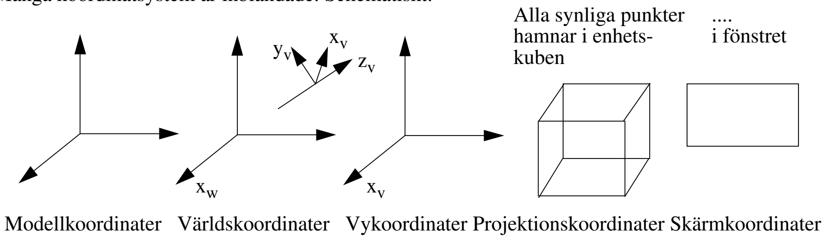
FRÅN VÄRLD TILL SKÄRM

Magnus Bondesson
2000-01-21, 2001-01-08, 2002-01-22, 2002-03-07

1 Översikt

Det här pappret handlar om hur koordinaterna för ett objekt i en tredimensionell värld översätts till koordinater i ett tvådimensionellt plan, typiskt en datorskärm. Mera påtagligt gäller det att bestämma vilken bildpunkt som motsvarar en tredimensionell punkt. Vi använder grundläggande linjär algebra, vilket inte hindrar att det kan kännas tungt. Trösta dig med att grafikprocessorn får jobba med sådant här för varje punkt som du anger!

Många koordinatsystem är inblandade. Schematiskt:



Världskoordinatsystemet (world coordinate system) är det system i vilket objekten befinner sig. Observatören betraktar objekten från en viss punkt (ögats position) och med en viss synriktning och inför härvid ett nytt koordinatsystem, **vykoordinatsystemet** (viewing coordinate system) eller **ögonkoordinatsystemet** med z-axeln i synriktningen och med origo vid ögat. Vid perspektivprojektion tänker man sig ett **projektionsplan** på avståndet D framför ögat. För enkelhets skull tänker vi oss tills vidare att alla objekten ligger framför det planet. Vykordinatsystemet är i vissa system ett vänstersystem (med positiva z-axeln i synriktningen), i andra ett högerkoordinatsystem (med positiva z-axeln i andra riktningen). I OpenGL har vi ett högerkoordinatsystem, dvs vykoordinatsystemets z-axel är motriktad synriktningen. Vi tittar alltså i negativ z-led. Vi förutsätter genomgående i fortsättningen att det är så.

Modellkoordinatsystemet är det system i vilket vi tillverkar modeller för objekten. T ex en grundmodell för ett hus som sedan skalas och roteras innan det med translation placeras i världen. I enklare fall kan vi bortse från detta system.

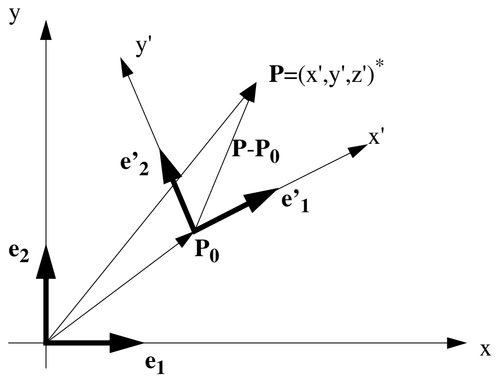
Vi betecknar världskoordinater ibland med index w (som i world), vykordinater med index v och projektionskoordinater med index p.

Övergången från modellkordinatsystem till världskoordinatsystem görs på ett uppenbart sätt med de grundläggande transformationerna skalning, translation och rotation och berörs inte vidare här.

I ett 3D-system behöver vi inte alls tänka på detaljerna i transformationerna. Det är självklart som det skall vara. Men det är olyckligt pedagogiskt sett, eftersom man bör ha mer än ett hum om dem.

2 Byte av koordinatsystem, speciellt från världskoordinatsystemet till vykoordinatsystemet

Vi har två rätvinkliga koordinatsystem, xyz-systemet respektive x'y'z'-systemet. Skalningen tänkes vara densamma i bägge. En punkt **P**$=(x,y,z)^*$ (jag skriver * i betydelsen transponat ibland för att få kolumnvektorer på litet utrymme) har koordinater givna i det första systemet. Vi vill ha reda på koordinaterna $(x',y',z')^*$ för **P** i det andra. I den vanligaste datorgrafiska tillämpningen är världskoordinatsystemet det första och vykoordinatsystemet det andra. Följande figur illustrerar i 2D det problem som här formulerats för 3D.



Figur 1: Övergång från ett koordinatsystem till ett annat.

Det finns flera sätt att ta fram övergången.

1. Böckerna

De flesta läroböcker i datorgrafik för ett resonemang som går ut på upprepade rotationer. Omständligt formelmässigt, men inte tankemässigt. Vi tar oss fram på något av följande två andra sätt.

2. Så här kan vi göra i stället

Låt $\mathbf{e}_i$ och $\mathbf{e}'_i$ vara enhetsvektorerna längs axlarna, dvs vektorer med längden 1 och låt $\mathbf{P}_0=[x_0, y_0, z_0]^*$ vara det nya koordinatsystemets origo uttryckt i det ursprungliga. Vi förutsätter att de nya enhetsvektorerna är kända i xyz-systemet, t ex

$$\mathbf{e}'_i = [a_i, b_i, c_i]^T, \ i=1,2,3.$$

För de ursprungliga gäller ju

$$\mathbf{e}_1 = [1,0,0]^* \text{ osv.}$$

Uppfatta punkterna **P** och $\mathbf{P}_0$ som vektorer. Nu är ju koordinaterna i x'y'z'-systemet projektionerna av vektorn

$$\mathbf{P} - \mathbf{P}_0$$

på respektive enhetsvektor $\mathbf{e}'_i$, dvs

$x' = (\mathbf{P} - \mathbf{P}_0)\bullet\mathbf{e}'_1$, $y' = (\mathbf{P} - \mathbf{P}_0)\bullet\mathbf{e}'_2$, $z' = (\mathbf{P} - \mathbf{P}_0)\bullet\mathbf{e}'_3$

där produkterna är skalärprodukter. Annorlunda uttryckt är t ex

$x' = (x-x_0)a_1 + (y-y_0)b_1 + (z-z_0)c_1$

Vi får därför följande formel

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} - \boldsymbol{P}_0 \right)$$

Naturligtvis kan vi här använda s k homogena koordinater för att få det enhetligare skrivsättet

$$\boldsymbol{P}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{P}$$

där talen d_i kan räknas ut (görs i nästa avsnitt).

