? Alla ritprogram har naturligtvis en metod för det, men hur kan den tänkas se ut? Alla läroböcker brukar ta upp frågan och ägnar ett par sidor åt den.



Låt oss börja med färgfallet. Vi väljer en punkt i området och kan sedan anropa den rekursiva proceduren

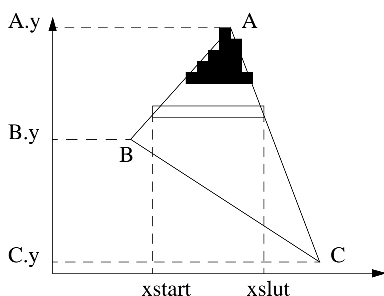
```
void fyll(int x, int y) {
    if (färgen i (x,y) inte är randens färg och inte
        är fyllnadsfärgen) {
        rita punkten (x,y);
        fyll(x+1,y); fyll(x,y+1); fyll(x-1,y); fyll(x,y-1);
    }
}
```

Rekursionsdjupet kan bli stort och metoden är inte så snabb. Den kan effektiviseras på olika sätt.

I mönsterfallet (jag tänker mig någon form av texturmönster som upprepas) kan man arbeta på en kopia av området, som fylls successivt med en genomgående färg. Varje gång en punkt (x,y) i kopian fylls i räknas motsvarigheten i texturen fram (x MOD N etc) och punkten (x,y) i originalet markeras på rätt sätt.

Rastrering

Vi har tittat på rastrering av en linje med Bresenhams algoritm. Även övriga primitiver behöver rastreras. Läroböckerna brukar gå igenom detta i detalj för godtyckliga polygoner och lyckas identifiera en del svårigheter. Men det känns inte särskilt angeläget.
Att rastrera en triangel är däremot problemfritt om än inte trivialt för en nybörjare.



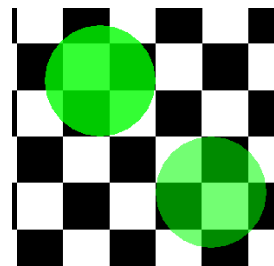
1. Sätt de aktiva kanterna till AB och AC
2. Upprepa för y = A.y, A.y-1,...,C.y
   2.1 Byt aktiva kanter om y=B.y. Räkna ut xstart och xslut. Kan ske med interpolation eller med variant av Bresenham.
   2.2 Räkna även ut färgvärden, djup, texturkoordinater etc för punkterna (xstart,y) och (xslut,y) med någon form av interpolation. Jfr "Från värld till skärm".
   2.3 För x = xstart, xstart+1, ...,xslut
   Räkna ut värden för bildpunkten (x,y) med interpolation. Nu har vi ett fragment i OpenGLs mening. Rita eventuellt (efter bl a djuptest).

Genomskinliga (transparenta) objekt

Genomskinlighet kan, som vi vet från avsnittet 28 om färgblandning i OGL-häftet, åstadkommas genom att vi ritar med ett alfa-värde som är mindre än 1. Det är dock viktigt att de genomskinliga ytorna ritas i rätt ordning, dvs att den närmsta genomskinliga ytan ritas sist.

I praktiken kan det gå till så här:

- Rita alla ogenomskinliga ytor med normalt utnyttjande av djupminnet.
- Se till att de genomskinliga ytorna är sorterade (t ex med BSP) och rita dem i rätt ordning. Använd även nu djupbufferten för att förhindra att en genomskinlig yta som döljs av en ogenomskinlig blir ritad.
  Ett fusksätt enligt OpenGL Prog Guide: Rita de genomskinliga utan sortering, men se till att djupbufferten inte **förändras** av dessa (gör anropet `glDepthMask(GL_FALSE)`; återställning till det normala med `glDepthMask(GL_TRUE)`). Fungerar helt korrekt för t ex enstaka kuber och sfärer. Högra bilden visar två genomskinliga gröna sfärer ritade framför ett schackbräde. Övre sfären ritad utan förändring av djupbufferten, dvs vi ritar två gånger per bildpunkt, undre med förändring av djupbufferten, dvs vi ritar eventuellt bara en gång per pixel. Den senare är något ljusare vilket visar att ritning bara skett en gång per pixel.



Som vanligt är den genomskinlighet som OpenGL åstadkommer halvdan. För perfekt resultat behövs strålföljning.

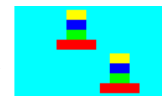
Mera OpenGL: Sprite

Ibland vill man att något litet icke-fyrkantigt objekt (populärt kallat **sprite**) skall röra sig över skärmen. Man kan använda rasterkopiering i kombination med färgblandning (jfr avsnitt 28). Med datatypen `GLbyte` betyder 127 1.0 och 0 0.0. I rastret ser jag därför till att genomskinliga punkter har alfa-värdet 0 och övriga 127.

```
void init {
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
GLbyte M[64]=
    {127,0,0,127, 127,0,0,127, 127,0,0,127, 127,0,0,127,
     0,127,0,0, 0,127,0,127, 0,127,0,127, 0,127,0,0,
     0,0,127,0, 0,0,127,127, 0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127, 127,127,0,127, 127,127,0,0};
int xpos1, xpos2, ypos1, ypos2;
void display(void) {
    glClearColor(0.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2d(xpos1,ypos1);
    glPixelZoom(8.0,8.0);
    glDrawPixels(4,4,GL_RGBA,GL_BYTE,M);
    glRasterPos2d(xpos2,ypos2);
    glDrawPixels(4,4,GL_RGBA,GL_BYTE,M);
    glPixelZoom(1.0,1.0);
    glutSwapBuffers();
}
```

Använd rasterbild (matrisen M)

Gul
Blå
Grön
Röd



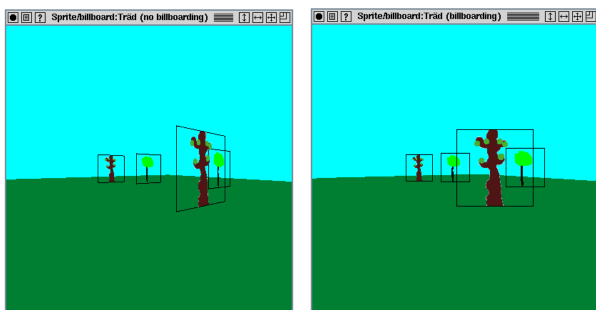
Allmännare med textur. T ex är en känd teknik att rita träd att man ritar två vinkelräta och korsande fyrkanter med en trädtextur där texlar motsvarande trädet har alfavärdet 1 och övriga 0. Mer strax.

Mera OpenGL: Billboarding 1(6)

Betrakta följande bild hämtad från DirectX SDK 9.0 och körd på en föga märkvärdig PC. Man kan röra sig i scenen i realtid. Hur går det till? Vi koncentrerar oss på träden, som är ritade med en teknik kallad billboarding (försvenskat affischering). Träden ritas som rektanglar belagda med en textur. Vi ser till att delar av texturen är genomskinlig genom att förse alla texlar med ett alfa-värde (fjärde komponenten i färgen). T ex 0.0 för genomskinliga och 1.0 för övriga. I scenen nedan används 2-3 olika trädtexturer. Även för marken/himlen används texturering.

Mera OpenGL: Billboarding 2(6)

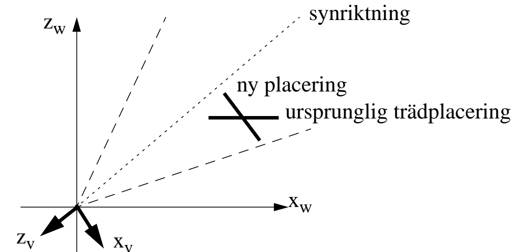
Innan man når fram till fullgod billboarding måste man lösa några problem.



Betrakta bilderna ovan. Scenen är litet enklare och vi har bara två trädtexturer (träd med rund krona respektive knotigt träd). Låt oss placera trädens kvadrater parallella med xy-planet. För tydlighets skull är kvadraternas kanter utritade. Om vi vrider oss runt y-axeln kommer det att se ut som i vänstra figuren och vi inser att i vissa lägen kommer ett träd bara att se ut om ett vertikalt streck. En lösning som antyddes tidigare är att använda två korsande kvadrater för varje träd. En annan är att se till att trädkvadraterna vid rotation följer den så att de hela tiden är vinkelräta mot användaren, se högra bilden. Detta hjälper naturligtvis inte om användaren tar sig före att flyga över träden, för då avslöjas ju fusket.

Mera OpenGL: Billboarding 3(6)

Det förståelsemässigt enklaste sättet att se till att trädkvadraterna är vända mot observatören, når vi genom att titta på följande figur, där scenen är ritad sedd uppifrån och med ett enda träd. Observatören tittar i en viss synriktning (observatörspositionen saknar betydelse för resonemanget; råkade bli origo i världen).



Trädets centrumpunkt skall vara oförändrad. Trädkvadratens över- och underkant skall vara riktad som vykoordinatsystemets x-axel (x_v). Sidokanterna får förbli vertikala (om synriktningen inte är parallell med horisontalplanet är det rimligare att låta dem vara parallella med y_v-axeln). Eftersom vi lätt tar reda på x_v-vektorn ur modellvymatrisen (som ju kan avläsas i OpenGL), kan vi räkna ut trädkvadratens nya hörn och sedan rita den som en GL_POLYGON.

Mera OpenGL: Billboarding 4(6)

Ett alternativt sätt att resonera är att direkt efter *gluLookAt* ta ut rotationsdelen av modellvy-matrisen (beskriver då hur världskoordinater övergår i vykoordinater och innehåller inga skalningar) och invertera den (vridning åt andra hållet), vilket kan göras så här.

