Introduktion till OpenGL

Magnus Bondesson
Institutionen för Datavetenskap
98-01-15, 99-01-12, 00-01-12, 01-01-07, 02-01-17, 03-01-14, 04-08-18, 05-08-23, 06-03-17

1 Inledning

OpenGL (uttytt Open Graphics Library) är namnet på ett grafikbibliotek som ursprungligen (1992) utvecklades av datorföretaget Silicon Graphics (SGI) för deras arbetsstationer. Numera finns det tillgängligt i olika former för många datormiljöer, bl a ingår det i Windows 95/NT/98/2000/XP och Mac OS 9. Med OpenGL kan man skapa såväl enkel grafik som mycket avancerad. Det kan användas från bl a programspråken C, C++, Java, C#, Ada, Haskell och Python. Tack vare bl a SGI:s välvilja kan man från år 2000 även fritt använda OpenGL under olika Linuxsystem.

OpenGL är ett renodlat grafikbibliotek utan någon som helst koppling till ett fönstersystem. Detta låg det länge i fatet. Man gjorde ett par olika försök att skapa ett systemoberoende gränssnitt mot fönstersystemet (bl a AUX och TK). Sedan några år finns **GLUT** (OpenGL Utility Toolkit), som funnit bred acceptans och fritt kan installeras. Man kan med det lätt öppna fönster och till med skapa enklare menyer. Finns även i en utvidgad/bantad version kallad *freeglut*, som fult nog använder samma biblioteksnamn. Ett annat liknande system är **SDL** (Simple DirectMedia Layer), som dessutom bjuder på kommunikations- och ljudhjälp. Men fortfarande envisas en del med att göra även enklare OpenGL-program plattformsberoende.

Vill man ha ett snyggare användarsnitt än vad GLUT erbjuder kan man ta till påbyggnaden **MUI** (Micro User Interface), som enbart bygger vidare på OpenGL och GLUT, och därför kan användas överallt där dessa fungerar. Det finns flera andra möjligheter, t ex **FLTK**.

En kommersiell (numera är källkoden släppt) C++-produkt kallad *Open Inventor* är tillgänglig på många plattformar och ger grafik på högre nivå än OpenGL, som dock ligger i botten (se avsnitt 31). För sina egna datorer och PC-datorer tillhandahåller SGI andra påbyggnadsprodukter som *Optimizer* och *Performer* som med olika tekniker (bl a s k scengrafer) effektiviserar arbetet med stora modeller. Även dessa är avsedda för C++.

OpenGL är konstruerat för att kunna utnyttja grafikarkitekturen i moderna system. Det finns åtskilliga grafikkort för PC-datorer med goda prestanda.

När jag började skriva detta i slutet av november 1997 föreföll det som om OpenGL stred med Microsofts eget DirectX (som innehåller Direct 3D men också annat) om herraväldet. Direct3D har "lånat" en hel del från OpenGL. Men eftersom DirectX bara finns för PC-miljö är valet i den situation vi befinner oss enkelt. I början av december 1997 förklarade SGI och Microsoft att de skulle samarbeta med målet att OpenGL skulle användas för seriösa tillämpningar och DirectX för spel. Efter ytterligare någon vecka deklarerade de båda företagen i stället att de skulle samarbeta om en ny produkt med namnet Fahrenheit. Men det samarbetet varade inte länge.

Varje datortillverkare med självaktning har haft egna grafikbibliotek med en funktionalitet liknande den i OpenGL. Förklaringen har varit att man velat göra finesser i sina grafikkort tillgängliga bekvämt och utan att avslöja några hemligheter. Exempelvis har Sun haft sitt XGL (X Graphics Library). Tiden för den typen av lösningar är ute. Standard skall det ju vara.

Framställningen här skall inte ses som en manual. Jag ger sällan alla detaljer och utesluter avsiktligt sådant som jag bedömer som mindre intressant och kanske är jag någon gång föga varsam med sanningen. För den fulla sanningen hänvisas du till annan litteratur, se slutet av häftet. Programkoden är i regel testad i bara en miljö. I exemplen används genomgående C. I ett avsnitt visas dock hur man kan koda i Java. Ordningen bland avsnitten är ibland något godtycklig.

Det sägs ofta att OpenGL är en standard, men som så ofta när det gäller sådana finns utrymme för implementeringar med varierande kapacitet. Se t ex avsnittet om buffertar längre fram. En standard utvecklas också mer eller mindre snabbt. OpenGL 1.0 gällde fram till 1996. Successivt kom sedan versionerna 1.1 till 1.5 med 1-2 års mellanrum. OpenGL 2.0 släpptes hösten 2004 och stöds av de flesta nyare grafikkort. Utvecklingen av OpenGL övervakas av en från SGI fristående grupp "OpenGL Architecture Review Board" (ARB) med representanter från diverse datorföretag.

Källkoden (dock ibland modifierad) till samtliga exempel finns tillgängliga i `$DG/EXEMPEL_MB`. Systemvariabeln *DG* ges värde med kommandot `setup_course TDA360`. I `$DG/EXEMPEL_GLUT` (länk) finns källkod och körbara program från GLUT-distributionen.

I förhållande till föregående version av denna text har ett antal korrigeringar och förtydliganden gjorts. Vid förra omarbetningen skedde viss Linux-anpassning och Ada-avsnittet 4 har ströks och Java-avsnittet 33 fick nytt innehåll. Säkert finns ett eller annat gammalt eller nytt fel, som jag är tacksam om upptäckaren påpekar. Läsaren förutsättes känna till transformationer och matriser.

2 Konventioner

Vi beskriver ibland rutiner (dvs procedurer/funktioner) på sedvanligt sätt, t ex
`void glTranslatef(GLfloat x, GLfloat y, GLfloat z)`
men ofta i stället med informella anrop. I de senare fallen används stora bokstäver genomgående för fördefinierade konstanter (motsv).