3. Ett måhända ännu enklare sätt, som vi använde 1999 och 2000

Låt origo för vykoordinatsystemet vara punkten $\mathbf{P}_0=[x_0, y_0, z_0]^*$ i världskoordinatsystemet. Låt enhetsvektorerna för vykoordinatsystemets axlar vara $[a_1, b_1, c_1]^*$, $[a_2, b_2, c_2]^*$ respektive $[a_3, b_3, c_3]^*$ uttryckta i världskoordinater. Dessa har längden 1 och är naturligtvis sinsemellan vinkelräta, dvs skalärprodukterna $\mathbf{e_i}\bullet\mathbf{e_j}$ är

$$\begin{bmatrix} a_i & b_i & c_i \end{bmatrix} \cdot \begin{bmatrix} a_j & b_j & c_j \end{bmatrix} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Låt oss nu se på en godtycklig punkt betecknad $\mathbf{P} =[x, y, z]^*$ i världskoordinatsystemet och $\mathbf{P}'=[x', y', z']^{*T}$ i vykoordinatsystemet.

Om vi accepterar resonemanget i punkt 1, så finns det då en 3x3-matris M sådan att

$\mathbf{P}' = M(\mathbf{P} - \mathbf{P}_0)$.

Välj här $\mathbf{P}$ så att $\mathbf{P} - \mathbf{P}_0 = [a_i, b_i, c_i]^*$, i = 1,2,3. Då måste

$$M\begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad M\begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad M\begin{bmatrix} a_3 \\ b_3 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Eftersom vektorerna $[a_i, b_i, c_i]^*$ är ortonormala följer att

$$M = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix}$$

Om vi går över till homogena koordinater, dvs t ex

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

så kan vi skriva

$$\boldsymbol{P}' = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ a_2 & b_2 & c_2 & 0 \\ a_3 & b_3 & c_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{P} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{P}$$

där

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = -M\boldsymbol{P}_0$$

3 Hur bestämmer vi enhetsvektorerna för vykoordinatsystemet?

I förra avsnittet betecknade vi dessa enhetsvektorer med $\mathbf{e}'_1$, $\mathbf{e}'_2$ och $\mathbf{e}'_3$, eftersom vi studerade ett allmänt byte mellan koordinatsystem. Nu använder vi $\hat{\boldsymbol{x}}_v$, $\hat{\boldsymbol{y}}_v$ och $\hat{z}_v$.

Enhetsvektorn $\hat{z}_v$: I de flesta system anges ögats position och en punkt mot vilken man tittar. Skillnaden ger oss en vektor som efter normalisering blir den sökta vektorn. Om man vill ha ett högerorienterat vykoordinatsystem (vilket är naturligast) skall vektorn vara motsatt tittriktningen.

Enhetsvektorn $\hat{\boldsymbol{y}}_v$: Denna vektor skall vara vinkelrät mot $\hat{z}_v$. Man brukar ange en uppåtvektor **u**, som i bästa fall är just sådan, men i allmänhet inte är det. Då låter man enhetsvektorn vara den normaliserade vinkelräta komponenten, dvs $\boldsymbol{u} - (\boldsymbol{u} \bullet \hat{z}_v)\hat{z}_v$ normaliserad.

Enhetsvektorn $\hat{\boldsymbol{x}}_v$: Bildar vi som den vektoriella produkten $\hat{\boldsymbol{y}}_v \times \hat{z}_v$.

Ett par nyttiga erinringar:

1. Om $\mathbf{a} = (a_1,a_2,a_3)$ och $\mathbf{b}$ är icke-parallella vektorer och vektorn $\mathbf{a}$ har längden 1, dvs är normaliserad, så är vektorn
   c = b - (b•a)a
   vinkelrät mot vektorn **a** och ligger således i det plan som har **a** som normalvektor. Skalärprodukten **b•a** är $a_1b_1+a_2b_2+a_3b_3$.

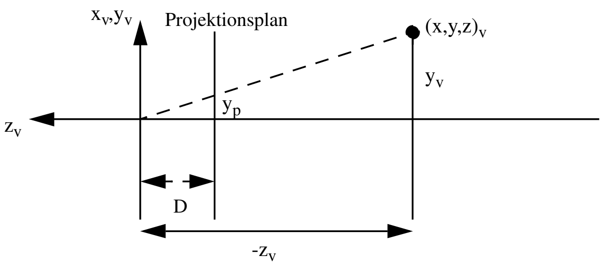
2. Om **a** och **b** är icke-parallella vektorer, så är vektorn (den vektoriella produkten av **a** och **b**)
c = **a** x **b**
vinkelrät mot både **a** och **b**. Vektorn **c** är riktad som när man vrider en högergängad skruv (kortaste vägen) från **a** till **b**. Den vektoriella produkten kan i ett högersystem beräknas som "determinanten" ($\mathbf{e}_i$) är enhetsvektorerna

$$\begin{vmatrix} e_1 & e_2 & e_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{bmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{bmatrix}$$

I ett vänstersystem får man sätta minustecken framför högerledet.

4 Projektion

Huvuduppgiften är att övergå från vykoordinatsystemet till en plan yta. Detta görs genom att man låter punkterna i vykoordinatsystemet projiceras på ett plan parallellt med x_vy_v-planet och på avståndet D framför origo (observatörens öga). Vid **parallellprojektion** (ortografisk projektion) görs en ren projektion på x_vy_v-planet (dvs $x_p=x_v$, $y_p=y_v$). Vid **perspektivprojektion** sker projektionen längs en linje mellan punkten och origo (ögat). I datorgrafiksammanhang vill man fortfarande oftast ha tillgång till full djupinformation och man låter i sådana fall projektionskoordinatsystemet ha en z-komponent. Observatörens position är origo i vykoordinatsystemet.



Figur 2: Perspektivprojektion.

Med hjälp av bilden ovan finner man lätt att:

$x_p = -x_vD/z_v$
$y_p = -y_vD/z_v$
$z_p = -D$ (Ingen bevarad djupinformation!)

