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.

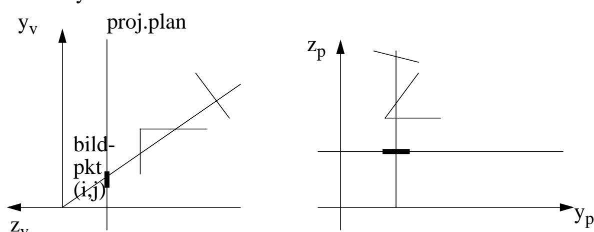
Förutsättningar: Vi tänker oss innehållet i världen uppbyggt av polygoner (kan vara enbart trianglar eller allmänna).

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 $0 \leq djup \leq 1$). 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 ....)`

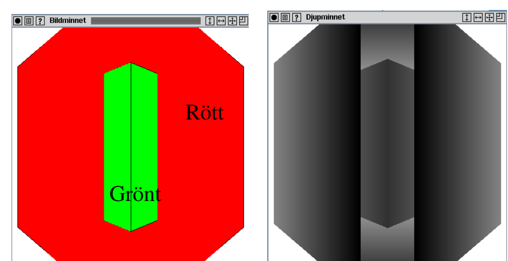
2. Slå på djupminnestestet med (skall göras efter det att man med `glutCreateWindow` skapat fönstret)
`glEnable(GL_DEPTH_TEST)`. Kan tillfälligt slås av med `glDisable(GL_DEPTH_TEST)`.

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

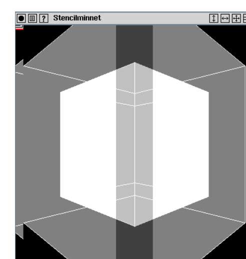
```
glClearColor(raderingsfärg);
glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);
```


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

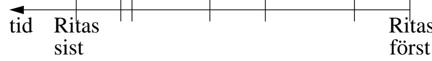


har gått så nära att man ser att det finns en grön kub inuti. Det är klippningen vid hitre klipp-planet som gör att den yttre kuben “öppnar” sig. Till höger har vi med gråskala avbildat djupminnet. Punkter längst bort har ju djupet 1.0, dvs de blir vita. Punkter närmare oss har mindre djup och blir följaktligen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



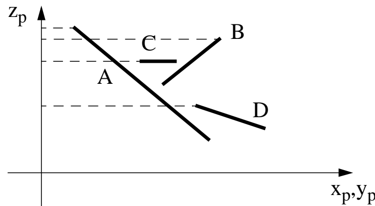
Dolda ytor: Målarns metod 1(2)

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.

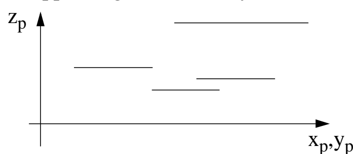


Vi inser att om vi låter listan vara ADBC blir uppritningen korrekt.

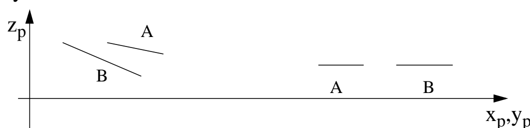
Ett första försök: Sortera efter växande $z_{p,max}$ (dvs hörnens största z_p-värde). Skulle i vårt exempel ge listan DCBA och uppritningsresultatet blir fel. En annan variant är att sortera efter $z_{p,medel}$. Men inte heller det sättet ger i allmänhet rätt resultat. Dessa enkla metoder brukar dock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

Dolda ytor: Målarns metod 2(2)

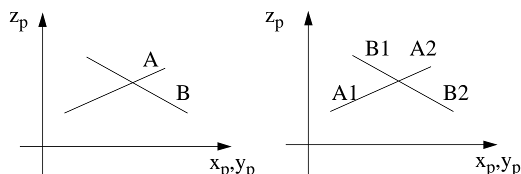
polygoner) och för s k 2 1/2-D-scener (objekten har inget djup och är vinkelräta mot betraktningsriktningen). MATLAB använde länge denna metod för uppritning av funktionsytor.