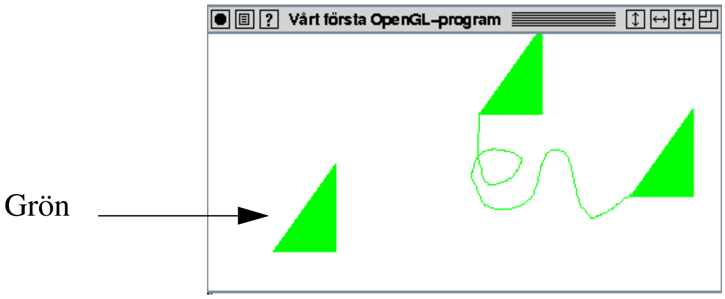
OpenGL har egna typnamn inledda med GL (t ex *GLfloat*), som i allmänhet motsvarar samma namn i den aktuella C-versionen (det är ju inte standardiserat i C hur dessa skall tolkas). Sist i rutinnamnet står ofta en bokstav (d=double, f=float, i=int), som anger vilken typ av parametrar som väntas. T ex finns *glTranslatef* även i form av *glTranslated* och *glTranslatei*. I litteraturen skriver man ofta glTranslate* för att täcka alla möjligheter. I praktiken betyder det på grund av C:s konverteringsregler inte så mycket, vilken version man använder. Exempelvis kan en rutin som har GLdouble/double-parametrar matas med storheter av typen GLdouble/double/GLfloat/float/GLint/int. Beroende på aktuell implementering kan effektiviteten variera mellan de olika versionerna. I mina exempel är jag litet vacklande. **Enklast för dig är att genomgående använda d-versionen om det finns flera**. Vissa rutiner finns i olika former med 2-4 koordinater. Man skjuter då in en sifferuppgift före parametertypsbokstaven. Ett typiskt exempel är glVertex-familjen för punkter *glVertex2f*, *glVertex3f* och *glVertex4f*. I anropen anger man (x,y)-koordinater, (x,y,z)-koordinater respektive (x,y,z,w)-koordinater. Ett antal rutiner arbetar med vektorer. Då får namnet sluta med ett v, t ex *glVertex3fv*, som tar en vektor med (x,y,z) som parameter.

3 Ett par små exempel

Vi kommer inte att separera OpenGL från GLUT i vår presentation.

OpenGL är ett bibliotek för grafik i 3D med bl a automatisk lösning av dolda-yt-problemet och med texturer för ytor. Men det kan också användas för enkel 2D-grafik.

Exempel 1: Låt oss börja med ett mycket enkelt exempel, där programmet bara ritar en en fylld polygon, den undre vänstra triangeln i figuren) och sedan väntar på att bli avslutat med våld.



Programmet ser ut så här:

```
/**************************************************************
*  GL_ENKEL1.c
*  Ritar en fylld triangel. Fönstret uppdateras korrekt.
**************************************************************/
#include <stdlib.h>
#include <stdio.h>                          // Används ej här. In/ut.
#include <math.h>                           // Dito. Bl a M_PI (π) och sin.
#include <GL/glut.h>                        // Inkluderar i sin tur gl.h, glu.h
void myFigure(int x, int y){
    glBegin(GL_POLYGON);                    // De tre hörnen i en triangel
         glVertex2f(x,y);                   // Fungerar lika bra med 2d och 2i
         glVertex2f(x+50,y);
         glVertex2f(x+50,y+70);
    glEnd();
    glFlush();
}

void myUpdate(void){                         // Anropas vid omritningsbehov
    printf("myUpdate called!\n");
    glClearColor(1.0,1.0,1.0,0.0);           // Raderingsfärg
    glClear(GL_COLOR_BUFFER_BIT);            // Raderar färgbuffert
    glColor3f(0.0,1.0,0.0);                  // Ritfärg
    myFigure(50,30);
}

void myReshape(int width, int height){       // Anropas vid fönsterändring
    printf("myReshape called!\n");
    glViewport(0, 0, width, height);         // Hela fönstret utnyttjas
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    glOrtho (0.0, width, 0.0, height, -1.0, 1.0); // Hur mycket skall synas
                                                  // av världen
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}
```

```
int main(int argc, char** argv) {
    glutInit(&argc,argv);  -- Kan uteslutas om programmet inte tar emot par
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutInitWindowSize(400,200);
    glutCreateWindow("Vårt första OpenGL-program");
    glutReshapeFunc (myReshape);
    glutDisplayFunc(myUpdate);
    glutMainLoop();
    return 0;                            //Krav i ANSI-C, som vi ofta ignorerar
}
```

Det ser säkert litet konstigt ut för den oinvigde. Vi skall inte diskutera alla enskildheter. Låt oss emellertid börja i *main*. Med anropen av *glutInitWindowSize* och *glutCreateWindow* sätter vi fönsterstorlek och fönsterrubrik och begär att ett fönster skall skapas. GLUT-program är **händelsestyrda**. Anropet *glutMainLoop* innebär att vi går in en s k händelsesnurra, som är en evig snurra med

```
Om det finns någon händelse i kö
     Utför åtgärderna som hör till den typen av händelse
Annars
     Utför åtgärderna i en procedur införd med glutIdleFunc(procedurnamnet).
     Vi har ingen sådan i vårt exempel. Används typiskt för att få
     självgående rörelse.Se avsnitt 17 (och 21).
```

En händelse kan vara att en mustangent trycks ned eller släpps eller att musen rör sig. Andra exempel är tryck på en vanlig tangent och uppdateringshändelser, som uppkommer när något behöver ritas om. När händelserna inträffar tas de omedelbart om hand av operativsystemet (fönstersystemet) och placeras som en **händelsepost** i en kö. Händelseposten innehåller diverse information om händelsen, t ex när den inträffade och var musen då befann sig. Kön betas sedan successivt av i händelsesnurran.

Med anropet `glutDisplayFunc(myUpdate)` anger vi att *myUpdate* skall anropas i händelsesnurran när en uppdateringshändelse (behov av omritning) påträffas. På motsvarande sätt gör `glutReshapeFunc(myReshape)` att *myReshape* anropas vid storleksändring av fönstret. Man kan kan också införa en funktion som anropas när datorn inte har något annat att göra. Kopplingen görs med anropet `glutIdleFunc(idle)`, där *idle* är namnet på önskad funktion. Den kan inte ha några parametrar.

När vi ändrar storlek på fönstret (inklusive när det skapas) anropas först den funktion som vi angivit som parameter i anropet av *glutReshapeFunc* och sedan den funktion *myUpdate* vi angivit med *glutDisplayFunc*. Den senare funktionen anropas ensam om ett fönster behöver uppdateras (t ex om en del blir synlig efter att varit osynlig). Det senare är i en del X-system onödigt, men GLUT kräver att det alltid finns en Display-funktion.

Proceduren *myUpdate* skriver ut att den anropats (s k pedagogisk finess), raderar det gamla innehållet i fönstret (raderingsfärg vitt) och ritar genom anropet `myFigure(50,30)` en fylld triangel. All figurritning bygger på principen

```
glBegin(figurtyp);
     Uppräkning av hörn (motsv)
glEnd();
```

Anropet

```
glFlush();
```

ser till att grafikkommando skickas till grafik-servern (detta görs automatiskt med viss regelbundenhet, men resultatet blir ofta bättre om vi styr själva). Precis som X arbetar OpenGL med en klient-server-modell.