Minustecknen kommer sig av att vi tittar i negativ synriktning. Om vi övergår till homogena koordinater, kan vi skriva övergången på matrisform. Sätt $w=-z_v$ (>0), varigenom (tredje raden är bara en omskrivning av $z_p=-D$ och fjärde raden är just $w=-z_v$)

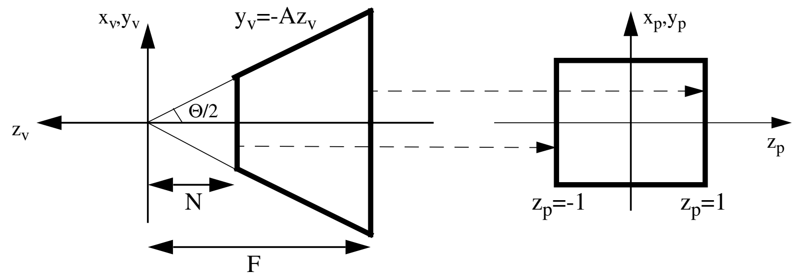
$$\begin{bmatrix} wx_p \\ wy_p \\ wz_p \\ w \end{bmatrix} = \begin{bmatrix} D & 0 & 0 & 0 \\ 0 & D & 0 & 0 \\ 0 & 0 & D & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

Observera att matrisen är oberoende av z_v, dvs samma matris kan användas för alla punkter. För att få fram de verkliga projektionskoordinaterna måste vi emellertid dividera med w, dvs den fjärde komponenten i den framräknade vektorn. Vårt mål nu är att hitta ett uttryck för z_p som bevarar djupinformationen och som dessutom gör att övergången mellan vykoordinater och projektionskoordinater kan skrivas på formen ovan (med en annan konstant matris). I så fall uppnår vi två viktiga saker:

1. Hela transformationen från modellkoordinater till projektionskoordinater kan beskrivas med en enda konstant matris.
2. Transformationen avbildar linjer på linjer och plan på plan. Detta är en konsekvens av punkt 1, vilket vi dock inte lägger ned möda på att visa. Detta är ett naturligt önskemål och är något som utnyttjas i många algoritmer. Däremot är det som vi kommer att visa senare inte självklart uppfyllt.

Ett naturligt val - som bevarar djupinformation - vore
$z_p = -z_v$ (PRELIMINÄR!)
men då går det inte att skriva projektionen på matris-vektor-form (med en konstant matris)!

I alla riktiga grafiksystem gör man en normalisering som innebär att man ser till att $-1 \le x_p,y_p \le 1$ och $-1 \le z_p \le 1$ (OpenGL) eller $0 \le z_p \le 1$ (DirectX). För detta krävs att man begränsar synfånget till en s k avhuggen synpyramid:



I OpenGL:s *gluPerspective* anger vi synvinkeln Θ i y-led och en synkvot (med vilken synvinkeln i x-led kan räknas ut) samt avstånden N (near)) och F (far) till de båda klipp-planen. Figurens A kan naturligtvis beräknas från Θ: A = tan (Θ/2). Vi antar för enkelhets skull att synkvoten är 1 så att synvinkeln och därmed A är samma i x-led. Den avhuggna synpyramiden kommer att avbildas på den önskade kuben.

Vi kan låta t ex det hitre klipp-planet utgöra projektionsplan, dvs D=N. För att få y_p i [-1,1] behöver vi bara dividera det tidigare värdet med halva höjden på det närmsta klipp-planet, dvs AN. Vi får då

$$y_p = -\frac{y_vD}{ANz_v} = -\frac{y_v}{Az_v}$$

och på samma sätt

$$x_p = -\frac{x_v}{Az_v}$$

För z skulle vi kunna göra den linjära transformationen

$$z_p = \frac{2z_v + N + F}{N - F}$$

som överför z_v=-N och z_v=-F i z_p=-1 resp z_p=1, men precis som för z_p=-z_v går det inte att skriva transformationen på matris-form.

I stället gör vi den olinjära transformationen (formen kan motiveras på olika sätt; ett pragmatiskt sätt är att vi vill ha division med z_v precis som för x_v och y_v)

$$z_p = 2\frac{1}{z_v}\frac{z_v + N}{1 - \frac{N}{F}} - 1 = \frac{z_v\left(1 + \frac{N}{F}\right) + 2N}{z_v\left(1 - \frac{N}{F}\right)}$$

Den första termen i mellanledet transformerar z=-N till 0 och z=-F till 2 och subtraktionen med 1 ser sedan till att intervallet i stället blir [-1,1].

Med w=-z_v som förut kan vi skriva

$$\begin{bmatrix} wx_p \\ wy_p \\ wz_p \\ w \end{bmatrix} = \begin{bmatrix} \frac{1}{A} & 0 & 0 & 0 \\ 0 & \frac{1}{A} & 0 & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & \frac{-2NF}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

dvs nu klarar vi oss med en konstant matris. Och precis som förut måste vi göra en division per punkt för att få fram de verkliga koordinaterna. Vi utgick ovan från att hitre klipp-planet var projektionsplanet. I själva verket har det dock ingen som helst betydelse för formeln ovan. I många läroböcker behandlar man allmännare synsituationer, vilket bara leder till en litet annorlunda matris.

En linje (ett plan) i vykoordinatsystemet blir inte säkert en linje (plan) i det preliminära (det med z_p=-z_v) projektionskoordinatsystemet. Däremot fortsätter den att vara en linje (plan) i det normaliserade systemet (vi visar det inte). Detta är en fördel eftersom vi vill göra en massa linjära saker i det normaliserade systemet, t ex interpolera. En annan fördel med det normaliserade systemet är att klippning sker mot plan som är parallella med koordinatplanen (i OpenGL sker klippningen före divisionen med w, dvs precis före normaliseringen, men formelmässigt blir det nästan samma sak).

Vi kan nu komma hela vägen från modellkoordinatsystemet till det normaliserade systemet med en konstant matris genom att bara på slutet dividera en gång per punkt. Den slutliga avbildningen till fönster/skärm-koordinater är trivialt linjär (se avsnitt 6).

Övning: Hur ser matrisen vid parallellprojektion med liknande normalisering ut?

5 OpenGL

Naturligtvis bör vi belysa teorin ovan genom att se på hur det verkligen är i OpenGL. Vi utnyttjar rutinen `void CheckMatrix(GLenum M)` (från avsnitt 8 i OpenGL-häftet) som tar en parameter som är antingen `GL_MODELVIEW_MATRIX` eller `GL_PROJECTION_MATRIX` och skriver ut motsvarande matris, dvs modell-vy-matrisen resp projektionsmatrisen, rad för rad.

Jag ser till att fönstret är kvadratiskt från början och låter det så förbli (dvs den omständligt skrivna synkvoten i proceduren nedan är 1, vilket vi ju för enkelhets skull förutsatte vid framtagningen av matriserna). Omskalningsproceduren ser ut så här:

```
void myReshape(int width, int height)
{
    ...
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(120.0,((GLfloat)width)/((GLfloat)height),1.0,9.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0,0.0,1.0, 0.0,0.0,-3.0, 0.0,1.0,0.0);
}
```

Och omritningsproceduren

```
void display(void)
{
    glClearColor(1.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    CheckMatrix(GL_MODELVIEW_MATRIX);
    CheckMatrix(GL_PROJECTION_MATRIX);
    RitaNgt();
    glPushMatrix();
       glRotatef(45.0,0.0,0.0,1.0);
       CheckMatrix(GL_MODELVIEW_MATRIX);
       CheckMatrix(GL_PROJECTION_MATRIX);
       RitaNgt();
    glPopMatrix();
    glFlush();
}
```

I omritningsproceduren skriver vi först ut modell-vy- och projektionsmatriserna. Därefter ritar vi något (godtyckligt vad, så ritproceduren finns inte med).

```
Utskrift av modell-vy-matris
 1.000000 0.000000 0.000000 0.000000
 0.000000 1.000000 0.000000 0.000000
 0.000000 0.000000 1.000000 -1.000000
 0.000000 0.000000 0.000000 1.000000
Utskrift av projektionsmatris
 0.577350 0.000000 0.000000 0.000000
 0.000000 0.577350 0.000000 0.000000
 0.000000 0.000000 -1.250000 -2.250000
 0.000000 0.000000 -1.000000 0.000000
```

Modellvy-matrisen skall först vara sådan att x- och y-värdena inte förändras medan $z_v = z_w - 1$ (vykoordinatsystemets origo ligger ju i $z_w = 1$). Detta ger den först utskrivna matrisen.