Globalt: `GLfloat mv[16];`
I display:

```
glFloat t;
// Efter gluLookAt
glGetFloatv(GL_MODELVIEW_MATRIX,mv);
// Bara rotationsdelen
mv[12]=0.0; mv[13]=0;mv[14]=0;
// Invertera genom transponering
t = mv[1]; mv[1]=mv[4]; mv[4]=t;
t = mv[2]; mv[2]=mv[8]; mv[8]=t
t = mv[6]; mv[6]=mv[9]; mv[9]=t;
```

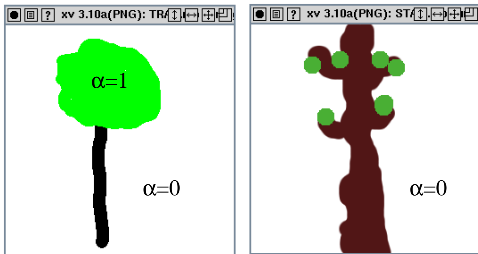
Sedan ritar vi i *display* ett enstaka träd med texturering påslagen:

```
// mittpunktens placering
glTranslatef(x,y,z);
// rotationen
glMultMatrixf(mv);
glBegin(GL_POLYGON);
   glTexCoord2f(0,0); glVertex3f(-0.5,0,  0);
   glTexCoord2f(1,0); glVertex3f(0.5, 0,  0);
   glTexCoord2f(1,1); glVertex3f(0.5, 1.0,0);
   glTexCoord2f(0,1); glVertex3f(-0.5,1.0,0);
glEnd();
```

Mera OpenGL: Billboarding 5(6)

Återstår ett problem. Texturernas genomskinlighet kan vi lösa genom att använda färgblandning (dvs glEnable(GL_BLEND)+glBlendFunc(...)). Men då visar det sig att träd längre bort ibland (beroende på ritordning) döljs av träd framför, vilket beror på att färgblandningen görs efter djuptestet (och djupskrivningen). Lyckligtvis kan man slå på ett annat test, alfa-testet, som görs före djuptestet och använda det i stället för färgblandning.

Vi ser till att den genomskinliga texturen har alfa-värdet 0.0 och att övriga delen har värdet 1.0. Slår på alfa-testet med




```
glEnable(GL_ALPHA_TEST);
```

Dessutom talar vi med

```
glAlphaFunc(GL_GREATER, 0.1);
```

om att fragment med alfa-värde större än 0.1 skall skickas vidare, medan övriga skall ignoreras. Detta gör att bara fragment tillhörande trädkrona och stam skickas vidare och utsätts för djuptest etc. Djupminnet kan se ut som i högra figuren.

Mera OpenGL: Billboarding 6(6)

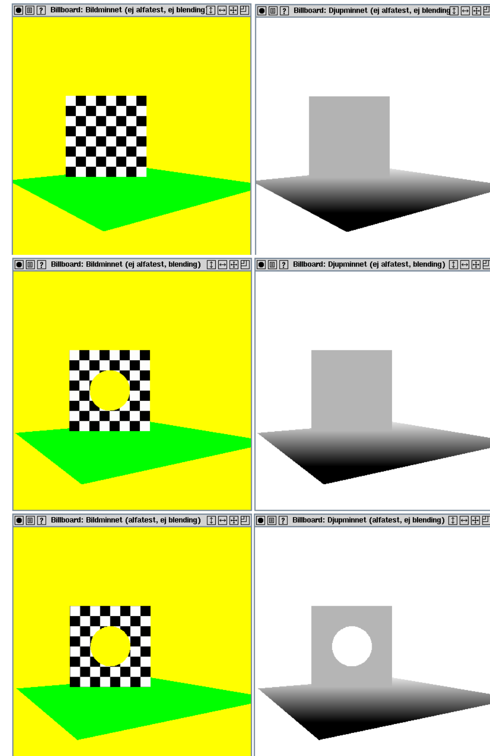
Bilderna på nästa OH har jag tillverkat genom att utöka ett program `billboard.c` gjort av David Blythe och som finns i GLUT-distributionen. Till vänster visas scenen och till höger djupminnet. Schacktexturen har alfavärdet 1 utom i en inre cirkel där värdet är 0. Normal ritning ger översta raden. Ritning med färgblandning mellersta och ritning med alfa-test understa. I det sista fallet återger djupminnet vad som verkligen ritats.

Tekniken kan användas i många andra sammanhang också, t ex i samband med partikelsystem när partiklarna är mer komplicerade än punkter eller trianglar. Det finns varianter av billboarding.

Några praktiska detaljer:

- Vi kan mycket väl läsa in en textur med bara 3 komponenter/texel och sedan ge alla texlar av en viss färg alfa-värdet 0.0 och övriga värdet 1.0. Själv använder jag funktionen *read_texture* från GLUT-distributionens `texture.c` för att läsa in texturer i form av rgb-filer. Har för mig att den automatiskt för in plats för en fjärde komponent om den inte redan finns (eller också har jag ändrat till det).
- Precis som för de vanliga basfärgerna har jag föredragit att tala om alfa-värden mellan 0.0 och 1.0. I praktiken använder vi ju `GLbyte` (C:s `signed char`) eller `GLubyte` (C:s `unsigned char`) för värdena. Härvid motsvarar `(GLbyte)127` värdet 1.0 och `(GLubyte)255` eller `0xff` värdet 1.0 medan 0 motsvarar 0.0.

Mera OpenGL: Blending/alfatest 1



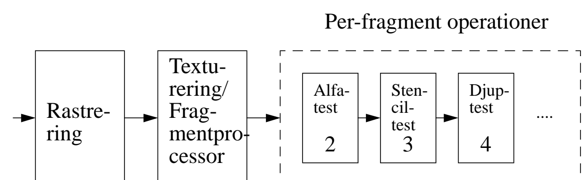
Blending: ingen ritning men djupminnet ändras

Alfatest: ingen ritning och djupminnet ändras ej

Slutet av OpenGL:s rörledning (pipeline)

Vi vet att OpenGL rastrerar, dvs bestämmer vilka bildpunkter som skall ritas och för var och en av dessa djup, färg, texturkoordinater, etc. I OpenGL kallas ett sådant paket med information om en punkt för ett **fragment**. Innan ritning - i bildminnet - sker utförs en mängd ytterligare saker, som i specifikationen kallas *Per-fragment Operations*. Dessa är i tur och ordning (enbart de i fet stil berörs under kursen):

1. Saxtestet (hindrar ritning utanför viss del av fönstret)
2. **Alfatestet** (fragment stoppas p g a sitt alfa-värde)
3. **Stenciltestet**
4. **Djuptestet**
5. **Färgblandning**
6. Dittring
7. **Logiska operationer** (XOR etc)

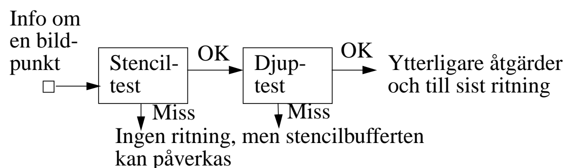


Mera OpenGL: Stencilminne 1(2)

Stencilminnet kan användas på flera sätt. Den vanligaste gäller begränsning av ritning till ett område med godtycklig form. Vi nämnde en annan i samband med BSP.

Stencilminnet kan närmast ses som ett nummerminne. Vi ritar inte i det utan placerar tal i det. Varje gång vi står i begrepp att på vanligt sätt rita i en bildpunkt görs ett test som involverar motsvarande punkt i stencilminnet. Testet kan påverka innehållet i stencilminnespunkten och det kan också förhindra ritning i det vanliga bildminnet.

Djuptestet är det kända, dvs om bildpunktens djup är mindre än det som är lagrat i djupminnet blir resultatet OK (det går att utforma testet annorlunda, men vi berör inte det).



Följande behöver göras.

Mera OpenGL: Stencilminne 2(2)

Nollställ stencilminnet: `glClear (GL_STENCIL_BUFFER_BIT);`

Slå på stenciltestet: `glEnable (GL_STENCIL_TEST);`

Definiera stenciltestet och det värde som eventuellt skall placeras i minnet:

```
glStencilFunc (villkor för OK,
               heltaligt referensvärde, 0xffffffff);
```

Villkoret kan bl a vara `GL_ALWAYS`, `GL_LESS` och `GL_EQUAL`. I det första fallet ger testet alltid resultatet OK. I det andra fallet blir det OK om referensvärdet är mindre än talet i stencilminnet.

Tala om hur stencilminnet skall påverkas vid olika testresultat:

```
glStencilOp(åtgärd när stenciltestet ger Miss,
            åtgärd när djuptestet ger Miss,
            dito när djuptestet ger OK);
```

Värdena kan vara bl a `GL_KEEP` (ändra inte), `GL_REPLACE` (ersätt värdet med det som definierats i glStencilFunc) och `GL_INCR` (öka värdet med 1).

Exempel 1: Räkna antalet ritningar per punkt

```
glStencilFunc(GL_ALWAYS, vad som helst, 0xffffffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
```

Exempel 2: Markera ett område i stencilminnet med 1:or.

```
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
Rita området (man kan förhindra att något syns på skärmen)
```

Exempel 3: Rita något på skärmen men begränsat till området bestämt av stencilminnet (se förra ex). T ex ett runt förstoringsglas.

```
glStencilFunc(GL_EQUAL, 1, 0xffffffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
Rita.
```

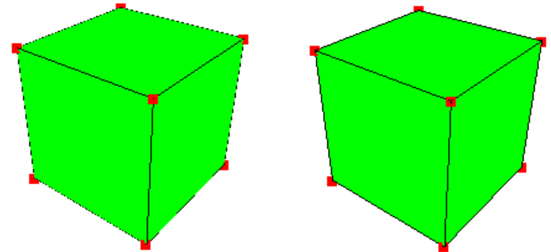

Mera OpenGL: PolygonMode och PolygonOffset

Ibland vill man att kanterna på de ritade polygonerna skall synas. Man kan då som vi gjorde i något tidigare exempel rita kanterna som linjer (GL_LINES). Men behändigare är att byta polygonritningsmod med (nämns som hastigast i OpenGL-häftets avsnitt 7)

```
              GL_FRONT_AND_BACK,  GL_FILL
glPolygonMode(GL_FRONT,           GL_LINE   )
              GL_BACK,            GL_POINT
```

till GL_FRONT_AND_BACK och GL_LINE från de i fetstil markerade standardvärdena.