Ritfärgen sattes precis före anropet av *myFigure* till grönt med

```
glColor3f(0.0,1.0,0.0);
```

Färger anges i RGB-systemet med tal mellan 0 och 1, varvid 1 betyder full intensitet. När vi med `glClearColor(1.0,1.0,1.0,0.0)` angav raderingsfärgen till vitt, står det fjärde talet för den s k alpha-komponenten, vars värde vi tills vidare kan bortse ifrån.

Proceduren *myReshape* skriver ut att den anropats och ser till att koordinatsystemet anpassas till fönstret på ett sådant sätt att figurer behåller sin storlek. Vi sparar detaljerna till senare.

Vi kan med ett program av denna typ rita statiska figurer av godtycklig komplexitet. Det är bara att ändra i *myFigure*. ❋

Exempel 2: Låt oss nu ändra kraven på programmet något så att vi får ett inslag av interaktion. Vi vill att vid mustryck skall vår triangel ritas på musmarkörens plats och om markören flyttas skall det dras en kurva som följer rörelsen. Se högra delen av figuren i exempel 1. Detta är lätt ordnat. Vi lägger till

```
glutMouseFunc(myMouse);
glutMotionFunc(myMoveMouse);
```

före inträdet i händelsesnurran, som gör att *myMouse* anropas vid mustryck och mussläpp samt att *myMoveMouse* anropas vid varje förflyttning (med nedtryckt mus). Sedan skriver vi motsvarande procedurer. Det visar sig att vi också behöver tre globala variabler. Den ena av dessa *GLOBAL_height* sätts till rätt värde i *myReshape* med `GLOBAL_height = height`.

```
int old_x, old_y, GLOBAL_height;

void myMouse(int btn, int state, int x, int y){
    // Mouse coordinates have their origin in the upper left corner
    y = GLOBAL_height - y;
    myFigure(x,y);[1]
    old_x = x; old_y = y;
    glFlush();  // annars syns kanske inte ritningen direkt
}
void myMoveMouse(int x, int y){
    // Mouse coordinates have their origin in the upper left corner
    y = GLOBAL_height - y;
    glBegin(GL_LINES);
       glVertex2f(old_x,old_y);  glVertex2f(x,y); // 2d eller 2i lika bra
    glEnd();
    old_x = x; old_y = y;
    glFlush();  // annars syns kanske inte ritningen direkt
}
```

1. Olämpligt - men just nu bekvämt - sätt. All ritning bör göras i omritningsproceduren (precis som i Java), som inte skall anropas direkt. Anropa i stället *glutPostRedisplay* för att skapa en uppdateringshändelse.

Vid anropet av dessa procedurer förses de automatiskt med bl a musens koordinater vid händelsens inträffande. Eftersom fönsterkoordinatsystemet och muskoordinatsystemet skiljer sig (se nästa avsnitt) måste vi göra en mindre konvertering av y-koordinaten. I *myMoveMouse* ritar vi ett streck med glBegin/glEnd-konstruktionen.

Hela programmet finns som GL_ENKEL2.c. ❋

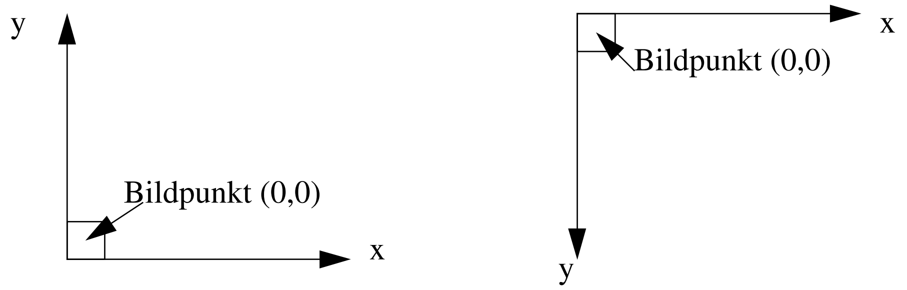
4 Samma exempel i Ada

Det går lika bra att koda i C++ eller Ada (gäller såväl Sun som PC).

Resten av avsnittet - som innehöll Ada-koden motsvarande exempel 1 och 2 i förra avsnittet - borttaget inför hösten 2004. Kan fås separat om någon är intresserad.

5 Koordinatsystem

I arbetet möter vi många olika koordinatsystem. GLUT använder sig liksom många andra system av ett heltalskoordinatsystem med y-axeln nedåtriktad, se högra figuren. Översta bildpunkten har koordinaten (0,0). OpenGL har däremot ett fönsterkoordinatsystem med y-axeln uppåtriktad, se vänstra figuren. Alltså är det nu den nederst vänstra bildpunkten som har koordinaterna (0,0). Upplagt för förvirring! Detta förklarar tolkningen av muskoordinaterna ovan, som ju sköts av en GLUT-rutin.



OpenGLs fönsterkoordinatsystem med glOrtho(0.0, width, 0.0, height, -1.0, 1.0)

GLUT:s fönsterkoordinatsystem

6 Kompilering och länkning

6.1 Linux på MD

Jag har gjort i ordning en kommandoprocedur som underlättar. Säg att vi har ett program som heter `OpenGL.c`. Då kan vi använda kommandot (kommandoprocedur, dvs textfil)
`OGL_COMPILE OpenGL`
för att få ett körbart program, vars namn blir *OpenGL*. Kommandoproceduren ser till att rätt inkluderings- och biblioteksfiler används. Den som vill göra på något annat sätt kan naturligtvis studera innehållet i kommandoproceduren, som ju är en ren textfil. Detta kommando förutsätter att man är inloggad på kurskontot eller att sökvägen utvidgats, men bör för övrigt kunna användas på godtycklig maskin. Det finns varianter av `OGL_COMPILE` för speciella ändamål (titta i `$DG/LINUX/bin`). Den som föredrar make-filer tillverkar lätt sådana utifrån `OGL_COMPILE`.

Om man använder `OGL_COMPILE` skall man för närvarande kunna köra det resulterande programmet i laborationssalarna 6220 (bäst grafikkort) och 6225 (ev sämre), men inte säkert i övriga.

6.2 PC med Windows

OpenGL, GLUT och MUI kan köras under såväl det kommersiella systemet Microsoft Visual C++, som den fria produkten GCC (C, C++ och Ada). Här ges allmän information. Jag har gjort ett särskilt dokument "OpenGL på PC", vilket ger alla detaljer för arbete på egen PC. Institutionen tillhandahåller en preliminär och inte helt testad OpenGL-miljö på de PC-ar med Windows som finns 6217 mittemot den ordinarie labsalen 6217.