Låt oss även verifiera den utskrivna projektionsmatrisen. Vi har N=1, F=9 och A = tan 60^o. Som ger 1/A=0.5774 och -(F+N)/(F-N) = -10/8 = -1.25 och -2NF/(F-N) = -18/8 = -2.25. Praktik och teori stämmer alltså när det gäller projektionsmatrisen.

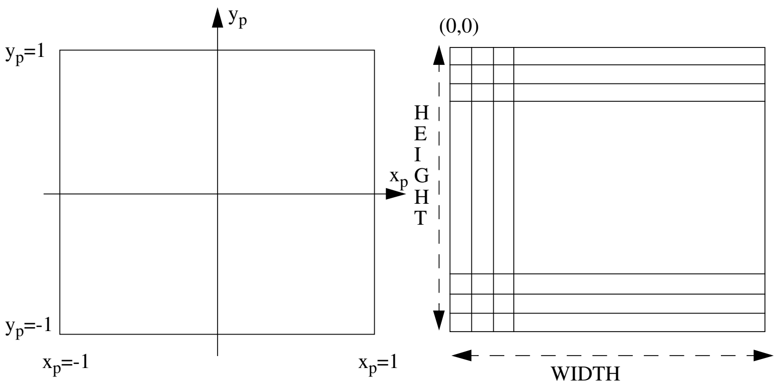
Sedan säger vi att allt som skall ritas skall roteras 45 grader runt z-axeln (i världskoordinatsystemet). Vi skriver återigen ut modell-vy- och projektionsmatriserna. Projektionsmatrisen blir identiska, eftersom perspektivet inte ändras, varför vi nu bara tar med den första utskriften.

```
Utskrift av modellvy-matris
 0.707107 -0.707107 0.000000 0.000000
 0.707107 0.707107 0.000000 0.000000
 0.000000 0.000000 1.000000 -1.000000
 0.000000 0.000000 0.000000 1.000000
```

Vi har ju gjort en rotation med 45 grader motsols kring z-axeln, vilket bara påverkar x och y. Därmed är även den andra modellvymatrisen som den skall vara enligt teorin.

6 Från reella 2D-koordinater till heltaliga

Vi har nu transformerat alla synliga punkter till att hamna i en enhetskub i projektionskoordinatsystemet. Återstår att transformera dessa värden (x_p,y_p,z_p) till heltaliga skärmkoordinater (fönsterkoordinater vore kanske numera ett bättre namn). Punkter med olika z_p men samma (x_p,y_p) hamnar i samma bildpunkt.



Och vi inser omedelbart att transformationerna

$$col = trunc(0.5WIDTH(1 + x_p))$$

$$row = HEIGHT - trunc(0.5HEIGHT(1 + y_p))$$

där funktionen *trunc* tänkes hugga av till närmsta lägre heltal, uträttar det vi vill. Varje bildpunkt motsvarar en liten kvadrat i (x_p,y_p)-planet. Se nästa avsnitt för kodning i C.

7 Ett eget 3D-system

Det är lätt att med utgångspunkt från våra transformationsmatriser göra ett litet 3D-system ovanpå ett befintligt 2D-system. Låt oss utgå ifrån att vi har ett kvadratiskt fönster med en given storlek

```
#define windowwidth 200
#define windowheight 200
```

där bildpunkterna numreras med ett heltalskoordinatsystem och att vi har en färdig rutin

`DrawLine(int x0,int y0,int x1,int y1)`

som drar ett streck från bildpunkten (x0,y0) till (x1,y1).

Låt datatypen *Point3D* beskriva en punkt i 3D, t ex

`typedef struct { double x, y, z; } Point3D;`

Vi behöver nu tillverka väsentligen 2 procedurer:

```
void DrawLine3D(Point3D p0, Point3D p1)
void Camera(Point3D eye, Point3D at, double A, double N, double F)
```

där A, N och F är som i avsnittet om projektion. Den första ritar ett streck i 3D. Den andra placerar kameran etc. Vi tänker oss för enkelhets skull att kamerans kvadratiska synfält skall avbildas på fönstret. Vi behöver också en global variabel (matris)

`double T[4][4]` som innehåller den matris som transformerar från världskoordinater till normaliserade projektionskoordinater.

För att rita ett koordinatsystem sett från (2,2,2) i världskoordinatsystemet behöver vi bara anropa

```
    Camera(makePoint3D(2.0,2.0,2.0), makePoint3D(0.0,0.0,0.0),
       makePoint3D(0.0,1.0,0.0), 1,1,10);
    DrawLine3D(makePoint3D(0,0,0),makePoint3D(1,0,0));
    DrawLine3D(makePoint3D(0,0,0),makePoint3D(0,1,0));
    DrawLine3D(makePoint3D(0,0,0),makePoint3D(0,0,1));
```

där *makePoint3D* är en funktion som returnerar ett värde av typen *Point3D* motsvarande de tre parametrarna. I det följande krånglar vi inte till det utan skriver "rakt-på-kod" utan att tänka på effektivitet eller generalitet. Proceduren *Camera* bygger upp matrisen *T*. De använda funktionerna *Normalize* och *VectorProduct* normaliserar en vektor respektive bildar en vektoriell produkt:

```
void Camera(Point3D eye, Point3D at, double A, double N, double F) {
    Point3D e1, e2, e3;
    double e3_dot_up;
    double V[4][4], M[4][4];
    // Beräkna enhetsvektorn för vykoordinatsystemets z-axel
    e3.x = eye.x - at.x; e3.y = eye.y - at.y; e3.z = eye.z - at.z;
    e3 = Normalize(e3);
    e3_dot_up = e3.x*up.x+e3.y*up.y+e3.z*up.z;
    // Beräkna enhetsvektorn för vykoordinatsystemets y-axel
    e2.x = up.x - e3_dot_up*e3.x;  e2.y = up.y - e3_dot_up*e3.y;
    e2.z = up.z - e3_dot_up*e3.z; e2 = Normalize(e2);
    // Beräkna enhetsvektorn för vykoordinatsystemets x-axel
    e1 = VectorProduct(e2,e3);
    // Nu kan vi omedelbart beräkna matrisen M (M-tilde) som beskriver
    // övergången från världskoordinater till vykoordinater och sedan
    // matrisen V som beskriver övergången från vykoordinater till
    // perspektivkoordinater. Slutligen beräknas T = VM.
    ...
}
```