Men resultatet blir normalt inte riktigt som vi önskar. Se vänstra figuren. Vissa av kantlinjerna verkar streckade eller prickade. Orsaken är djupminnestekniken. Punkterna på ett objekt ritas som standard bara om de är närmre betraktaren än vad som redan ritats. Det betyder att punkterna på kantlinjerna (även om de teoretiskt skulle sig skilja sig en aning från polygonernas inre) på grund av djuprepresentationen inte säkert ritas korrekt (man kan med `glDepthFunc` ställas om till bl a “närmre eller lika”, men resultatet blir ändå likartat).



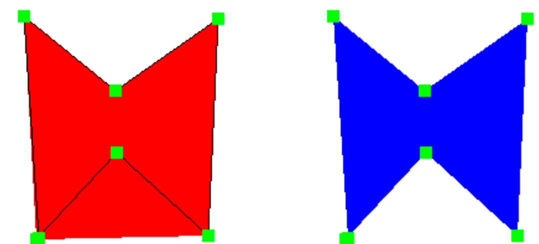
Men vi kan under linjedragningen se till att alla djupvärden minskas något (precis så mycket att de skiljer sig från det tidigare djupvärdet), dvs att punkterna kommer närmare oss. Vi gör detta med en *glEnable* + *glPolygonOffset*.

```
glEnable(GL_POLYGON_OFFSET_LINE);
glPolygonOffset(-1.0,-1.0);
glColor3f(0.0,0.0,0.0);
ritakub();
```

Resultatet blir den högra figuren, som är som den skall.

Mera OpenGL. Tessellering 1(2)

OpenGL klarar konvexa polygoner (glBegin(GL_POLYGON)), men inte (alla) konkava polygoner och inte “polygoner med hål”. För att klara av allmännare fall finns tessellering (eng. tesselate = lägga mosaik). Detta är en “vetenskap” i sig, som dessutom visat sig vara plattformsberoende. Här tar vi bara upp en liten del av det hela



Till vänster har vi sex punkter som skall utgöra hörn i en polygon. Ordningen framgår av de heldragna linjerna. Punkterna har vi samlat i en vektor

`GLdouble P[50][3];`//ej mitt favoritsätt, men gillas av OpenGL

och ritningen har skett med

```
glBegin(GL_POLYGON);
      for (i=0; i<Nr_Of_Points; i++) {
         glVertex2d(P[i][0], P[i][1]);
      }
glEnd();
```

Polygonen är misslyckad eftersom resultatet skulle vara som till höger.

Mera OpenGL. Tessellering 2(2)

Tesselleringskoden (med för enbart fullständighets skull) är tekniskt lik NURBS-kod.

```
GLUtesselator *theTess;    // ett tesselleringsobjekt
```

Man gör diverse initieringar

```
theTess = gluNewTess();
gluTessCallback(theTess,GLU_TESS_VERTEX,
     glVertex2dv);
gluTessCallback(theTess,GLU_TESS_BEGIN, glBegin);
gluTessCallback(theTess,GLU_TESS_END, glEnd);
gluTessCallback(theTess,GLU_TESS_ERROR, tessError);
gluTessCallback(theTess,GLU_TESS_COMBINE, combineCB);
```

Man ritar

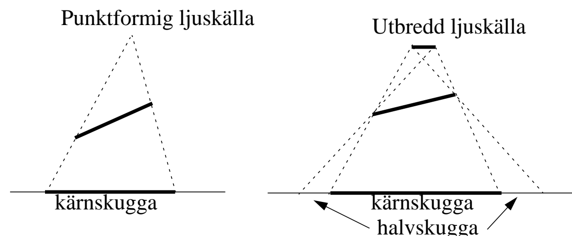
```
void Tessellera() {
      int i;
      glColor3f(0.0,0.0,1.0);
      gluTessBeginPolygon(theTess, NULL);
           gluTessBeginContour(theTess);
           for (i=0; i<Nr_Of_Points; i++) {
              gluTessVertex(theTess, P[i], P[i]);
           }
           gluTessEndContour(theTess);
      gluTessEndPolygon(theTess);
      glFlush();
}
```

Man kan behöva användarskrivna “callbacks”.

```
void combineCB(GLdouble coord[3],GLdouble
  *vertexData[4], GLfloat w[4],GLdouble **dataOut) {
    GLdouble *vertex;
    vertex=(GLdouble *)malloc(3*sizeof(GLdouble));
    vertex[0]=coord[0]; vertex[1]=coord[1];
    vertex[2]=coord[2]; *dataOut = vertex;
}
```


Skuggor 1(5)

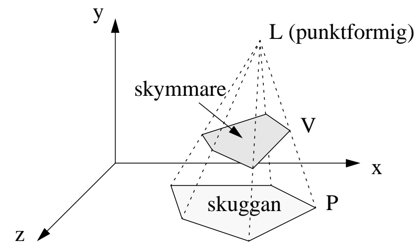
Får vi naturligtvis gratis i strålföljningsprogram, men som alltid vill vi att det skall gå fort. Punktformiga ljuskällor ger ren kärnskugga, medan utbredda ljuskällor ger mjuka skuggor bestående av en kärnskugga (eng. umbra), som inte nås av något ljus från ljuskällan och omgivande halvskugga (eng. prenumbra).



Det finns många metoder för snabb skugg-generering, men ännu ingen given segrare. Oftast blir kodningen något omständlig. Vi belyser bara området översiktligt och förbigår de flesta komplikationerna. En utmärkt sammanfattning av skuggor gjordes av Mark Kilgard vid The Game Developers Conference, 2000. Hittas via NVIDIAs sidor. Objekten i scenen måste delas upp i skymmare (eng. occluder) , som ger upphov till skuggor, och mottagare (eng. receiver), på vilka skuggor bildas.

Skuggor på plana ytor 2(5)

Enklast är det när mottagarna är plana ytor. I figuren tittar vi på ett sådant fall, där mottagaren är planet y=0 (eller möjligen $y=y_0$)



Vi kan gå tillväga så här:

1. Räkna ut skuggpolygonen (se nedan)
2. Rita mottagande plan. Om ej oändligt (t ex bord) fyll även motsvarande platser i stencilbufferten med 1:or.
3. Rita skuggpolygonen med offset (glPolygonOffset(GL_POLYGON_OFFSET_FILL) eller eventuellt avslaget djuptest och kanske i kombination med färgblandning. Rita bara där stencilbufferten har 1:or.

För att räkna ut ett hörn P, utnyttjar vi som ofta förr linjens parameterframställning P = L + t(V-L). Eftersom vi känner $P_y=y_0$, blir $t=(y_0-L_y)/(V_y-L_y)$, som insatt i ekvationerna för x och y ger ($y_0=0$ nu) $P_x=(L_xV_y-L_yV_x)/(V_y-L_y)$ och $P_z=(L_zV_y-L_yV_z)/(V_y-L_y)$. Lätt att generalisera till godtyckligt plan. Kan skrivas på matrisspråk. Skuggan beror enbart på ljuskällan och skymmaren. Skuggpolygonen behöver därför bara räknas om när dessa rör på sig.
Exempel: Programmet *projshadow.c* i GLUT-distributionen.

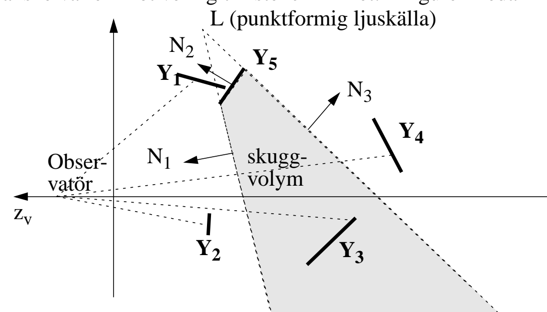
Skuggor: Med texturkarta 3(5)

Här börjar man med att rita scenens skymmare sedda från ljuskällan. Ger en bild som kan sparas som en textur (man kan rita direkt). Sedan ritas scenens mottagare med texturkartan (s k projektiv texturering som OpenGL har stöd för). Därefter skymmarna. Tekniken liknar alltså den som kan användas vid ljussättning i spel.

Exempel: Programmet *projtex.c* i GLUT-distributionen demonstrerar projektion av textur, men inte skuggdelen.

Skuggor: Skuggvolymer 4(5)

Vid sidan av skuggminne (eng. shadow map, som vi inte tar upp) är detta en huvudmetod, som dock kan utformas praktiskt på ett otal sätt. Jag tar upp en listig version som nog hörrör från SGI-kretsar och som kanske var en motivering till stencilminnet. I figuren nedan har vi en



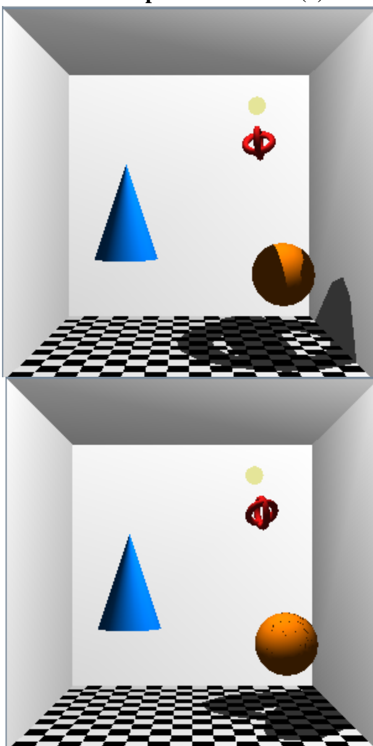
ljuskälla L. Ytan Y_5 är en skymmare. Den bildar tillsammans med strålarna från ljuskällan en halvoändlig avhuggen sned pyramid, som kallas **skuggvolymen**. Enbart objekt helt eller delvis i skuggvolymen

skuggas av Y_5. I figurens fall ligger enbart Y_3 i skugga. För en enstaka bildpunkt ligger motsvarande punkt på den träffade ytan i skugga om strålen från observatören och “genom bildpunkten“ skär skuggvolymens sidoytor precis en gång. Med flera ljuskällor/skymmare i stället ett udda antal gånger.
Algoritm utan praktiska detaljer (1 och 4 kan efter viss modifiering kastas om):
1.Rita scenen normalt.
2.“Rita i stencilminnet“ skuggvolymens framsidor, dvs de som har normalen vänd mot observatören genom att öka motsvarande platser i stencilminnet med 1. Använd djuptest men ändra inte i djupminnet.
3.“Rita i stencilminnet“ skuggvolymens baksidor, dvs de som har normalen vänd från observatören genom att minska motsvarande platser i stencilminnet med 1. Använd djuptest men ändra inte i djupminnet.
4. Rita scenen igen (med t ex enbart omgivningsljus) med djuptestet glDepthFunc(GL_LEQUAL), men enbart om motsvarande stencilpunkt är 1.
Punkterna 1-3 gör att bildpunkten för strålen som träffar Y_4 har stencilvärdet 0, eftersom skuggvolymens fram- och baksida "ritats". Bildpunkten för strålen träffande Y_3 har däremot stencilvärdet 1 eftersom bara skuggvolymens framsida "ritats". I steg 4 kommer därför den senare bildpunkten att förmörkas.
Exempel: Programmen *shadowfun.c* ($DG/EXEMPEL_GLUT/advanced/shadowfun) och *dinoshade.c* i GLUT-distributionen. På nästa sida visas skärmdumpar från körningar av programmet *shadowfun.c* i två olika miljöer, vilket visar på ett av flera problem med denna algoritm. Genom att använda *glPolygonOffset* blir resultatet korrekt i båda miljöerna.