I tabellen nedan visas vilka **DLL-er** (Dynamic Link Libraries, motsvarande delade bibliotek (.so) i Unix/Linux), som behövs och lämpligen läggs i `C:\Windows\System`. (Windows 95-98; annan placering i övriga). DLL-erna från Microsoft finns redan på plats. Övriga DLL-er utnyttjar Microsofts och kan hämtas fritt via nätet. Nate Robbins har anpassat GLUT till PC och det är hans version vi behöver. I hans paket ingår massor med exempel. Nate Robbins bifogar också manualblad till GLUT på HTML-form. När det gäller övrig dokumentation är det sämre beställt. Dock finns fullständig information tillgänglig via nätet, men den är försedd med en hård copyright.

DLL	Version	Leverantör/Impl
opengl32.dll	OpenGL 1.1	Microsoft
glu32.dll	1.3	Microsoft
glut32.dll	3.7	Nate Robbins

Vidare behöver man **inkluderingsfiler** (h-filer: gl.h, glu.h och glut.h) och **bibliotek** (lib-filer alternativt a-filer: opengl32.lib, glu32.lib, glut32.lib alternativt libopengl32.a, libglu32.a, libglut32.a). Med undantag av glut-filerna brukar de medfölja kompilatorn. Glut-filerna och ev ändrade .a-filer kan hämtas via kurssidan. SGI tillhandahöll tidigare egna DLL-er, men de finns från början av 1998 inte längre officiellt. MUI kräver ingen extra DLL.

För ytterligare detaljer betr MS Visual C++, se kurssidan. När det gäller gratisprodukter för C rekommenderar jag GCC (gcc/g++ ingående i MinGW eller Cygwin) som även klarar C++. Det behövs ersättare till lib-filerna, s k a-filer, som bl a jag gjort i ordning. Kompilering/länkning av en programfil `Prog.c` görs sedan med (en rad)

```
gcc Prog.c -lglut32 -lglu32 -lopengl32
```

som ger ett körbart program `a.exe`. Om du vill att programmet skall heta `Prog.exe`, lägg till `-o Prog.exe` direkt efter `gcc`. Om du använder Cygwin (gäller våra PC-ar) och vill ha ett klickbart, helt fristående program lägg till flaggan `-mno-cygwin`. Annars är programmet beroende av en speciell DLL.

7 Hur byggs objekten upp?

Egentligen byggs alla objekt upp utifrån punkter. En punkt beskriver vi med

```
void glVertex2f(GLfloat x, GLfloat y)
```

eller

```
void glVertex3f(GLfloat x, GLfloat y, GLfloat z)
```

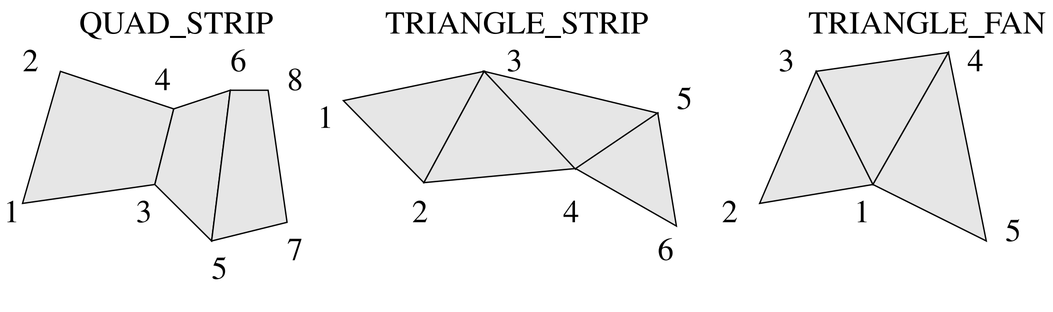
beroende på om vi arbetar i två eller tre dimensioner. I praktiken är dock allt tre-dimensionellt och anrop av den första proceduren är samma sak som att anropa den andra med z = 0. Alternativt kan vi använda *glVertex3d* med parametertypen *GLdouble*.

Ett antal anrop av *glVertex* samlas i en bunt

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

I det inledande exemplet använde vi GL_POLYGON (anses ha typen *GLenum*, som i praktiken är samma sak som *GLint*) 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.



Alla kort för 3D-grafik kan hantera trianglar och linjer. Polygoner bryts därför i praktiken ofta ner i sådana. Genom att beskriva en yta som en triangel-remsa (TRIANGLE_STRIP) eller triangel-solfjäder (TRIANGLE_FAN) behöver bara en ny punkt skickas till grafikprocessorn för trianglarna efter den första, vilket kan spara en hel del tid.

För fallet GL_POINTS kan vi sätta punktstorleken med `void glPointSize(GLfloat size)`. Om t ex size=3, ritas varje punkt som 3x3 bildpunkter. Angiven placering avser en ungefärlig mittpunkt. Vi kan (för fallet GL_LINES) sätta linjetjocklek med `void glLineWidth(GLfloat width)`. Standarden motsvarar size=1 resp width=1.0. Dessa inställningar gäller till dess de ändras.

När det gäller ytor som trianglar och polygoner är det viktigt att hörnen anges på ett konsistent sätt, förslagsvis i moturs ordning sett utifrån det objekt som ytan anger en del av. Skälen till detta avhandlas i läroboken.

Polygoner skall i praktiken vara plana, vilket kan vara litet svårt att åstadkomma själv i tre dimensioner om antalet hörn överstiger 3. I annat fall kan resultaten bli märkliga. Därför är trianglar betydligt vanligare.

Normalt ritas ytor fyllda. Men med anrop av typen

```
glPolygonMode(GL_FRONT_AND_BACK, ritsätt);
```

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

Exempel 3: Vi ser på ett par sätt att rita en enhetskub. Det sparar en del skrivarbete om vi lägger de åtta hörnpunkterna i en vektor.

```
GLfloat P[8][3]={{0,0,0},{1,0,0},{1,1,0},{0,1,0},{0,0,1},{1,0,1},
                 {1,1,1},{0,1,1}};
```

Vi kan nu rita kuben som sex polygoner varav en t ex (C indexerar från 0!)

```
glBegin(GL_POLYGON)
    glVertex3fv(P[0]); glVertex3fv(P[3]);
    glVertex3fv(P[2]); glVertex3fv(P[1]);
glEnd();
```

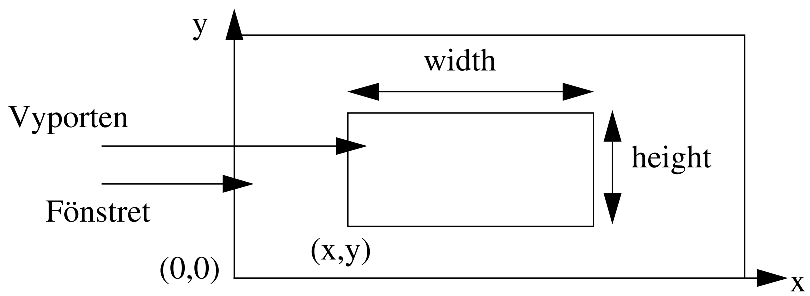
eller som en fyrhörningsremsa följd av två polygoner eller fyrhörningar. ❋

8 Modellering i 3D och 3D-betraktande

Detta avsnitt förutsätter att du är bekant med transformationer i 3D och matrisrepresentation av sådana. Se bok eller utdelat material.