Proceduren *DrawLine3D* blir som följer och utnyttjar de därpå två följande procedurerna.

```
void DrawLine3D(Point3D p0, Point3D p1) {
    Point3D p, q; int x0, y0, x1, y1;
    toProj(p0,&p); toProj(p1,&q);
    toScreen(p,&x0,&y0); toScreen(q,&x1,&y1);
    DrawLine(x0,y0,x1,y1);
}

void toScreen(Point3D p, int* x, int* y) {
    // -1<p.x,p.y<1 is the relevant portion
    *x = (int) (0.5*windowwidth*(p.x + 1)); // (int) hugger av
    *y = windowheight - (int) (0.5*windowheight*(p.y + 1));
}

void toProj(Point3D p, Point3D* q) {
    double w;
    q->x=T[0][0]*p.x+T[0][1]*p.y+T[0][2]*p.z+T[0][3];
    q->y=T[1][0]*p.x+T[1][1]*p.y+T[1][2]*p.z+T[1][3];
    q->z=T[2][0]*p.x+T[2][1]*p.y+T[2][2]*p.z+T[2][3];
    w = T[3][0]*p.x+T[3][1]*p.y+T[3][2]*p.z+T[3][3];
    q->x=q->x/w; q->y=q->y/w;  q->z=q->z/w;
}
```

Med den här enkla tekniken kan vi komma att rita sådant som inte skall synas. Det enda som skall synas är ju det som ryms inom den avhuggna synpyramiden, dvs kuben i normaliserade projektionskoordinater, dvs vi måste klippa bort allt som ligger närmre användaren än det hitre klipp-planet, dvs planet z_p=-1. Gör vi inte det kommer de delarna att ritas upp och ner. Klippning av en punkt är lätt, men betydligt mer komplicerad för t ex linjer. Vi måste naturligtvis också förhindra z_v=0 (annars blir det ju division med 0).

Skall vårt system även lösa synlighetsproblemet behöver vi en simulerad djupbuffert med åtminstone en punktritningsprocedur.

```
// Depth-buffer part
double dbuff[windowheight][windowwidth];

void ClearBuffer() {
     int i,j;
     for (i=0;i<windowheight;i++) {
        for (j=0;j<windowwidth;j++) dbuff[i][j]=-30.0;
     }
}

void DrawPoint3D(Point3D p0) {
     Point3D p;
     int x0, y0;
     toProj(p0,&p);
     toScreen(p,&x0,&y0);
     if (p.z>dbuff[x0][y0]) {
         DrawPoint(x0,y0);
         dbuff[x0][y0]=p.z;
     }
}
```

där *DrawPoint*, som ritar i fönsterkoordinater, tänkes finnas färdig

Och så här kan vi fortsätta i all oändlighet till dess vi nått det fulländade grafiksystemet! Grafiksystemet som det beskrivits här finns i en X -version `Xex3D.c` i `~graf/DEMOS`. Koden är tyvärr inte

strukturerad så att man lätt byter till t ex Windows. Figuren ovan innehållande tre koordinataxlar och fyra olikfärgade kubsidor är ritad med det.

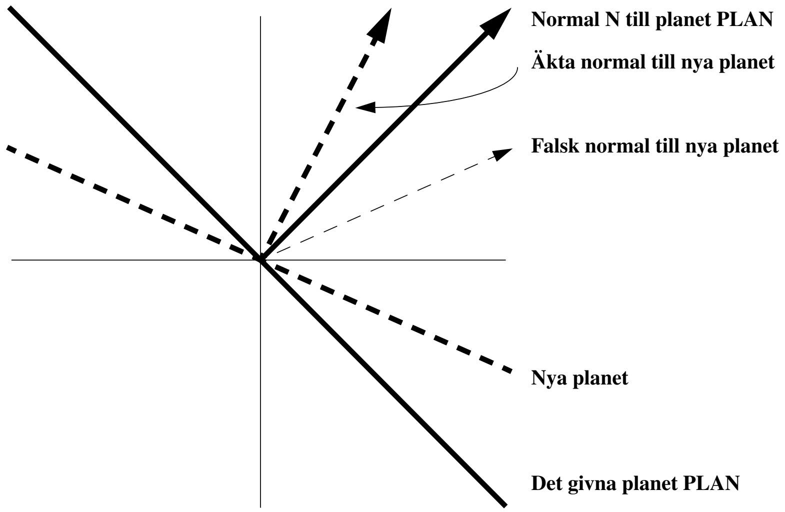
8 Transformation av normaler

Vi vet nu att man transformerar punkter från ett modell- eller världskoordinatsystem (koordinater (x_w,y_w,z_w)) till ett vykoordinatsystem och vidare till ett normaliserat projektionskoordinatsystem (koordinater (x_p,y_p,z_p) med $\mathbf{P}_v = T\mathbf{P}_w$ respektive $w\mathbf{P}_p = T\mathbf{P}_w$, där T är någon 4x4-matris, som kan ses som sammansatt av matriser för translation, rotation och skalning.

Hur är det då med normaler. Behöver vi bry oss? Egentligen inte, men om vi vill veta hur OpenGL bär sig åt är frågan befogad. Det finns dessutom en - enligt min mening - smart tillämpning av sådan kunskap i samband med synpyramidgallring ("rita bara sådana föremål som berör synpyramiden"), vilken jag tycker förtjänar ett eget avsnitt 9.

Betrakta som inledning planet PLAN i figuren nedan med tillhörande normal **N**. Låt oss göra en olikformig skalning, som består i att alla y-koordinater halveras medan x-koordinaterna bevaras. Då övergår planet i det streckade nya planet. Transformationen kan beskrivas med matrisen.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix}$$



Det ligger möjligen nära till hands att tro att normalen till det nya planet fås genom att låter samma transformation verka på **N**, men resultatet blir då i stället den vektor som i figuren kallas "falsk normal". Däremot ser vi att en äkta normal är T^{-1}**N** (i figuren är den något kortare), vilket visar sig vara nästan sant även allmänt.