Skuggor: Varför är det på detta viset? 5(5)

Mark Kilgards program shadowfun kört på min PC eller efter ändring av MB på Sun

Ljuskällan är den gula sfär som på bilden syns nära de två ringarna



Samma program kört på Sun: ingen skugga på högra väggen eller på klotet (bara enstaka skuggprickar syns; det mörka på undersidan är belysningsmodellens förtjänst).

Orsak: Man ritar t ex golvet två gånger. Första gången normalt. Andra gången bara den del som ligger i skugga. P g a beräkningsfel klarar inte djupbufferten av att göra rätt.

```
glPolygonOffset(0.0,-1.0); glEnable(GL_POLYGON_OFFSET_FILL)
// Rita polygon GL_POLYGON, GL_QUADS etc
glDisable(GL_POLYGON_OFFSET_FILL);
```


Färg 1(3)

Varje datorgrafikbok brukar ägna ett kapitel åt detta. Ett av de bättre finns i Hills bok. Vi nöjer oss med mycket mindre.
Två huvudproblem: Färgval och färgbeskrivning. Båda är rejält komplicerade.

Färgval. Alla färger passar inte ihop. Den här diskussionen gäller inte så mycket vårt slag av 3D-grafik (med verklighetskontakt) utan främst informationsmaterial av olika slag. Ett utdrag ur en undersökning gjord av Tektronix för många år sedan med ett antal försökspersoner:

Bakgrund	Tunna linjer och text BRA	Tunna linjer och text DÅLIGT	Tjocka linjer och ytor BRA	Tjocka linjer och ytor DÅLIGT
Vit	Blått (94%) Svart (63%) Rött (25%)	Gult (100%) Cyan (94%)	Svart (69%) Blått (63%) Rött (31%)	Gult (94%) Cyan (75%)
Svart	Vitt (75%) Gult (63%)	Blått (87%) Rött (37%)	Gult (69%) Vitt (50%)	Blått (81%) Magenta(31%)
Röd	Gult (75%) Vitt (56%) Svart (44%)	Magenta(81%) Blått (44%)	Svart (50%) Gult (44%) Vitt (44%)	Magenta(69%) Blått (50%)

Beskrivning av färg. RGB är inte ett objektivt format. Det synintryck bilden ger beror naturligtvis på betraktningsförhållandena men än värre på egenskaper hos t ex den skärm som visar upp bilden. Bl a skärminställningarna och lysmaterialet har betydelse.
Det finns en objektiv färgstandard (som kan verifieras mätningsmässigt) kallad CIE-standarden (CIE=Commission Internationale de l’Eclairage) med koordinater XYZ. De normaliserade värdena $x=X/(X+Y+Z)$ och $y=Y/(X+Y+Z)$ representerar kromaticiteten. Man kan göra RGB-systemet mera enhetsoberoende genom att utöka

Färg 2(3)

det med två tal (vitpunkt och gamma) för den skärm som används. Härigenom kan man nämligen transformera mellan RGB och CIE. Vi avstår från alla detaljer. PNG och PDF använder denna teknik.

RGB-systemet lider också av att det inte är ett särskilt naturligt system. Det är svårt att förutse vilken färg som beskrivs.

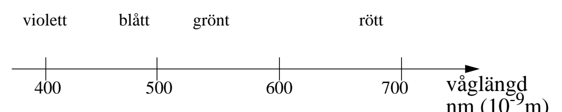
Ett möjligen användaranpassat system är HLS-systemet som innebär att man uttrycker färgen i en dominerande färgton, en ljushet (luminans) och en mättnadsgrad. Mer om den fysiska bakgrunden senare.

RGB-systemet är ett additivt system som passar vid emitterande strålning, t ex skärmar.

CMY(K)-systemet (med basfärgerna Cyan, Magenta och Gult, K=keycolor som vanligen är blacK)) är i stället ett subtraktivt system.

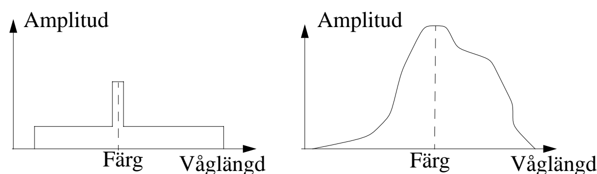
Det finns många andra färgsystem, t ex används ett av färghandeln. I tryckbranschen har länge Pantone varit rådande. Här beskrivs färger med hjälp av omsorgsfullt tryckta färgkartor. En del ritprogram utnyttjar detta.

Litet fysik. Från skolan har vi lärt oss:



Färg 3(3)

Denna skala är vad regnbågen eller ett brytande prisma visar. Men i praktiken är det vi ser en kombination av flera våglängder. T ex finns inte vitt i färgskalan utan uppkommer genom att flera våglängder blandas. Det kan se ut så här



Den vänstra bilden är idealiserad medan den högra är mera verklighetsförankrad. Ett förenklat sätt att beskriva färgen är att ange den dominerande färgen (markerad i figurerna) (kallas på engelska **Hue**), total intensitet (dvs ytan under kurvan) kallad **Luminans** samt kvoten mellan den dominerande färgens energi och den totala mättnadsgraden (eng. **Saturation**). Detta är vad vi har rattar eller knappar för på en TV. Detta är också grunden för det nämnda HLS-systemet.
Ett gammalt problem är hur man förfar när presentationsutrustningen inte har tillräckligt många färger. Redan en tidning som vill trycka bilder med gråskala drabbas av detta, eftersom trycket antingen är vitt eller svart. Lösningen i just den situationen är att trycka större svarta prickar där bilden är mörk och mindre där den är ljus. Kolla med ett förstoringsglas. I datorvärlden har man väl alltid kunnat reglera intensiteten mera kontinuerligt, men för inte så länge sedan var problemet ändå akut inom datorgrafik. Systemet tillät kanske bara 4 eller 16 olika färger, eventuellt fördefinierade. En efterlikning av tidningsmetoden kallas **halvtoning** (det använda mönstret använder olika antal färgade punkter). En variant för dittring (eng **dithering**).

Format 1(3)

Det finns format för modeller (inkl scener), bilder och animeringar etc. Ett format beskriver hur informationen lagras. Ett format kan vara textbaserat eller binärt, okomprimerat eller komprimerat. Många program har sina egna format och den stora frågan är hur t ex geometriinformation skall kunna utbytas. Detta har varit ett stort problem i CAD-världen.

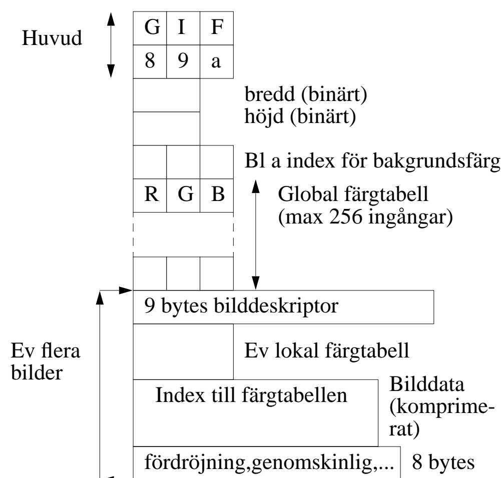
När det gäller modellformat nöjer vi oss med vad vi redan mött. Men det finns många andra, t ex det tidiga `.obj`.

När det gäller bilder är i UNIX-världen olika varianter av formatet PPM (Portable PixMap) vanliga. På nätet dominerar JPEG (Joint Photographic Experts Group) och GIF (Graphics Interchange Format). I PC-världen var det länge BMP (BitMaP) som gällde. För högkvalitativ grafik (hög upplösning), t ex från skannrar, används TIFF (Tag Image File Format). Rit- och bildbehandlingsprogram brukar kunna konvertera mellan några av formaten. JPEG är till skillnad mot de övriga förstörande, i den meningen att det lagrar en approximation till bilden som uppkommer genom att man bl a tar bort höga frekvenser. Ett företag hävdar upphovsrätt till GIF. Som svar på detta skapades det mer avancerade formatet PNG (Portable Network Graphics), som stöds av t ex *Netscape* och *xv*, och en del program, t ex *MATLAB*, sa upp bekantskapen med GIF.

Det mest uppenbara sättet att lagra en (digitaliserad) färgbild vore kanske att lagra RGB-värdena för varje bildpunkt. Vilket skulle betyda 3 bytes/bildpunkt med binär lagring. Detta skulle kunna kallas RGB-format.

Format 2(3)

Låt oss beröra GIF och senare PNG litet mer.



GIF kan visa upp icke-fyrkantiga bilder genom att man låter en färg släppa igenom bakgrundsfärgen (jfr sprites). GIF-animering har vi redan nämnt. Den har sin grund i att en GIF-fil kan lagra flera bilder. GIF uttrycker färger i RGB-systemet, vilket gör att bilden kan se litet olika ut beroende på presentationsutrustning.