Vi arbetar i modelleringsarbetet i OpenGL precis som i vår verkliga värld i ett högerorienterat tredimensionellt koordinatsystem. Detta system kallas i litteraturen omväxlande **världskoordinatsystem** eller **objektkoordinatsystem**. Vi placerar objekt i vår värld genom att beskriva dem i absoluta koordinater eller genom att transformera normaliserade objekt givna i ett **modellkoordinatsystem**. Vi för sedan in en användare (kamera) i världskoordinatsystemet som riktar in sitt eget koordinatsystem, ofta kallat **vykoordinatsystem** eller **ögonkoordinatsystem**. Det användaren ser projiceras sedan med parallellprojektion eller perspektivprojektion till **projektionskoordinater** eller **normaliserade koordinater**. Slutligen sker en transformation till **fönsterkoordinater**.

Det sista steget beskrivs vanligen först i omskalningsproceduren (den som kopplats med *glutReshapeFunc*) genom anrop av proceduren



```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

(x,y) anger nedre vänstra hörnet (i OpenGL-s koordinatsystem) för den ruta (vyport) i fönstret som skall användas för ritningen, medan de två sista parametrarna anger bredd och höjd. När vi vill utnyttja hela fönstret är (x,y)=(0,0) och *width* och *height* lika med fönstrets bredd och höjd (kommer ju som parametrar till omskalningsproceduren).

Åter till de övriga stegen. Om exempelvis ett hörn i en kub från början har koordinaterna P = $(x,y,z,1)^T$ har det i projektionskoordinatsystemet koordinaterna P' = $(x',y',z',1)^T$ varvid wP'=**Proj*****V*****Mod***P, där var och en av matriserna **Proj**, **V** och **Mod** är 4x4. Talet w är den fjärde koordinaten i wP' och måste divideras bort för att vi skall få fram P'. **Proj** står för projektionstransformationen (från vykoordinater till projektionskoordinater), **V** för vytransformationen (från världskoordinater till vykoordinater) och **Mod** för transformationer i modelleringsarbetet. Ofta skall samma typ av objekt finnas på flera platser i vår scen eller också skall vissa objekt röra sig. **Mod** kan alltså skilja sig mellan olika objekt.

I OpenGL kallar man kombinationen **M** = **V*****Mod** för **modell-vy-matrisen** och **Proj** för **projektionsmatrisen**.

I OpenGL konstruerar man de båda systemmatriserna **Proj** och **M** var för sig (de finns alltså automatiskt, men är inte direkt tillgängliga under dessa namn). **Proj** gäller för hela scenen, medan **M** som sagt kan variera mellan de olika objekten. Vi kan arbeta på ren matrisnivå, vilket är föga intuitivt och väl lågt. I stället konstruerar vi i allmänhet modell-vy-matrisen **M** med OpenGL-rutinerna *glTranslate*, *glScale*, *glRotate* och *gluLookAt*. Projektionsmatrisen **Proj** konstruerar vi med *glOrtho* eller *gluPerspective*. Nej, det är inget skrivfel. Ovanpå OpenGL finns också ett bekvämlighetsbibliotek **GLU** (Graphics Utility Library), från vilket vi framför allt kommer att utnyttja *gluLookAt* och *gluPerspective*. Inga extra åtgärder behövs för att komma åt dessa rutiner.

Vi påverkar **Proj** och **M** med samma typer av operationer. Vi måste därför ställa in vilken av matriserna som skall påverkas. Detta sker med proceduranropet

`glMatrixMode(GL_MODELVIEW)` resp `glMatrixMode(GL_PROJECTION)`

Vi måste också initiera den aktuella matrisen med anropet `glLoadIdentity()`, som ser till att en enhetsmatris stoppas in. Typiskt anropas denna två gånger. En gång för modell-vy-matrisen och en gång för projektionsmatrisen. Det finns ett antal matrisorienterade procedurer för att påverka **Proj** och **M**. Men vi tar inte upp dem här. I stället diskuterar vi mer intuitiva sätt att påverka transformationsmatriserna.

Låt oss börja med de rutiner som används i modelleringsarbetet:

```
glTranslatef(GLfloat dx, GLfloat dy, GLfloat dz)
glScalef(GLfloat sx, GLfloat sy, GLfloat sz)
glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)
```

Ett sådant anrop bildar motsvarande transformationmatris M_{lok} och multiplicerar gällande matris - typiskt **M** - från höger med denna enligt **M** = **M***M_{lok}. Om vi gör anropen

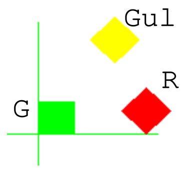
```
  glTranslatef(...); glRotatef(...)
```

får vi alltså **M** = **M***T*R, dvs i praktiken görs rotationen först och sedan translationen. Detta är en påtaglig svårighet för åtminstone nybörjare (ett alternativt tankesätt beskrivs i litteraturen).

Procedurerna *glTranslatef* och *glScalef* gör det vi väntar oss. Proceduren *glRotatef* roterar medurs den angivna vinkeln (**i grader!**) kring linjen från origo till (x,y,z) (i världskoordinatsystemet).

Ofta skall objekten transformeras på olika sätt. För att inte alltid behöva starta på ny kula med en enhetsmatris, kan man utnyttja möjligheten att "stacka" den aktuella matrisen. Detta görs med proceduranropet `glPushMatrix()`. Avstackning sker med `glPopMatrix()`. Detta är OpenGL's mekanism för hierarkiska strukturer!

Exempel 4: Skadar säkert inte med ett exempel. Vi visar omritningsproceduren i ett program som först ritar en liten (G=grön) fyrkant med ena hörnet i origo. Fyrkanten vrids sedan 45 motsols (i xy-planet, medsols kring z-axeln) och translateras därefter längs positiva x-axeln. Resultatet blir den (R=röda) fyrhörning som vilar på x-axeln. Som jämförelse visas också vad en translation följd av en rotation skulle ge, nämligen den tredje (gula) fyrhörningen. Ett helt annat resultat. Ordningen är alltså avgörande.