Det allmänna resultatet är att om w**P**’ = T**P**, så är **N**’ = $(T^{-1})^*$**N**, där * betyder transponat (som ju innebär att matriselementet a_{ij} byter plats med a_{ji}). Här är alla matriser 4x4 och alla vektorer 4x1, vilket innebär att **N** har en fjärde komponent, vars värde strax anges.

Vi använder i fortsättningen det bekvämare skrivsättet (**a**,**b**) för skalärprodukten **a•b** mellan två vektorer **a** och **b** (3-dimensionella eller 4-dimensionella), dvs

$$(\boldsymbol{a},\boldsymbol{b}) = \sum_{i=1}^{n} a_i b_i$$

där n = 3 eller n = 4. Vi kommer att utnyttja att om A är en matris, så är (A**a**,**b**) = (**a**,A^***b**) och $(A^*)^{-1} = (A^{-1})^*$.

Låt oss först se på fallet att inga homogena koordinater behövs, dvs att inga translationer är inblandade. Då har vektorerna tre komponenter och matriserna är 3x3. Vi utgår från ett plan med känd normal **N**, som alltså är vinkelrät mot planet. Alla punkter **P** transformeras enligt T**P**. Om $\mathbf{P}_0$ och **P** är punkter på planet gäller alltså ($\mathbf{N},\mathbf{P}-\mathbf{P}_0$)=0. Detta är inget annat än planets ekvation. Jag hävdar nu att **N**’ = $(T^*)^{-1}$**N** = $(T^{-1})^*$**N** är en normal till det transformerade planet. Detta följer av att
$(\mathbf{N}', T(\mathbf{P}-\mathbf{P}_0)) = (T^*\mathbf{N}', \mathbf{P}-\mathbf{P}_0) = (T^*(T^*)^{-1}\mathbf{N}, \mathbf{P}-\mathbf{P}_0) = (\mathbf{N}, \mathbf{P}-\mathbf{P}_0) = 0$

Slutligen tittar vi på det allmänna fallet med homogena koordinater, som ju behövs för att klara hela transformationskedjan. Vi har

$$w\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = w\boldsymbol{P}' = T\boldsymbol{P} = T\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Normalen **N** skall nu ha fyra komponenter. Vi vill ha en motsvarighet till ($\mathbf{N},\mathbf{P}-\mathbf{P}_0$)=0 i det tidigare fallet. Om planets ekvation är Ax + By +Cz +D = 0, vet vi att $(A,B,C)^*$ är en normal och sätter

$$\boldsymbol{N} = \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

Då är (**N**,**P**) = 0 för punkter **P** på planet och med samma typ av resonemang som tidigare får man att det transformerade planet har parametrarna

$$\begin{bmatrix} A' \\ B' \\ C' \\ D' \end{bmatrix} = \boldsymbol{N}' = (T^*)^{-1}\boldsymbol{N} = (T^*)^{-1}\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix}$$

Detta kan alternativt uttryckas på formen

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = T^*\begin{bmatrix} A' \\ B' \\ C' \\ D' \end{bmatrix}$$

Vi har alltså direkta samband inte bara mellan normalerna utan även mellan planen. Omedvetet har vi faktiskt samtidigt visat att ett plan förblir ett plan under de transformationer som är aktuella.

Övning: Visa att om punkten **P** ligger på framsidan av planet definerat av **N** (samma som att (**N**,**P**)>0), så ligger T**P** på framsidan av det transformerade planet $(T^*)^{-1}$**N** och omvänt.

9 Synpyramidgallring

Det är ju onödigt att försöka rita sådant som inte kommer att synas. Annorlunda uttryckt är det onödigt att skicka objekt som ligger helt utanför synpyramiden till grafikprocessorn. Den stora frågan som inte kommer att besvaras här är vilket som går fortast: test eller ritande. Men låt oss i alla fall se hur detta kan gå till (jag påstår inte att sättet här är det praktiskt bästa).

Låt oss först tänka oss att objekten är kända i världskoordinatsystemet. Det är inte särskilt svårt att givet positionen för betraktaren, betraktningsrikning och synvinklar räkna ut ekvationer för de sex begränsande planen i synpyramiden (inklusive hitre och bortre klipp-plan).

Om objektet är en sfär med viss radie och med mittpunkten i en viss position, är det sedan bara att räkna ut avståndet mellan mittpunkten och ett av planen. Om punkten ligger på utsidan och avståndet är större än radien, ligger sfären i sin helhet utanför just detta plan. Processen upprepas för övriga plan. Avståndet mellan en punkt **P** = (x,y,z) och planet (A,B,C,D) ges av F(x,y,z) = Ax + By +Cz +D om $A^2 + B^2 + C^2 = 1$, dvs om den riktiga normalen är normaliserad.

Om objektet i stället är en allmän konvex polyeder, kan vi kontrollera hörn efter hörn genom att stoppa in i planens ekvationer. Eller också använder vi en omskriven sfär.

Men vi arbetar normalt inte med koordinater i världskoordinatsystemet när det gäller de grafiska objekten utan i modellkoordinatsystemet. Visst skulle vi kunna räkna ut världskoordinater, men det är ju ett arbete som vi gärna överlåter på grafikprocessorn. I stället kan vi göra de olika testen i modellkoordinatsystemet om vi uttrycker synpyramidens plan i det. Verkar lika illa eftersom i så fall den inverterade modelltransformationen förefaller att behöva användas. Och då kan vi ju lika väl ta steget fullt ut och utgå från synpyramiden i projektionskoordinatsystemet. En strålande idé. I projektionskoordinatsystemet känner vi ju de begränsande planen (jfr fig sid 6). Transformationsmatrisen T=**Proj*****M** (från modellkoordinater till projektionskoordinater) kan vi beräkna genom att läsa av dels projektionsmatrisen **Proj**, dels modell-vy-matrisen **M** på det sätt som beskrivs i avsnitt 8 i OpenGL-häftet (användes även i avsnitt 5 här). Därefter kan vi beräkna planen i modellkoordinatsystemet eller världskoordinater (om vi inte har någon modelltransformation) med

$$N = T^* N' = \begin{bmatrix} t_{11} & t_{21} & t_{31} & t_{41} \\ t_{12} & t_{22} & t_{32} & t_{42} \\ t_{13} & t_{23} & t_{33} & t_{43} \\ t_{14} & t_{24} & t_{34} & t_{44} \end{bmatrix} N'$$

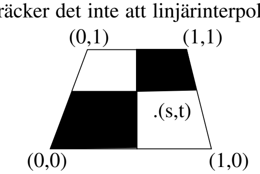
Det hitre klipp-planet har ekvationen -z - 1 = 0 (med utåtriktad normal), dvs (A,B,C,D) = (0,0,-1,-1). Detta tillsammans med sambandet mellan **N** och **N'** ger kolumnen för Hitre i tabellen nedan. Övriga kolumner fås på motsvarande sätt.