Format 3(3)

PNG (uttalas ping) har en något utökad funktionalitet jämfört med GIF. Dock saknas animeringsmöjligheterna, dvs det finns bara en bild per fil. Formatet kan hantera såväl färgtabellbaserade bilder som RGB-bilder. Komprimering görs med samma metod (LZ77) som zip och gzip använder. Animering byggande på PNG finns nu som MNG (uttal ming). PNG och MNG stöds av moderna webbläsare.

En PNG-fil inleds med ett huvud om 8 bytes: \211 P N G \r \n \032 \n. Därefter följer ett antal block (på engelska kallade chunks) med strukturen (CRC=Cyclic Redundancy Check, dvs kontrollinformation).

Längd	Typ	Data	CRC

Varje del utom data-delen är 4 bytes lång. Det första blocket skall ha typen IHDR och innehåller bildens bredd och höjd, bildtyp m m. Exempel på andra block

PLTE: Färgtabell
IDAT: Bilddata
CHRM: CIE-kromaticitet (x,y) för vitpunkt, rött, grönt och blått (32 bytes). Se färgavsnittet.
gAMA: Gamma (4 bytes). Intensitet=(angiven intensitet)$^{\gamma}$.

När det gäller animeringar finns bl a formaten AVI, MPEG och MOV (QuickTime) och som sagt MNG.

Effektivisering med OpenGL. Tidmätning. 1(2)

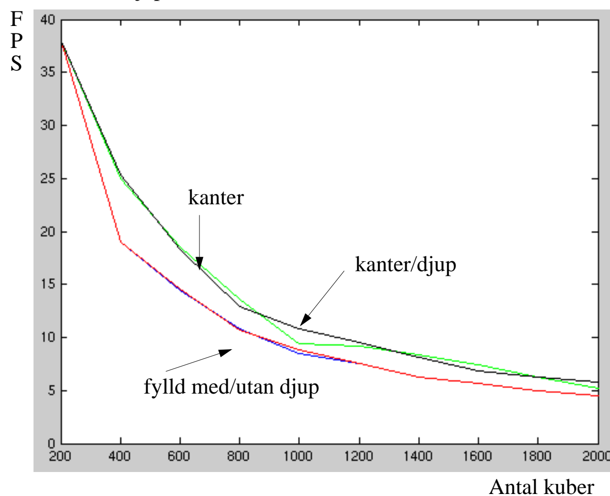
Prestandamätning baseras på tidmätning enligt t ex följande i uppdateringsproceduren. Vi använder funktionen *TickCount* som mäter förfluten tid i millisekunder (OpenGL-häftet, avsnitt 17). I programmet (`TIDER2001_NY.c`) beräknas ett medelvärde för antalet uppdateringar per sekund (FPS=Frames/s) ungefär en gång i sekunden.

```
void display(void) {
    unsigned  long starttime, stoptime, difftime;
    starttime = TickCount();
        glPushMatrix();
        // Radering+gluLookAt+Eventuella transformationer
        // Nu själva ritandet
        myFigure();
        glPopMatrix();
        glutSwapBuffers ();
    stoptime = TickCount();
    Frames = Frames + 1;
    difftime = stoptime - OldTime; // OldTime global
    if (difftime > 1000) {
        printf("MFPS = %f\n", Frames/(0.001*(difftime)));
        Frames = 0; OldTime = stoptime;
    }
}
```

I koden har variabeln *starttime* betydelse bara om man vill mäta tiden för en enstaka omritning.

Effektivisering med OpenGL. Tidmätning. 2(2)

Här visas resultatet av en körning (SUN). Programmet ritar upp ett antal kuber. Diagrammet (gjort med MATLAB) visar antalet uppdateringar per sekund när polygonerna ritas fyllda resp ofyllda, dels med och dels utan djupbufferttest.



Vi ser att antalet kuber som ritas har stor betydelse. Medan djupbuffert eller ej saknar betydelse. Däremot går det litet snabbare om man bara ritar kanterna på sidoytorna. Exempel på MATLAB-kod:

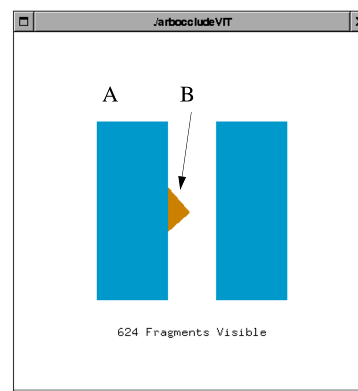
```
>> wire=[38.0,25.3,18.3,12.9,10.8,9.5,8.1,6.8,6.3,5.8];
>> plot(t,wire,'black');
```


Effektivisering med OpenGL

Inom OpenGL finns många möjligheter att effektivisera. Väl så viktigt kan vara att skriva bra C-kod, men det går vi inte in på.

- Remsor och fjädrar. Se OpenGL-häftet, avsnitt 7. I fallet `GL_QUAD_STRIP` behöver vi bara skicka 2 nya hörn per fyrkant i stället för 4 som för `GL_QUADS`. I fallet GL_TRIANGLE_STRIP 1 hörn i stället för tre. I fallet GL_LINE_STRIP bara ett hörn i ställe för två per ny linje. Sparar tid både kommunikations- och transportmässigt.
- Fördröjd ritning (displaylistor). Se OpenGL-häftet, avsnitt 25.
- Polygongallring, som hindrar onödig ritning av frånvända ytor. Redan avhandlat.
- Hörnvektorer (vertex arrays).
- Ocklusionsgallring (eng. occluding). Att ett objekt döljs av ett redan ritat.

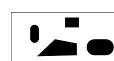
Effektivisering med OpenGL: Ocklusionsgallring 1(1)



Körning av ett program *arboccludeVIT* (finns i `$DG/DEMOS`) i MESA-distributionen

Ett objekt B döljs ibland av ett (eller flera) redan ritat A. Ritningen av B (eller som i figuren delar av) hindras då av djuptestet. Men vi måste ju skicka B:s geometridata till grafikprocessorn. Eller också i vårt eget program avgöra om det föreligger ocklusion eller inte. Ett nyare alternativ (fr o m 1.5-kärnan) är att låta grafikprocessorn sköta avgörandet. Detta går i princip till så här:

1. Sätt med `glBeginQuery(...)` grafikprocessorn i frågeläge och "rita" B (inget ritande i djupminne eller bildminne, vilket kan fixas med `glDepthMask(GL_FALSE)` och `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE,GL_FALSE)`).
2. Vänta på svar, som talar om hur många fragment som är synliga.
3. Sätt grafikprocessorn i vanligt läge och rita B om det inte döljs helt, dvs om antalet synliga fragment > 0.



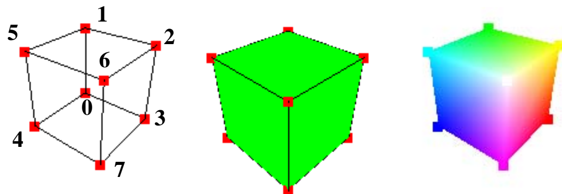
Eftersom vi även nu skickar geometridata (t o m eventuellt två gånger) är inget vunnet. Tanken är i stället att man skall bunta ihop objekt och göra ocklusionstestet för begränsningskroppen. Mer om detta snart.

Effektivisering med OpenGL: Hörnvektorer. Ingår ej 2005.

Hittills har vi skickat en hörnpunkt i taget och tillhörande information. T ex
`glVertex3f(7.5,3.4,1.0);` eller `glVertex3fv(P[2]);`

Men man kan också skicka många hörn i ett svep. Detta sparar proceduranrop. Det kan också tänkas reducera kommunikations- och transformationstid om grafikprocessorn "cachar" hörnen.

Detta belyses bäst med ett exempel. Säg att vi vill rita kuben i mellersta figuren. I första hand sidoytorna men även hörnpunkterna. Hörnet 0 ligger i origo och hörnet 6 i (1,1,1). För detaljer se manualblad.



Nedan visar vi koden vid användning av hörnvektorer (eng vertex array) och tillhörande globala variabler. Nyheterna är fetlagda.

```
GLfloat horn[]={0,0,0, 0,1,0, 1,1,0, 1,0,0,
                0,0,1, 0,1,1, 1,1,1, 1,0,1};
GLuint  index[]={0,1,2,3, 4,7,6,5, 7,3,2,6, 6,2,1,5,
                 4,5,1,0, 0,3,7,4};

void display(void) {
    glClearColor(1.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
```



```
    //*** glEnableClientState(GL_COLOR_ARRAY);
    // Röda hörnpunkter och med storleken 8
    glColor3f(1.0,0.0,0.0);
    glPointSize(8.0);
    glVertexPointer(3,GL_FLOAT,3*4,horn);
    //*** glColorPointer(3,GL_FLOAT,3*4,horn);
    glDrawArrays(GL_POINTS,0,8);
    glColor3f(0.0,0.0,0.0);
    //glDrawElements(GL_LINES,24,GL_UNSIGNED_INT,indexkant);
    glColor3f(0.0,1.0,0.0);
    glDrawElements(GL_QUADS,24,GL_UNSIGNED_INT,index);
    glutSwapBuffers();
}
```

I vektorn *horn* finns koordinaterna för de 8 hörnen. I vektorn *index* finns kvadraternas hörn via index till horn-vektorn. Med anropet av *glDrawArrays* skickas alla hörnen till grafikprocessorn och ritas som punkter. Med anropet av *glDrawElements* skickas alla hörn som det refereras till i index-vektorn och hörnen bildar hörn i successiva fyrhörningar som ritas. Man kan här hoppas på att grafikprocessorn bara behöver transformera nya hörn. För att allt detta skall fungera måste man med *glVertexPointer* definera att just vektorn *horn* skall användas och samtidigt ange vissa layout-data. Dessutom måste ett tillstånd sättas med *glEnableClientState*.
Om vi inför

```
GLuint indexkant[]={1,2, 2,6, 6,5, 5,1, 0,3,
         3,7, 7,4, 4,0, 4,5, 0,1, 3,2, 7,6};
```

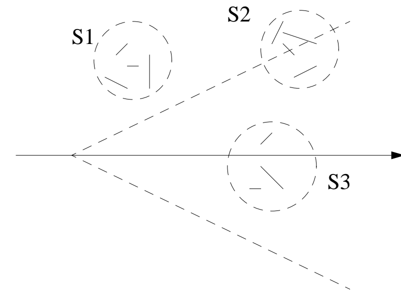
och tar bort kommentartecknen före det näst sista *glDrawElements* kommer även kanterna att ritas. Man kan skicka flera vektorer samtidigt, t ex en vektor med färginformation för hörnen. Då tar man i ovanstående kod bort //*** på två platser. För enkelhets skull använder jag hörndata som färgdata, men det hade gått utmärkt att ha en separat färgvektor. Resultatet blir figuren till höger.

