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.

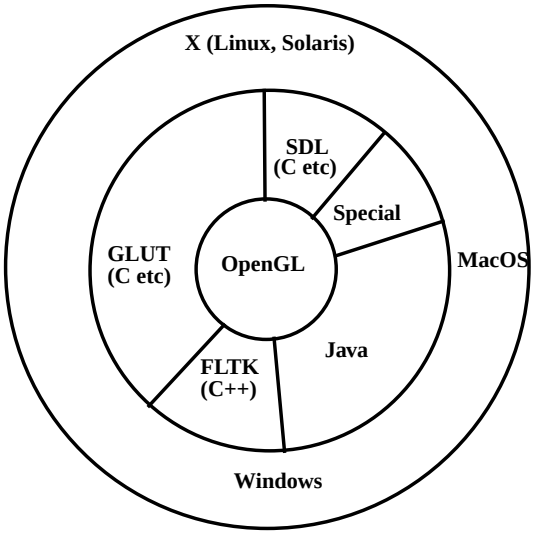
Litteratur
Kursen täcks med diverse småskrifter och OH-material. Vid senaste genomförandet av kursen rekommenderades bl a Hill: Computer Graphics Using OpenGL, 2nd ed, Prentice-Hall 2000 alternativt Akenine-Möller/Haines: Real-Time Rendering, 2nd ed, A K Peters 2002. Aktuell information om litteratur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

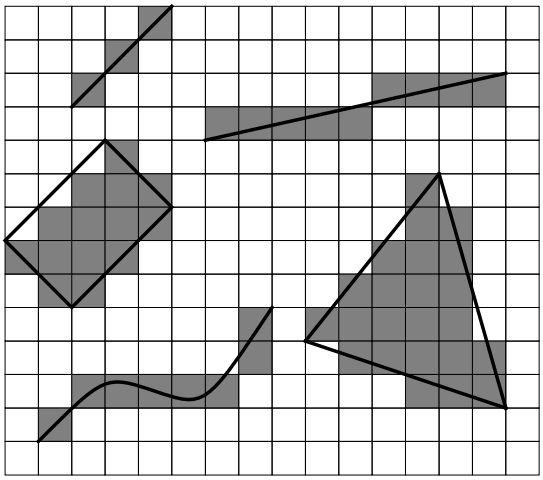
GLUT <-> JOGL (Java bindings for Open GL))

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)).

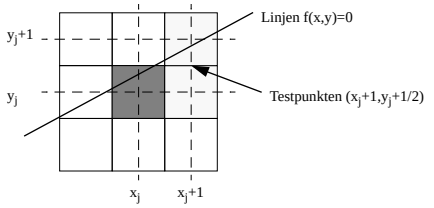
Nackdel: Flyttalsaritmetik. Men det kanske inte numera är en nackdel.

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
   $\mathbf{q_0} = \Delta \mathbf{x - 2\Delta y}$ (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
   $\mathbf{q_{j+1}} = 2\Delta x f(x_j+2,y_j+3/2) = 2\Delta x f(x_j+1,y_j+1/2) - 2\Delta y + 2\Delta x = \mathbf{q_j + 2(\Delta x - \Delta y)}$

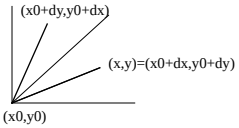
Rastrering - Linjer 3(3)

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**:
  $$\left.\begin{aligned} x &= x(t) \\ y &= y(t) \end{aligned}\right\}, \quad t \in [0,1]$$
  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:

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

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)

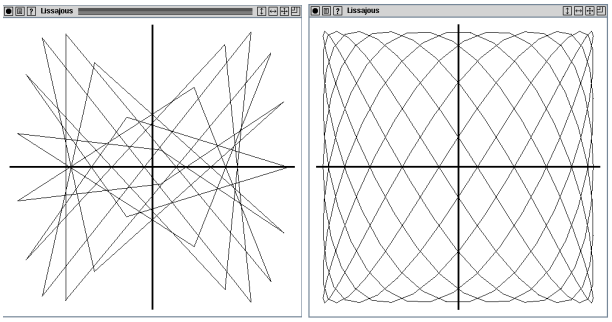
Algoritm (i 2D):

- 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:

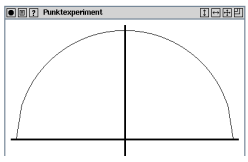


För fullgott resultat måste N väljas stort nog.

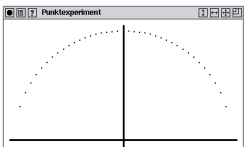
Uppritning av allmänna kurvor 3(3)

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.