	Hitre	Bortre	Undre	Övre	Vänster	Höger
N' i projektionskoordinater	$\begin{bmatrix} 0 \\ 0 \\ -1 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ -1 \\ 0 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}$
N i modellkoordinater	$-\begin{bmatrix} t_{31}+t_{41} \\ t_{32}+t_{42} \\ t_{33}+t_{43} \\ t_{34}+t_{44} \end{bmatrix}$	$\begin{bmatrix} t_{31}-t_{41} \\ t_{32}-t_{42} \\ t_{33}-t_{43} \\ t_{34}-t_{44} \end{bmatrix}$		$\begin{bmatrix} t_{21}-t_{41} \\ t_{22}-t_{42} \\ t_{23}-t_{43} \\ t_{24}-t_{44} \end{bmatrix}$		

10 Rastrering

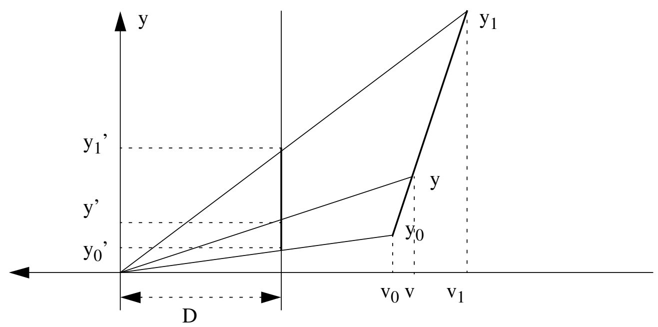
Detta avsnitt blir aktuellt först när vi tittar på texturer.

Det som är en linje i vår värld fortsätter att vara en linje på vår skärm (eller i skärm+djup). Det innebär att vi 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 linjärinterpolera texturkoordinater om vi vill ha



perspektivistiskt korrekt texturering (bilden är korrekt), eftersom objekt skall "krympa" när avståndet till betraktaren ökar. Vi skall här reda ut hur vi (eller grafikkretsen) måste gå tillväga.

Vi har en linje med ändpunkterna $\mathbf{P}_0 = (x_0,y_0,v_0)^*$ och $\mathbf{P}_1 = (x_1,y_1,v_1)^*$ i vykoordinatsystemet men med den tredje axeln i synriktningen (härigenom slipper vi ett tankesteg), dvs $v_i = -z_i > 0$.



För en godtycklig punkt (x,y,v) på linjen finns det ett tal $0 \le \alpha \le 1$, sådant att

(1) $y = y_0 + \alpha(y_1 - y_0)$, $x = x_0 + \alpha(x_1 - x_0)$, $v = v_0 + \alpha(v_1 - v_0)$.

Detta är ju helt enkelt en parameterframställning av linjen.

På motsvarande sätt gäller för motsvarande punkt (x',y',D) på den perspektivprojicerade linjen att det finns ett tal $0 \le \beta \le 1$, sådant att

(2) $y' = y'_0 + \beta(y'_1 - y'_0)$, $x' = x'_0 + \beta(x'_1 - x'_0)$.

Det är uppenbart att om $\alpha=0$ så är $\beta=0$ och vice versa. Likaledes är $\alpha=1$ om och endast om $\beta=1$. Om transformation hade varit linjär hade det genomgående gällt att $\alpha=\beta$. (**P** punkt på linjen med ändpunkter $\mathbf{P}_0$ och $\mathbf{P}_1$, dvs $\mathbf{P}=\mathbf{P}_0+\alpha(\mathbf{P}_1-\mathbf{P}_0)$, $\mathbf{P}' = M\mathbf{P}$ ger $\mathbf{P}'=M\mathbf{P}_0+\alpha(M\mathbf{P}_1-M\mathbf{P}_0)=\mathbf{P}'_0+\alpha(\mathbf{P}'_1-\mathbf{P}'_0)$). Men nu är den inte det. Vi söker ett samband mellan α och β.

Bilden är litet missledande eftersom det kan finnas en ytterligare transformationer (normalisering och övergång till skärmkoordinater), men dessa är ju linjära (i x och y; vi skalar ju bara x och y).

Ekvationen (2) säger att

$$\beta = \frac{y' - y'_0}{y'_1 - y'_0}$$

Eftersom (från figuren; likformiga trianglar)

$$\frac{y'}{D} = \frac{y}{v}$$

får vi

$$\beta = \frac{\frac{y}{v} - \frac{y_0}{v_0}}{\frac{y_1}{v_1} - \frac{y_0}{v_0}}$$

Om vi här stoppar in uttrycket för y från (1) och räknar på litet, finner man att

$$\beta = \alpha\frac{v_1}{v}$$

vilket väl också borde kunna tas fram med likformighet. Stoppar vi sedan in uttrycket för v från (1) blir det

$$\beta = \frac{\alpha v_1}{v_0 + \alpha(v_1 - v_0)}$$

eller om vi så vill

$$\alpha = \frac{\beta v_0}{v_1 + \beta(v_0 - v_1)}$$

Vi kan slutligen skriva

$$(3) \qquad y = y_0 + \alpha(y_1 - y_0) = y_0 + \frac{\beta v_0}{v_1 + \beta(v_0 - v_1)}(y_1 - y_0) = \frac{\frac{y_0}{v_0} + \beta\left(\frac{y_1}{v_1} - \frac{y_0}{v_0}\right)}{\frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)}$$

Vi kan här ersätta y med vilken annan storhet som helst som varierar linjärt längs linjen. Speciellt med v i stället för y fås

$$v = \frac{1}{\frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)}$$

eller

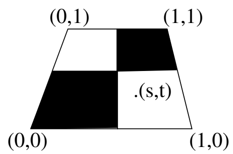
$$\frac{1}{v} = \frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)$$

Formeln (3) kan formuleras som

$$y = \frac{y/v}{1/v}$$

där täljare och nämnare var för sig kan framräknas med linjär interpolation med β mellan värdena vid ändpunkterna.