Effektiviseringar i applikationen

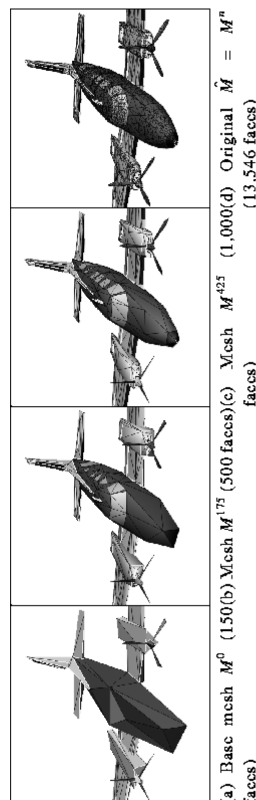
Det finns åtskilliga principiella effektiviseringar som tills vidare bara kan göras i det egna grafikprogrammet. De två viktigaste är kanske **synpyramidgallring** (eng. view frustum culling) och **detaljnivåer** (LOD=Levels of detail).

synpyramidgallring

OpenGL ser ju till att det som ligger utanför den stympade synpyramiden klipps bort. Men detta sker efter det att motsvarande hörn skickats till grafikprocessorn och transformerats, dvs en del arbete har utförts i onödan. Om objekten - i figuren symboliserade med streck - buntas ihop i begränsningsobjekt kan det löna sig att göra egna test i förväg. I figuren finns tre sådana begränsningsobjekt sfärerna S1, S2 och S3.

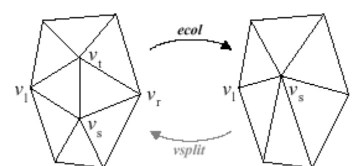


I figurens fall kan vi utan vidare förkasta objekten i S1, dvs vi reducerar antalet hörn som skickas med cirka 30%. Se även "Från värld till skärm", avsnitt 9.



(a) Base mesh M^0 (150 faces) (b) Mesh M^{175} (500 faces) (c) Mesh M^{425} (1,000 faces) (d) Original $\hat{M} = M^n$ (13,546 faces)

Detaljnivåer
Objekt som befinner sig långt ifrån användaren behöver inte ritas med samma noggrannhet som när de är nära. En teknik är därför att ha ett objekt i flera olika upplösningar. Tyvärr störs betraktaren när övergång sker mellan två nivåer.
En lösning på det problemet är **progressiva nät** som infördes av Hugues Hoppe. Man startar med en fin modell och reducerar successivt denna genom kantkollapser (se fig nedan) till grövre och slutar med en grov M_0. Genom att spara information om hur reduceringen gått till kan de finare återskapas genom hörndelning: $M_0 \to M_1 \to \ldots \to M_n = M$



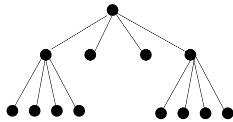
I figuren till vänster är n=6698. Man kan också skapa en mjuk övergång mellan de olika stegen (geometrisk morfing). Enligt Hoppe sparas minne på att utgå från M_0 snarare än M_n.

Kvadrär- och oktalträd 1(2)

Vi skall nu se på ett par “automatiska” sätt att bunta ihop objekt, vilket är en förutsättning för att kunna göra synpyramidgallring och ocklusionsgallring på stora scener.

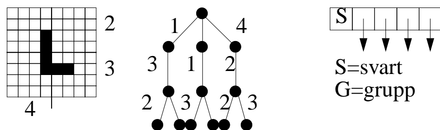
Vi har i denna kurs mött binära träd i samband med BSP. Kvadrärträd (eng. quadtree) och oktal träd (eng. octal tree) är också vanliga i datorgrafik. De kan användas för att öka hastigheten vid uppritandet (och vid kollisionskontroll) och i vissa fall även för att spara minne.

Ett **kvadrärträd** är ett träd med som mest 4 barn per nod.



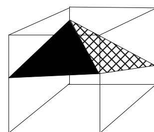
Används typiskt för 2D-kvadrater successivt halverade. I ett **oktalträd** har noderna som mest 8 barn. Används typiskt för kuber successivt halverade.

Exempel: En svart-vit 8x8-bild där de flesta punkterna är svarta kan med fördel representeras med ett kvadrärträd. En tidig rasterskärm från Tektronix (70-talet) använde f ö den tekniken för bildminnet.



Kvadrär- och oktalträd 2(2)

För en digital terrängmodell bestående av

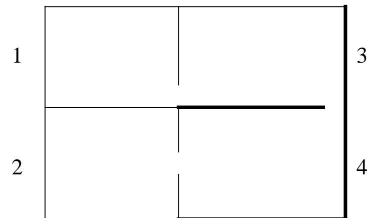


dvs höjder i ett rutnät, säg nxn st (n gärna en 2-potens) passar ett kvadrärträd bra. För varje nod lagrar vi bl a max- och minhöjd och kan med rekursion göra synpyramid- eller ocklusionsgallring på de enskilda blocken. Om vi t ex står så att terrängen inte alls är synlig räcker det med att testa rotnoden.

För en allmännare 3D-scen där vi kan ha flera objekt med samma xy-värden, passar oktalträd. Vi slår in hela scenen i en kub, som sedan successivt halveras. Delkuber som inte innehåller objekt ignoreras. Delkuber som innehåller objekt testas rekursivt vid synpyramid- eller ocklusionsgallring.

PVS. Portaler

I gallringsmetoderna sänder vi en begränsad del av scenen till grafikprocessorn. Denna uppsättning av scenens alla polygoner kallas PVS (potentially visible set) och varierar beroende på var betraktaren befinner sig. Hittills har vi bestämt den dynamiskt under exekveringen. Man skulle också kunna tänka sig att man i förväg beräknade PVS-er för de tänkbara positionerna. Mest omtalat är detta nog för scener med rum (celler) och portaler (dörrar och fönster), vilka förekommer i spel och arkitektprogram.



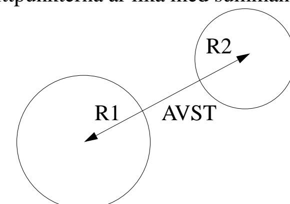
Låt oss först anta att rummen är tomma på föremål. Från rum 2 kan vi som mest se rummets väggar samt vissa av väggarna i rum 4 (fetlagda) och en del av en vägg i rum 3. Motsvarande för övriga rum. Om rummen inte är tomma utökas naturligtvis PVS-arna. Det viktiga är att PVS-arna i princip kan beräknas i förväg och att vi sedan kan välja PVS efter position. Det gäller naturligtvis att som vanligt lagra informationen på ett vettigt sätt, men det kan vi inte gå in på.

Kollisionsdetektering

I scener med rörliga objekt är det för realismens skull viktigt att upptäcka kollisioner och att se till att lämplig fysisk åtgärd vidtages då. Eftersom man “samplar” rörelsen (tiden), kan man ibland missa en aning. Området är aktuellt i bl a spel och robotkinematik.

I en scen med N objekt innebär fullständig kollisionsdetektering att N^2 tester skall göras för varje bildruta. Men ofta är de rörliga objekten få, dvs testantalet kan reduceras till O(N). I allmänna fallet behövs hierarkier och enkla begränsande objekt i form av t ex sfärer eller rätblock. Med rätt datastruktur kan man hitta O(NlogN)-algoritmer.

Enklaste fallet: Sfär mot sfär. Vi får kollision när avståndet AVST mellan de två mittpunkterna är lika med summan av radierna.



Någorlunda enkelt fall: Axelparallella rätblock (AABB=Axis Aligned Bounding Boxes).

Svårare fall: Andra objekt. Flera beräkningsgeometriska problem. Men fallet med snett belägna rätblock (OBB=Oriented Bounding Box) också väl utrett.

Stereografik 1(5)

Vi har nu sett på en mängd olika metoder att få höggradig realism. Emellertid fattas något, nämligen riktigt 3D-intryck. Perspektivtransformationen ger t ex bara en partiell djupkänsla (depth cueing). Denna brist har man i andra sammanhang försökt råda bot på ända sedan människan började kunna producera kopior av verkligheten i form av framför allt foton. Och man har trott sig ha lösningar som skulle bli kommersiella framgångar. Nu har problemet uppmärksammats i datorvärlden. Grundidén i alla lösningar är att producera en bild för vardera ögat och att se till att respektive öga ser bara sin bild. Seriösa intressenter hittar man inom områden som CAD och vetenskaplig visualisering. Men kommersiellt är troligen privatmarknaden (datorspel och TV) väl så betydelsefull. Vi nämner ett par tekniker: färgade glasögon respektive blinkande glasögon. Sensationella system för stereografik dyker upp med en viss regelbundenhet.