```
void display(void) {
    glClearColor(1.0,1.0,1.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,1.0,0.0);
    axlarna();                                   // Ritar axlarna
    myFigure(0,0);                               // Ritar fyrhörningen G
    glPushMatrix();                              // Spara matrisen M
       glTranslated(90,0,0); glRotated(45,0,0,1);// Rotera och translatera
       glColor3f(1.0,0.0,0.0);
       myFigure(0,0);                            // Ritar R
    glPopMatrix();                               // Hämta sparade M
    glPushMatrix();
       glRotated(45,0,0,1); glTranslated(90,0,0);// Translatera och rotera
       glColor3f(1.0,1.0,0.0);
       myFigure(0,0);                            // Ritar Gul
    glPopMatrix();
}
```
✻

Vi fortsätter med en procedur för vytransformationen. Eftersom den skall utföras sist måste motsvarande procedur anropas först när vi arbetar med modell-vy-matrisen. Typiskt ser det ut så här i omskalningsproceduren (OBS! Om betraktaren rör sig måste dessa rader i stället placeras efter suddningen i uppdateringsproceduren, som i detta exempel kallats *display*):

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(ögats plac, vad vi tittar på, uppåtvektor);
```

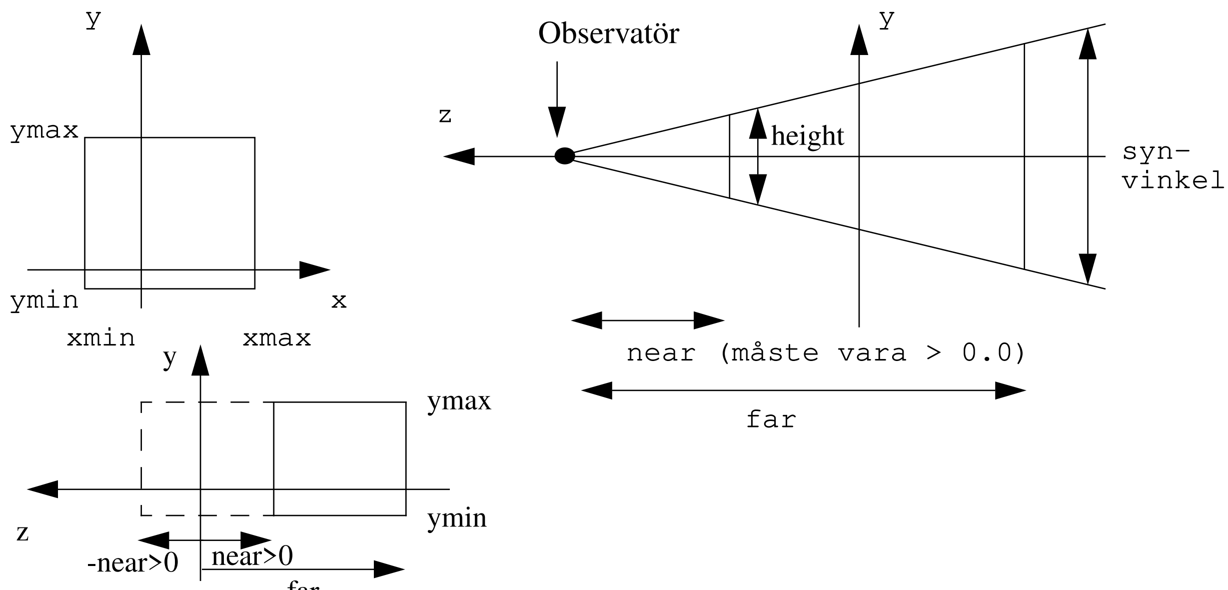
Var och en av våra parametrar till *gluLookAt* är i själva verket tre stycken, alla av typen *GLdouble*. Koordinaterna anges i världskoordinater, dvs det koordinatsystem där vi startade. Betraktningspunkten ("andra" parametern) avbildas på fönstrets mittpunkt och uppåtriktningen blir rakt uppåtriktad i fönstret.

Som standard - om man inte anropar *gluLookAt* - tittar man från origo med y-axeln som uppåtriktning och i negativ z-riktning. Det kan också vara intressant att tänka igenom att därför är t ex
`gluLookAt(0,0,3,   0,0,0,  0,1,0);`
samma sak som
`glTranslatef(0,0,-3);`

Till slut det återstående steget, dvs projektionstransformationen. Typiskt ligger anropen direkt efter *glViewport*-anropet i omskalningsproceduren och ser ut så här:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```
direkt följt av antingen (**parallellprojektion**, även kallad ortografisk projektion)
```
glOrtho(xmin, xmax, ymin, ymax,
        avstånd near från z=0 till närmsta klipp-plan,
        avstånd far från z=0 till bortersta klipp-plan);
```

eller (**perspektivprojektion**)
```
gluPerspective(synvinkel i vertikala tittplanet, synkvot,
               avstånd near till närmsta klipp-plan,
               avstånd far till bortersta klipp-plan);
```



Parallellprojektion
(vykoordinater)

Perspektivprojektion
(vykoordinater)

Samtliga parametrar till *glOrtho* och *gluPerspective* är av typen *GLdouble*. Med *gluPerspective* definieras betraktningspyramiden. Synvinkeln skall ligga mellan 0.0 och 180.0 grader och synfältet ligger symmetriskt kring z-axeln. Synkvoten är förhållandet mellan projektionsplanets bredd och höjd (i allmänhet ritytans bredd resp höjd), dvs den bestämmer indirekt synvinkeln i det horisontella planet. Som projektionsplan kan vi tänka oss vilket plan som helst mellan främre och bortre klipp-planet. Projektionsplanet avbildas på den ruta som definierats av *glViewport*.

När det gäller parallellprojektionen kommer innehållet i ett rätblock
`[xmin,xmax]x[ymin,ymax]x[-near, -far]`
att parallellprojiceras på den ruta som definierats av *glViewport*. Se de vänstra figurerna ovan. Om

near>0 innebär det ett block bakom xy-planet. Om near<0, finns blocket både framför och bakom detta plan (streckning i figuren). Normalt är synriktningen i negativ z-led (detta har bara betydelse om djuptest är påslaget), men om både *near* och *far* är negativa är den i positiv z-led.