Vi får ta till sådana här formler t ex när vi vill ha perspektivistiskt korrekt texturering. Tag t ex följande situation med en kvadrat



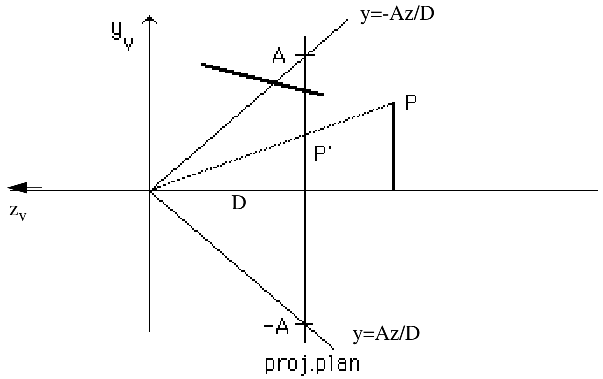
Texturkoordinaterna är kända i hörnen. Men vi kan inte beräkna dem i en godtycklig punkt på skärmen med vanlig linjär interpolation. Däremot med

$$(4) \qquad s = \frac{\frac{s_0}{v_0} + \beta\left(\frac{s_1}{v_1} - \frac{s_0}{v_0}\right)}{\frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)} \qquad t = \frac{\frac{t_0}{v_0} + \beta\left(\frac{t_1}{v_1} - \frac{t_0}{v_0}\right)}{\frac{1}{v_0} + \beta\left(\frac{1}{v_1} - \frac{1}{v_0}\right)}$$

som innebär en flyttalsdivision per ny bildpunkt och texturkoordinat. Flyttalsdivisioner har hittills alltid varit extremt dyra i förhållande till andra operationer. Ett trick i den tidiga datorspelsindustrin där man framför allt lade texturer på vertikala väggar och horisontella tak och golv, var att rita väggarna i form av successiva vertikala streck, längs vilka ju v är konstant. Härigenom räcker det med en division per streck eftersom övriga kan ersättas av multiplikationer.

11 Om klippning i 3D

Vi skall här beröra några principiella frågor kring klippning, dvs förhindrande av att sådant som inte skall synas ritas. Vi går inte för närvarande in på några effektivitetsresonemang och inte heller på algoritmer. I avancerade grafiska system sköts klippningen liksom mycket av en grafisk processor.



Vi gör först klart för oss vad som skall synas och hur ett par parametrar påverkar vår bild. Figuren ovan visar vykoordinatsystemet med origo där observatören (ögat) befinner sig. På avståndet D från ögat finns ett projektionsplan. Föremålen projiceras på detta plan, t ex övergår punkten **P** i **P'**. Våra tredimensionella objekt förs på detta vis över till tvådimensionella avbildningar. En del av projektionsplanet avbildas sedan på ett aktuellt rifönster hos en datskärm. Hur stor del av projektionsplanet som tas med bestäms enligt tidigare diskussion av vad vi sätter synvinkeln till.

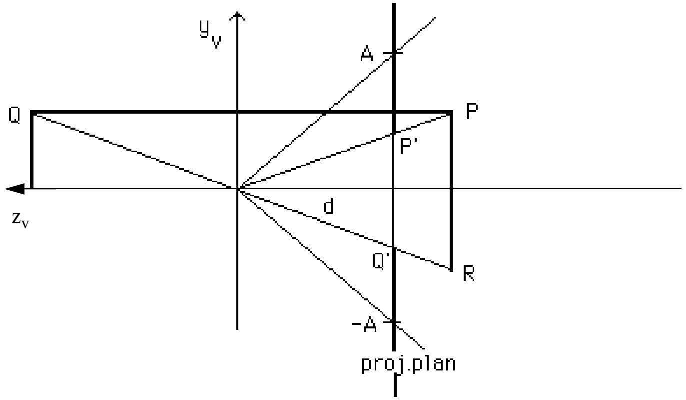
Det är uppenbart att ju större A är, desto mindre ter sig ett föremål av en given storlek. Vidare blir intrycket av ett föremål mindre om det avlägsnas från observatören eller om projektionsplanet närmar sig observatören.

Allt som finns utanför en pyramid (som i figuren visas som en triangel) begränsad av bl a de sneda planen y=±Az/D, x=±Az/D med z≥0 är osynligt eftersom sådana punkter projiceras utanför den valda delen av projektionsplanet. T ex är hela det högra tjocka strecket synligt, medan bara den undre delen av det vänstra är det.

Frågan är då vilka åtgärder som behöver vidtas. Vi skall först se att klippning av objekt som finns bakom observatören måste ske. Vi erinrar oss formlerna för perspektivprojektionen, dvs övergången från vykoordinatsystem till perspektivkoordinatsystem.

$x_p = x_vD/z_v$, $y_p = y_vD/z_v$, z_p = preliminära eller definitiva

vilka vi tagit fram med det underförstådda antagandet att z_v är större än 0, vilket naturligtvis inte gäller för punkter bakom observatören. Geometriskt betyder dessa formler det som åskådliggörs i nästa figur. En punkt **P** framför ögat projiceras på projektionsplanets **P**'. En motsvarande punkt **Q** bakom ögat men med samma (x_v,y_v) hamnar på projektionsplanets **Q**' och kan därför efter projektionen inte skiljas från punkten **R**:s projektion. Punkter bakom ögat måste således klippas före perspektivprojektionen. Notera även att linjen PQ vid projektion utan klippning skulle brytas ned i två segment, vara det ena sträcker sig från **P**' till ∞ och det andra från **Q**' till ∞.



Men punkter mellan xy-planet och pyramiden behöver de klippas? Matematiskt sett behövs det inte. De hamnar ju utanför gränserna i användarkoordinatsystemet och ritas därför inte heller om 2D-ritningen fungerar som den skall. Men inte alla grafiksystem sköter sitt jobb korrekt. T ex arbetar X med 16-bitars heltal när det gäller heltalskoordinater och därför uppfattas t ex x=66000 (65536+464) och x=464 som samma tal. Detta är lätt att kontrollera med t ex

```
XDrawLine(Xdpy,window,Xgc,100,10,200,10);
XDrawLine(Xdpy,window,Xgc,65536+100,40,65536+200,40);
```

som ger två horisontella linjer under varandra, trots att den ena ligger helt utanför fönstret. Ytterligare ett skäl är effektivitetsmässigt: man vill bli av med punkter som inte skall visas så tidigt som möjligt i transformationskedjan så att man inte behöver räkna så mycket på dem.

Nu lägger man oftast inte klippningsplanet vid z_v=0, utan någonstans mellan det planet och projektionsplanet. Härigenom undviker vi de problem som små z_v-värden skulle ställa till med vid tillämpandet av formlerna ovan (egentligen är det bara en omgivning till origo som är "farlig").

Men klippning med D>0 introducerar ett nytt problem, som illustreras i nästa figur. Nu kan plötsligt delar av föremål framför observatören bli synliga, trots att de egentligen döljs av av andra föremål. När vi i figuren klipper föremål F1 mot klippningsplanet så blir övre delen av föremål F2 synlig. Problemet blir naturligtvis mera accentuerat ju längre bort klippningsplanet placeras. Det är något

vi får lov att leva med. Det finns fall där man låter klippningsplanet sammanfalla med projektionsplanet. Detta kan försvaras med att man ser paret observatör/projektionsplanet som en observatör utrustad med en kamera (tubkikare är väl kanske en bättre analogi).