Röd-blå glasögon

Man tillverkar en enda bild, som innehåller det vänstra ögats bild ritad i blått (typiskt) och det högra ögats ritad i rött. Observatören utrustas med enkla glasögon, med röd plast för vänstra ögat och blå plast för det andra. Om vi ritar mot en vitt bakgrund, kommer det vänstra ögat att uppfatta bakgrunden som röd och det som är ritat i rött (högra ögats bild) sammanfaller med bakgrunden. Det som ritats med blått, syns emellertid som svart eftersom motsvarande ljus inte släpps igenom. På motsvarande vis uppfattar höger öga det röda som svart mot en blå bakgrund. Härigenom ser vardera ögat enbart sin egen bild. Metoden klarar naturligtvis bara svart-vita bilder, men de kan ha gråskala. Den användes på 50- och 60-talet för framställning av 3D-film, men blev inte någon större succé, kanske delvis beroende på att en annan filmteknik samtidigt kom i allmänt bruk: färgfilmen. Och när

Stereografik 2(5)

svensk television för några år sedan visade ett par av dem, blev det-knappast någon längtan efter mer. Tidningar och bokverk använde samma teknik men den fungerade sällan mera än som nyhetens behag. Med en dator och färgskärm är det lätt att tillverka bilder av denna typ utgående från formlerna nedan.

Blinkande glasögon

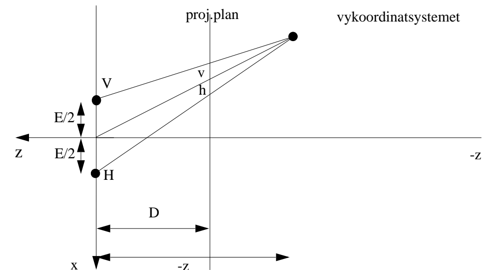
Man visar bilder med dubbla bildfrekvensen, varvid varannan bild är avsedd för vänstra respektive högra ögat. Observatören måste blinka med sina ögon synkroniserat med bildvisningen, vilket torde vara omöjligt, så därför får hon/han ta på sig speciella glasögon i stället. Numera används "blinkande" glasögon av LCD-typ. Synkroniseringen sköts med trådförbindelse eller IR-ljus. Resultatet kan bli mycket bra! (Jag har bara sett det i arbetsstationsmiljö). Många grafikkort har stöd för sådana glasögon. Får jag tippa (1993) så kommer denna teknik att "tvinga" sig på oss så fort som HDTV (högupplösnings-TV) blivit vardagsmat om några år. Tekniken har använts inom filmindustrin och på s k speldatorer (med 3D). I t ex TIME, Nr 16, 1990, berättas om en sådan biograf i Japan. Effekten förstärks där av att filmduken är globformad och jättelik. Göteborg har ju skaffat sig en egen sådan sevärdhet.

Ett par andra sätt som jag kanske nämner, men som inte är värda OH-utrymme: Pulfricheffekten och SIRDS (Single Image Random Dot Stereogram

Enkla stereoskop, föregångare till View Master, var populära redan i fotografins barndom. Man monterade då två bilder i en ställning och försökte få vardera ögat att fokusera på sin bild.

Stereografik: Formler 3(5)

Låt oss börja med att ta fram den erforderliga matematiska formeln utifrån figuren nedan. Hittills har observatören befunnit sig i origo i vykoordinatsystemet. Men nu måste vi särskilja de två ögonen och placerar dem lämpligen symmetriskt kring x=0. Ögonavståndet E är i allmänhet normalt 6-7 cm. En punkt på ett objekt kommer vid projektionen att resultera i två punkter på projektionsplanet, en för vardera ögat. Dessa kommer av symmetriskäl att ligga lika långt ifrån den tidigare enda projektionspunkten, dvs i figuren är v = h.



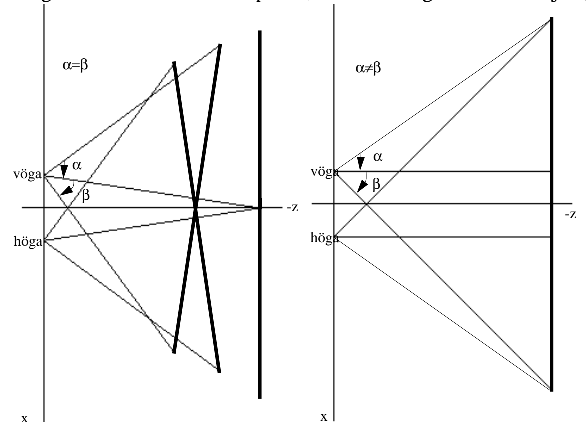
Vi kan beräkna v med likformighet: v förhåller sig till E/2 som -z-D till -z, dvs

$$v = \frac{E}{2}\left(1 + \frac{D}{z}\right)$$

Vi behöver nu bara minska och öka alla x-koordinater med v för att få vänster respektive höger ögas bild. De andra koordinaterna y och z påverkas inte. Vi noterar att stereoseparationen 2v ökar när objektpunktens z-koordinat avtar och är som mest E/2 (när z = -∞).

Stereografik: OpenGL 4(5)

Ett sätt avhandlas i "Introduktion till OpenGL". Det bygger på att båda ögonen tittar mot samma punkt, se vänstra figuren. För varje öga

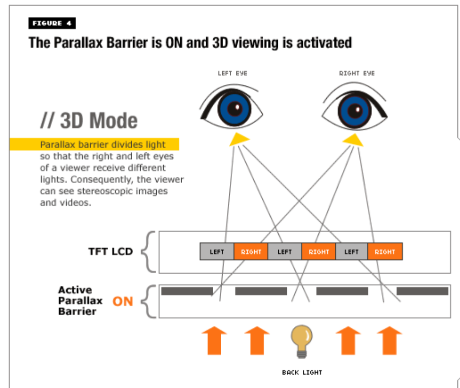


sätter vi med `gluLookAt` upp en symmetrisk synpyramid. Vi projicerar därmed på två olika plan, som jag för tydlighets skull ritat ut något närmare än det egentliga projektionsplanet. Ett korrektare sätt är det som visas i högra figuren. De båda ögonen tittar mot olika punkter. Men det gör också att synpyramiden blir osymmetrisk, vilket gör att `gluLookAt` inte räcker till, utan man måste i stället utnyttja en mera grundläggande procedur `glFrustum`. Vill du ha en utttömmande diskussion kring detta och belysande program, så titta på t ex Paul Bourkes webbsida `http://www.swin.edu.au/astronomy/pbourke/stereographics`. Paul Bourke har under en följd av år gjort i ordning mycket material kring datorgrafik.

Stereografik: Sharps autostereoskop 5(5)

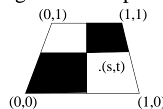
Nu lämnar vi “vetenskap och beprövad erfarenhet”. Hösten 2004 började företaget Sharp sälja en LCD-skärm som inte kräver speciella glasögon. Ett par år tidigare hade man introducerat en mini-version för mobiltelefoner, vilken påstås ha varit en framgång i Japan. Skärmens vanliga LCD-skikt innehåller en bild för vardera ögat. Mellan bakgrundsbelysningen och detta skikt finns en “parallax barrier” av LCD-typ som på något sätt delar upp ljuset så att en del passerar vänster ögas bildpunkter och den andra höger ögas. Man inser att det förmodligen gäller att placera ansiktet på rätt ställe.

Bilden hämtad från `http://www.sharp3d.com/technology/howsharp3dworks/`



Texturkoordinater m m vid rastreringen

Det som är en linje i vår värld fortsätter att vara en linje i projektionskoordinatsystemet och på vår skärm. Det innebär att vi (OpenGL) bara behöver transformera ändpunkterna och sedan kan generera linjen på skärmen. Motsvarande gäller trianglar och polygoner. Men säg att vi har något som varierar linjärt utmed en linje eller över en triangel i den verkliga världen, t ex djup, färg eller en textur (precisare en texturkoordinat). Som framgår av exemplet räcker det inte att linjärin-



terpolera texturkoordinater om vi vill ha perspektivistiskt korrekt texturering (bilden är korrekt), eftersom objekt skall "krympa" när avståndet till betraktaren ökar. I avsnitt 10 i “Från värld till skärm” reds det ut hur man(grafikkretsen) måste gå tillväga, men vi låter det avsnittet utgå och sammanfattar bara resultatet. Texturkoordinaterna är kända i hörnen, t ex för en linje (s_0,t_0), (s_1,t_1). Men vi kan som sagt inte beräkna dem i en godtycklig punkt på skärmen med vanlig linjär interpolation. Däremot med (v_i är djupet, β är linjeparametern)

β=1 (s_1,t_1,v_1)

β=0 (s_0,t_0,v_0)

$$s = \frac{\frac{s_0}{v_0} + \beta\left(\frac{s_1}{v_1} - \frac{s_0}{v_0}\right)}{\frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)} \qquad t = \frac{\frac{t_0}{v_0} + \beta\left(\frac{t_1}{v_1} - \frac{t_0}{v_0}\right)}{\frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)}$$

som innebär en flyttalsdivision per ny bildpunkt och texturkoordinat. Flyttalsdivisioner har hittills alltid varit extremt dyra i förhållande till andra operationer.

Fysikmotorer

Sådana behövs för realistiska rörelseförlopp. Tre som använts kommersiellt kommer från företagen **Havok**, **Novodex** och **Meqon** (svenskt). Under kursens gång har de två sista köpts upp av ett amerikanskt företag **Ageia**, som har annonserat en fysikprocessor (PPU= Physics Processing Unit), som skall påskynda fysikrelaterade beräkningar.
Ett fritt litet enklare system heter **Open Dynamics Engine** (ODE). Presenteras på `www.ode.org` med “ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.”
Vid föreläsning F17 förevisade jag ett demoprogram byggt på en fysikmotor kallad **Newton Game Dynamics** (`www.newtondynamics.com`). På en nyare Windows-dator än min lyser den demon med full glans (kör $DG/PC/NEWTON/newtonplayground/NewtonPlayGround.exe och använd musen). Det visade sig också att min misstanke om att företaget upphört var inkorrekt; länkarna var bara tillfälligt ur funktion. Newton-biblioteket finns för nedladdning till Windows, Linux och Mac. Licensbestämmelserna verkar något oklara. Jag har installerat det under Linux. I distributionen ingick några enklare tutorial-program, som finns färdiga för körning i mapparna $DG/LINUX/NEWTON/newtonSDK/samples/tutorial_*. I avsaknad av källkod går det nog inte att köra de fristående mer avancerade demoprogrammen. Om kvaliteten på Newton uttalar jag mig inte.
Varje sådant här bibliotek för med sig en massa nya metod/funtionsnamn, vilket är en orsak att vi inte går in djupare på dem.