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:



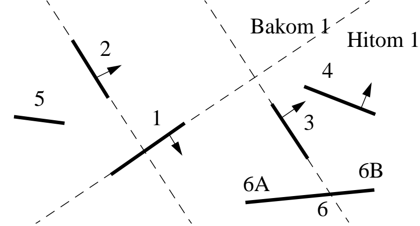
Då får man ta till uppdelning (klipp vardera planet mot det andra).

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

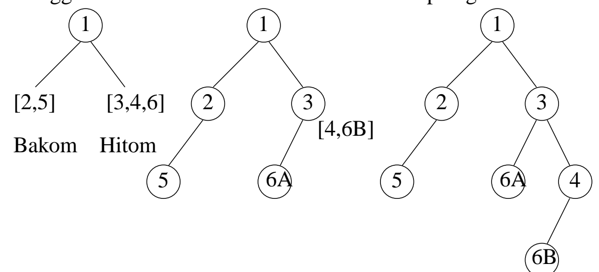
Låt oss först illustrera steg 1 med en artificiell exempelsituation,



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.
}
```

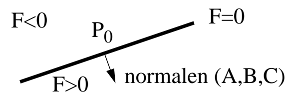
Uppritningsteget (kolla även om trädet tomt! Fick ej plats i koden):

```
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 $F(x,y,z)>0$.

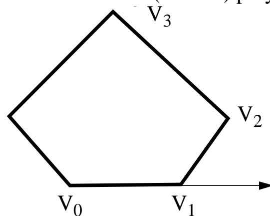


Vi inser att det förhåller sig som i figuren, ty om (x,y,z) ligger på den utritade normalens sida är skalärprodukten av $(x,y,z)-P_0$ och (A,B,C) större än noll.

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.

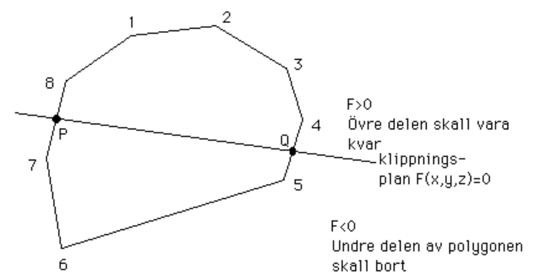
BSP i sin grundform klarar dem (bortsett från att vi inte ännu vet hur man skall blanda den genomskinliga och bakomliggande ytans färger).

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 principellt 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.

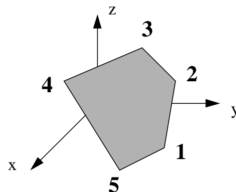
Men det betyder att vi måste lägga tester tidigare. Har vi en långsam dator och ett snabbt grafikkort uppnår vi troligen inte någon förbättring. Men annars. Och resultatet blir naturligtvis bäst om vi kan testa många polygoner i ett svep, dvs med begränsningsobjekt.

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

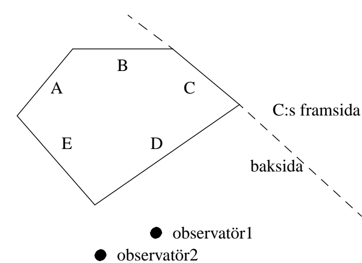
- Belysning (eng. lighting)
- Gallring (eng. culling)

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();
```

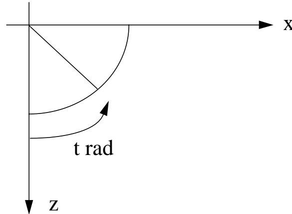
där *dx* är steglängden. Vi ändrar alltså betraktningspunkten samtidigt med positionen, vilket förefaller naturligt. Motsvarande för y- och z-komponenterna.Valet av tangenter bör ske med omsorg och med hänsyn till användaren. Piltangenterna räcker inte riktigt till för våra sex möjliga rörelser.

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

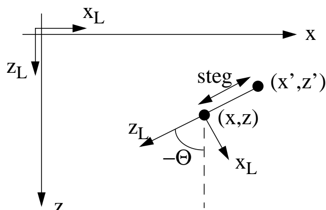
```
if (key == 't') t = t - dt;
if (key == 'T') t = t + dt;
pos_z = R*cos(2*M_PI*t); pos_x = R*sin(2*M_PI*t);
glutPostRedisplay();
```

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
$M = L_n ... L_2 L_1 V$,
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
$Proj L_n ... L_2 L_1 Mod$

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)