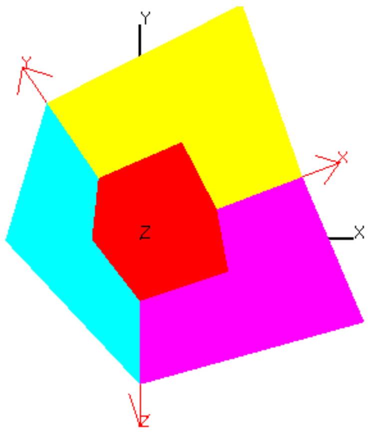
Det gäller naturligtvis för både `glOrtho` och `gluPerspective` att man måste välja klipplanen så att inte det som skall synas klipps bort. I `gluPerspective` måste parametern *near* vara positiv, dvs 0.0 duger inte (resultatet kan bl a bli att inget ritas).

Exempel 1, forts: Låt oss belysa parallellprojektion i anslutning till exempel 1. I omskalningsproceduren *myReshape* fanns

```
    glViewport(0, 0, width, height);              // Hela fönstret utnyttjas
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    glOrtho (0.0, width, 0.0, height, -1.0, 1.0); // Hur mycket skall synas
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
```

som säger bl a att x=0 motsvaras fönstrets vänsterkant och att x=fönsterbredden (width) i bildpunkter motsvarar högerkanten samt att utrymmet mellan z=-1 och z=1 skall tas med. Vi har i vårt ritarbete använt z=0. Dessa satser är typiska då vi vill använda OpenGLs eget fönsterkoordinatsystem för renodlat 2D-arbete. ✻

Exempel 5: Vi visar omskalningsproceduren som ingår i ett exempel `GL_DUBBLAKOORD.c` där man kan vrida sin 3D-scen (tre väggar med en kub i ena hörnet. Se figuren nedan. Världskoordinatsystemets axlar är ritade med litet fetar linjer.




```
void myReshape(int width, int height){
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(55.0,width/((GLfloat)height,1.0,5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0,0,3, 0.0,0.0,0.0, 1.0,0.0,0.0);
}
```

I omritningsproceduren *myUpdate* (som vi inte har plats för här) ritas dels världskoordinatsystemet och dels den vridna scenen. Vridningen åstadkommes med

```
    glRotatef(VinkX,1.0,0.0,0.0);
    glRotatef(VinkZ,0.0,0.0,1.0);
```

dvs vi roterar **först** kring världskoordinatsystemets z-axel och sedan kring x-axeln. Vi kan styra vridningen med tangenter genom att införa en funktion för tangenthändelser med

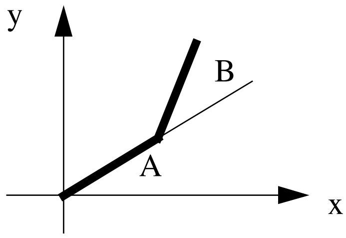
```
glutKeyboardFunc(Keyfunc);
```
och själva tangenthanteraren
```
void Keyfunc(unsigned char key, int x, int y) {
   if (key == 'x') VinkX = VinkX + 3.0;
   if (key == 'z') VinkZ = VinkZ + 3.0;
   myUpdate();// Hellre glutPostRedisplay(), se avsnitt 9
}
```

Varje tryckning på tangent x eller z ökar motsvarande vinkel och framtvingar en omritning, så att vi ser det aktuella utseendet. När vi ändrar z-vinkeln, sker vridningen kring den vridna scenens z-axel (pilförsedd i figuren ovan)! Men ändring av x-vinkeln vrider kring världskoordinatsystemets x-axel (utan pil). Förklaringen är att rotationen kring z-görs först! Det är rotationen kring x-axeln som gör att scenens z-axel avviker från det fasta världskoordinatsystemets.

Om vi i stället hade velat ha en evig rörelse runt x-axeln, hade vi infört en idle-funktion med
```
glutIdleFunc(idle);
```
och
```
void idle() {
   VinkX = VinkX + 3.0;
   myUpdate(); // Hellre glutPostRedisplay(), se avsnitt 9
}
```
❋

Om man av någon anledning skulle vilja inspektera de matriser som bildas går det utmärkt. T ex skriver följande procedur ut kolumn efter kolumn i **modell-vy-matrisen**.
```
void CheckMatrix() {
      GLfloat v[16]; int i;
      // lägger in kolumnerna i modell-vy-matrisen i v
      glGetFloatv(GL_MODELVIEW_MATRIX,v);
      for (i=0;i<16;i++) {
          printf("%d %f\n",i,v[i]);
      }
}
```
Trevligare är att få matriserna utskrivna rad för rad, vilket görs av följande förbättrade version. Den tar en parameter som är antingen `GL_MODELVIEW_MATRIX` eller `GL_PROJECTION_MATRIX` och skriver ut motsvarande matris, dvs **modell-vy-matrisen** eller **projektionsmatrisen**.
```
void CheckMatrix(GLenum M) {
    GLfloat v[16];
    int i,j;
    glGetFloatv(M,v);
    if (M==GL_PROJECTION_MATRIX)
        printf("Utskrift av projektionsmatris\n");
    else printf("Utskrift av modellvy-matris\n");
    for (i=0; i<4; i++) {
        for (j=i; j<16; j=j+4) {
            printf(" %f",v[j]);
        }
        printf("\n");
    }
}
```

Exempel 6: Vi tittar på ett exempel där två axlar kan vridas var för sig i xy-planet, dvs rotation kring z-axeln. Ändringen av vridningsvinklarna A och B (som måste vara globala i programmet) kan vi låta ske som i förra exemplet. Vi visar bara proceduren som ritar ett streck och två varianter av omritningsproceduren.




```
void myFigure(void){
    glBegin(GL_LINES);     // Horisontellt streck
        glVertex3f(0.0,0.0,0.0); glVertex3f(1.0,0.0,0.0);
    glEnd();
    glFlush();
}
```

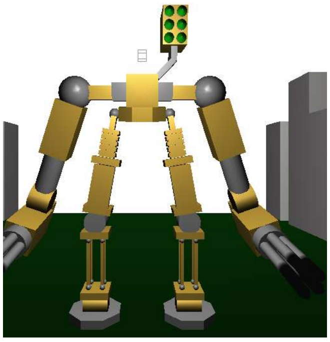
I den första omritningsproceduren ritas den första leden vriden A grader (motsols). Den yttre leden (leden har fortfarande utgångspunkt i origo) vrids först B grader motsols kring origo och translaterar sedan 1 enhet längs x-axeln och roteras **till sist** ytterligare A grader motsols kring z-axeln.

```
void myUpdate(void) {
    glClearColor(1.0,1.0,1.0,0.0); glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
       glColor3f(0.0,1.0,0.0);
       // Första leden
       glRotated(A,0,0,1); myFigure();
       // Andra leden
       glTranslated(1,0,0); glRotated(B,0,0,1); myFigure();
    glPopMatrix();
}
```