Selektering med gluUnproject 1(4)

GLU-metoden `gluUnproject` nämns i nästan kränkande ordalag i avsnitt 26 i OpenGL-häftet. I det handlar selektering om val av ett visst objekt. Vi går igenom det summariskt. Men ibland vill man peka på ett objekt och få reda på vad motsvarande punkt på objektet har för koordinater (i modellkoordinatsystemet eller ev världskoordinatsystemet). Här kommer `gluUnproject` till vår hjälp. Den omvandlar nämligen en muskoordinat till t ex modellkoordinater.

Exempel: Vi modellerar jordklotet med en snurrande texturerad enhetssfär (texturen tas ej med i koden). Vi vill med musen kunna peka på olika plaser på jorden och få reda på motsvarande koordinater i någon form (här nöjer vi oss med (x,y,z)). T ex skall vi för “nordpolen” få (0,0,1) oberoende av hur roterad sfären är. Kör vi programmet och klickar på nordpolen får vi

```
> GL_UNPROJECT
Modellkoord: (0.016598, -0.008299, 0.992942)
```

Samma resultat om vi först snurrar på sfären (med piltangenterna). Trycker vi utanför sfären, får man något av följande typ.

```
Du pekade utanför
```

Programmet (exklusive kod för piltangenterna och de vanliga include)

```
// Flera av dessa globala variabler kan ligga lokalt i mouse
// Modellkoordinater från gluUnproject
GLdouble wx=0, wy=0, wz=0;
// Vektorer för tranformationsmatriser
GLint viewport[4];
GLdouble mvmatrix[16], projmatrix[16];
// Rotationsvinklar
GLdouble VinkX = 0, VinkY = 0, VinkZ = 0;
```

Selektering med gluUnproject 2-3(4)

```
void InitGL(GLvoid) {
     glClearColor(1.0f, 1.0f, 1.0f, 1.0f);   // Vit bakgrund
     glEnable(GL_DEPTH_TEST);
     glColor3f(1,0,0);   // Röd ritfärg
}

void update() {
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glLoadIdentity();
     gluLookAt(0,0,5, 0,0,0, 0,1,0);
     glRotatef(VinkZ, 0,0,1);
     glRotatef(VinkY, 0,1,0);
     glRotatef(VinkX, 1,0,0);
     glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
     glutWireSphere(1,10,5);
     glutSwapBuffers();
}

void reshape(int width, int height) {
      glViewport(0,0,width, height);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluPerspective(45.0, ((GLfloat)width)/height,
                 0.1f, 10.0f);
      glMatrixMode(GL_MODELVIEW);
}

void mouse(int button, int state, int x, int y) {
   GLint realy;  /*  OpenGL y coordinate position  */
   GLfloat zbuff;
   if (button==GLUT_LEFT_BUTTON) {
      if (state == GLUT_DOWN) {
          glGetIntegerv (GL_VIEWPORT, viewport);
          glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);
          /*  viewport[3] är fönsterhöjden */
          realy = viewport[3] - (GLint) y - 1;

          // Räkna ut z via djupminnet
```



```
          glReadPixels(x,realy,1,1,GL_DEPTH_COMPONENT,
                       GL_FLOAT, &zbuff);
          gluUnProject ((GLdouble)x, (GLdouble)realy, zbuff,
             mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
          if (zbuff>0.9999) printf("Du pekade utanför\n");
          else printf ("Modellkoord: (%f, %f, %f)\n",
                       wx, wy, wz);
          glutPostRedisplay();
      }
   }
}

int main(int argc, char** argv) {
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA |GLUT_DEPTH);
   glutInitWindowSize(400,400);
   glutCreateWindow("Peka på plats");
   InitGL();
   glutReshapeFunc(reshape);
   glutDisplayFunc(update);
   glutKeyboardFunc(key);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

Programmet tar fram modellvymatrisen (i *update* eftersom den ändras fortlöpande) och projektionsmatrisen (i *mouse*). Genom att transformera punktens normaliserade koordinater med inverserna av dessa-matriser kan *gluUnproject* få fram koordinater i modellkoordinatsystemet. Nu känner vi inte den normaliserade z-koordinaten (djupet). Men vi tar den från djupbufferten. Programmet tar fram träffpunkten på ytterligare ett sätt, som inte finns med i listningen och som inte är generellt.

Selektering med gluUnproject 4(4)

Utdrag ur manualblad

```
NAME
gluUnProject - map window coordinates to object coordinates
C SPECIFICATION
GLint gluUnProject(
   GLdouble winX, GLdouble winY, GLdouble winZ,
   const GLdouble *model, const GLdouble *proj,
   const GLint *view,
   GLdouble* objX, GLdouble* objY, GLdouble* objZ )
PARAMETERS
  winX, winY, winZ
    Specify the window coordinates to be mapped.
  model
    Specifies the modelview matrix  (as  from glGetDoublev).
  proj
    Specifies the projection matrix (as  from  glGetDoublev).
  view
    Specifies the viewport (as from a glGetIntegerv call).
  objX, objY, objZ
    Returns the computed object coordinates.
```


Mera OpenGL

Det finns massor av ytterligare guldkorn i OpenGL.

- Ackumuleringsminne. Kan användas för att få rörelseoskärpa och utjämnade bilder.
- Dimma (`man glFog` så får du reda på hur lätt det är att sprida sådan)

Hårdvaruutveckling

Utvecklingen var ett tag mycket rask.

Hösten 1999 presenterades den första grafikprocessor för konsumentmarknaden med inbyggt stöd för transformationer och belysningsberäkningar GeForce. Det var något som inte ens SUNs dåtida grafikkort Creator3D hade (Elite3D har det).

Den grafikprocessor GeForce 3 som NVIDIA började leverera i april 2001 bestod av 57 miljoner transistorer (grovt sett samma som Pentium 4) och uppgavs kunna utföra 76 miljarder flyttalsoperationer på en sekund. Det är mycket mer än vad en vanlig processor klarar (kanske 1 miljard) och låter mycket. Den principiellt viktigaste nyheten var att man själv kunde programmera delar av beteendet (hörn- och fragmentprogrammering). Under år 2002 kom GeForce 4 (OBS! GeForce 4MX är egentligen en GeForce 2) med ännu högre komplexitet och bättre prestanda, liksom nya modeller med likartat funktionssätt från främst konkurrenten ATI (Radeon).

Därefter har det främst handlat om prestandaökning. Och att man funnit nya användningsområden för de olika tilläggen.

Sista bilden. Vad har delats ut?

Allt med OH finns på nätet via kurssidan. OH och övrigt kan fås av mig, så länge jag har restexemplar. Eventuellt kan jag lägga ut någon småskrift tillfälligt.

• Introduktion till OpenGL, 54 sidor. Kostar.
• Datorgrafik OH 1-335. Finns på kurssidan.
• Ett modelleringsprogram - Blender. 8 sidor. Finns på kurssidan.
• Ett annat modelleringsprogram - Art Of Illusion. 2 sidor.
• Från värld till skärm, 20 sidor.
• Kurv- och ytapproximation med polynom, 16 sidor
• PostScript - en introduktion, 12 sidor.
• Fraktaler och kaos, 12 sidor.
• Bildbehandling, 16 sidor (enbart för GU)

Totalt 124 sidor + 335 OH + 16 sidor (GU)

Feltryck

OH 257 Man måste begära Alpha-buffert, som finns på modernare kort, med GLUT_ALPHA i glutInitDisplayMode.

Tyvärr säkert fler.

Labgodkännande: Sista officiella torsdagen den 13 oktober, vilket meddelades i början av kursen. Några få kan få redovisa lab 3 under tentamensperioden. Ännu färre i första läsveckan, då jag dock förmodligen är rätt jäktad. I båda dessa fall efter kontakt med mig.

Inför tentan 1(2)

Tentans utformning: c:a 1/3 kodning, 2/3 teori. Svaren på de större teori-frågorna skall vara förklarande och begripliga för en kamrat som inte läst kursen.

Kursen har presenterats i form av OH, demonstrationer, laborationer, OpenGL-kod och småskrifter.

Demonstrationer och laborationer
Har haft som syfte att belysa annat material. Dvs inga krav på minneskunskaper.

OpenGL-kod
Du skall på ett korrekt sätt kunna tillämpa det material som finns i OpenGl-häftet (tillåtet hjälpmedel). Kodning sker i det språk som du själv valt i kursen. Smärre språkfel ger ej avdrag. När det gäller funktioner/metoder som bara tagits upp på OH, krävs inga minneskunskaper, utan i förekommande fall ges tillräcklig hjälp på tentan.

Småskrifterna
Från värld ...: Du skall kunna återge och praktiskt tillämpa materialet i avsnitten 1-4 och 8-9. Avsnittet 10 ingår bara som OH 326. Avsnitt 11 inte alls. Övriga avsnitt är exemplifierande.
Kurvor och ytor:Du skall kunna återge och praktiskt tillämpa materialet i avsnitten 1, 3-12.
Fraktaler: Du skall kunna beskriva de begrepp som införs i häftet. Avsnitt 8 ingår inte.
Postscript: Utdelat men ej diskuterat, så det får väl utgå, om det inte genom ett under blir tid över på F18.

Inför tentan 2(2)

Bildbehandling (enbart GU): Häftet ingår i sin helhet. Du bör känna till begreppet fourierkoefficient, men behöver inte kunna några formler för dem.

OH
Du skall kunna beskriva de begrepp som successivt införs. Du skall kunna återge och praktiskt tillämpa de resonemang som förs kring bl a: rastering av linjer, 2D- och 3D-transformationer, dolda ytor, navigering, fotorealism OH 92-112, uppdelningsmetoder OH 123-124, Perlin-brus OH 190-196, vertex- och fragment OH 231-239, OH 245-252 (ev program blir enkelt), marscherande kuber, billboarding, skuggor, Följande OH kan du lugnt hoppa förbi: 54-61, 130-131,167-169, 241-244, 268-269, 296-297, 313-314.