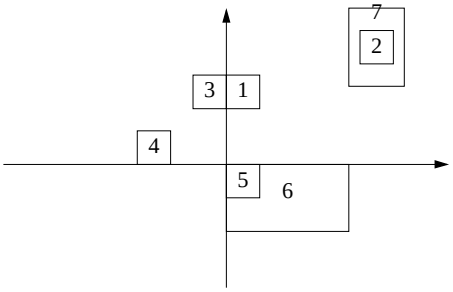
Det finns många andra sätt att rita en cirkel.

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).

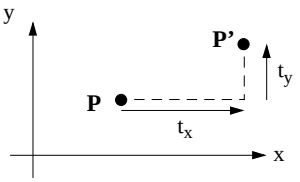


Kvadraten 2 i figuren har vi fått genom att translatera kvadrat 1. Kvadrat 3 genom att rotera kvadrat 1 kring sitt undre vänstra hörn. Kvadrat 4 genom att rotera kvadrat 1 runt origo. Rektangel 6 genom att skala kvadrat 5 med origo som fixpunkt. Rektangel 7 genom att skala kvadrat 2 med mittpunkten som 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

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

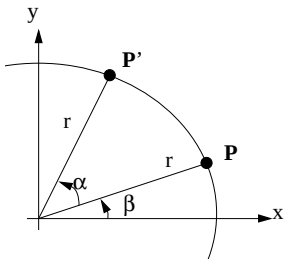
eller

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

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**

$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ eller allmännare $\begin{bmatrix} wx \\ wy \\ w \end{bmatrix}$ 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

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

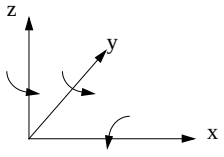
och vid skalning

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

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)

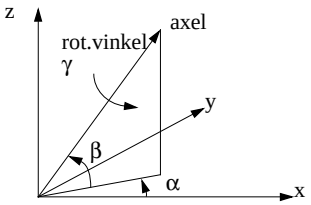
$$\mathbf{P'} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}$$

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).

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

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).

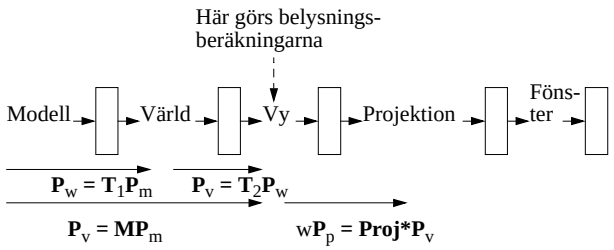
Efter det fortsätter vi med OH-bilder om lösning av dolda yt-problemet.

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 $\mathbf{P}'$ = **Proj*****V*****Mod*****P** med **P**=[x y z 1]^T. och tänkt **P**' = [wx',wy',wz',w]^T. Det blir mycket tydligare om man som vi gjort här skriver $w\mathbf{P}_p$ = **Proj*****V*****Mod*****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);
```

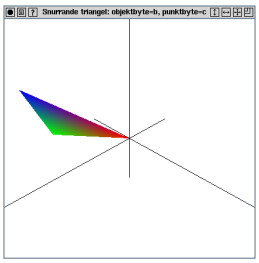
Ersätter aktuell matris med innehållet i `GLfloat v[16]`.

```
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.

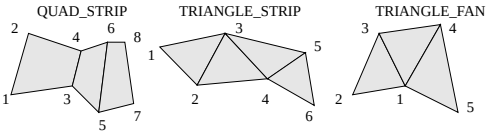
Geometriska objekt 1(2)

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.

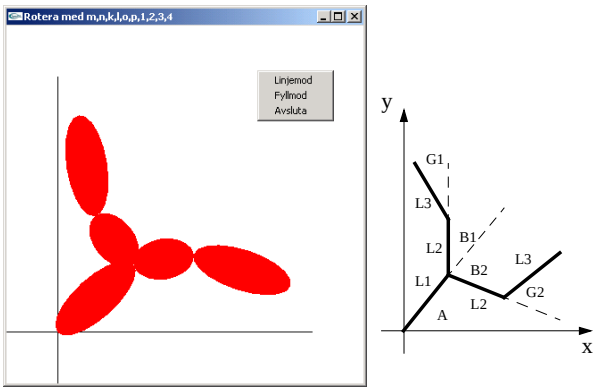
```
void glutWireSphere(GLdouble radius,
                    GLint slices, GLint stacks);
void glutSolidSphere(GLdouble radius,
                    GLint slices, GLint stacks);
```

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();
}
```