I den andra gör vi på samma sätt för den första leden. Den andra vrids direkt A+B grader och translateras därefter till rätt punkt. Den ursprungliga vridningen med A grader upphävs i koden med en lika stor vridning åt andra hållet. I ett större exempel skulle man i stället gjort det med ett *glPushMatrix*/*glPopMatrix*-par.

```
void myUpdate(void)  {
    glClearColor(1.0,1.0,1.0,0.0); glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
       glColor3f(0.0,1.0,0.0);
       // Första leden
       glRotated(A,0,0,1); myFigure();
       // Andra leden
       glRotated(-A,0,0,1);            // Upphäv
       glTranslated(cos(A/360*2*M_PI),sin(A/360*2*M_PI),0);
       glRotated(A+B,0,0,1); myFigure();
    glPopMatrix();
}
```

Förklara för dig själv varför det finns *glPushMatrix/glPopMatrix* i *myUpdate*. Samma tankegångar som här är naturligtvis överförbara till riktiga modeller. En något mer komplicerad modell av denna typ syns här (programmet `glutmech.c` i GLUT-distributionen). *



9 Djupbuffert, dubbelbuffring, buffring av händelser

Med proceduren *glutInitDisplayMode* begär vi hårdvaruresurser. Ett typiskt anrop är

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH)
```

som innebär att vi vill ha dubbelbuffring, äkta RGB-grafik (GLUT_RGB betyder samma sak) samt djupbuffert. Om resursen inte finns bryts programmet med ett felmeddelande. Det finns också möjlighet att fråga om en viss resurs finns eller inte och anpassa sig efter svaret (det måste kommersiella program göra!). Alternativet till GLUT_DOUBLE är GLUT_SINGLE. Djupbufferten hjälper oss att lösa synlighetsproblemet. **Det räcker dock inte att sådan finns, utan vi måste också slå på dess användning med `glEnable(GL_DEPTH_TEST)`**. Dessutom måste vi vid varje omritning ha `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

När dubbelbuffring används växlar vi mellan de två buffertarna med `glutSwapBuffers()` (`glFlush` anropas då automatiskt). Anropet görs lämpligen i slutet av uppdateringsproceduren. Dubbelbuffring är nödvändig vid all animering eller rörelse i en scen. Om du begärt dubbelbuffring och inte ser det du begärt ritat, kan det bero på att `glutSwapBuffers` inte anropats.

I exempel 5 framtvingade vi omritning från tangentproceduren genom att anropa *myUpdate*. Detta fungerar bra om hanterandet av händelserna sker i samma takt som händelserna inträffar. Detta är dock sällan fallet. Om omritningen är omfattande byggs det upp en lång kö av händelser om vi håller nere en tangent (automatisk repetering gäller) och när vi släpper tangenten fortsätter rörelsen. Detta är inte en njutbar interaktion. Det är bättre att anropa `glutPostRedisplay()`, som bara lägger omritningshändelsen till kön om det inte redan finns en obearbetad av samma slag där. Vi missar alltså ibland enstaka omritningar, men i gengäld går det i rätt takt.

10 Färdiga modeller

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).

T ex ger

```
void glutWireCube(GLdouble size); // Med glScalef kan vi lätt ordna rätblock
void glutSolidCube(GLdouble size);
```

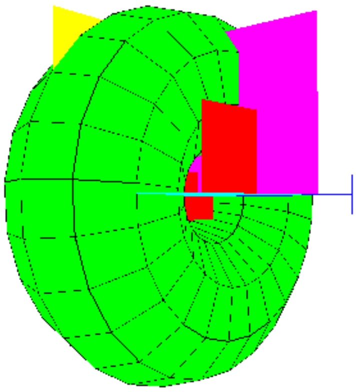
en kub (tråd- resp solidform) med sidan size och med mittpunkt i origo. För sfärer finns

```
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);
void glutSolidTeapot(GLdouble size); // Klassisk modell
void glutSolidOctahedron(void);
void glutSolidTetrahedron(void);
void glutSolidIcosahedron(void);
```

Exempel 7: Här har vi ritat en torus i vår tidigare väggscen.



11 Färg

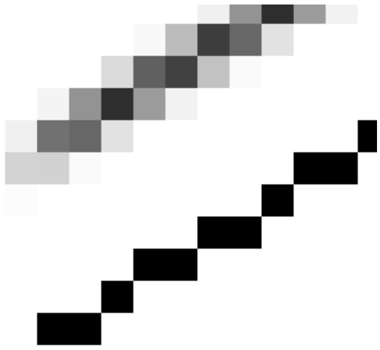
Vi har redan sett hur det går till att sätta aktuell ritfärg. Vi kan göra det var som helst i koden och till och med för enstaka punkter inuti glBegin/glEnd. Vad händer med fyllda objekt om vi har olika färger för hörnen? I normalfallet används ett visst hörns färg och objektet blir enfärgat. Men med anropet `glShadeModel(GL_SMOOTH)` interpoleras i stället färgen över objektet. Normalfallet återfås med `glShadeModel(GL_FLAT)`.

Vi har förutsatt att resursen GLUT_RGBA är tillgänglig. För fullständighets skull bör det dock nämnas att det finns ett alternativ GLUT_INDEX som är aktuellt för datorer med enbart färgtabell. Kvaliten blir då sämre, vissa saker går inte att göra eller kräver annorlunda kodning.

12 Linjer, antialiasing

Vi har redan nämnt att linjetjocklek kan sättas med `void glLineWidth(GLfloat width)`. Streckade linjer går också lätt att åstadkomma, nu med `void glLineStipple(GLint factor, GLushort pattern)`. Den andra parametern ger ett 16-bitars mönster och kan t ex vara `0xa57f`. Den första parametern *factor* bestämmer hur många gånger som en bit skall upprepas. Streckningen måste också slås på med `glEnable(GL_LINE_STIPPLE)`. Kan sedan upphävas med `glDisable(GL_LINE_STIPPLE)`. Ytor kan mönstras med en liknande teknik.

Något svårare är det att producera antialiasade linjer och vi anger ingen kod utan nöjer oss med att visa ett resultat i två olika uppförstoringar.



13 Animeringsmod (Xor-mod)

Från och med OpenGL 1.1 finns denna mod även för 24-bitars-färg som vi är intresserade av.

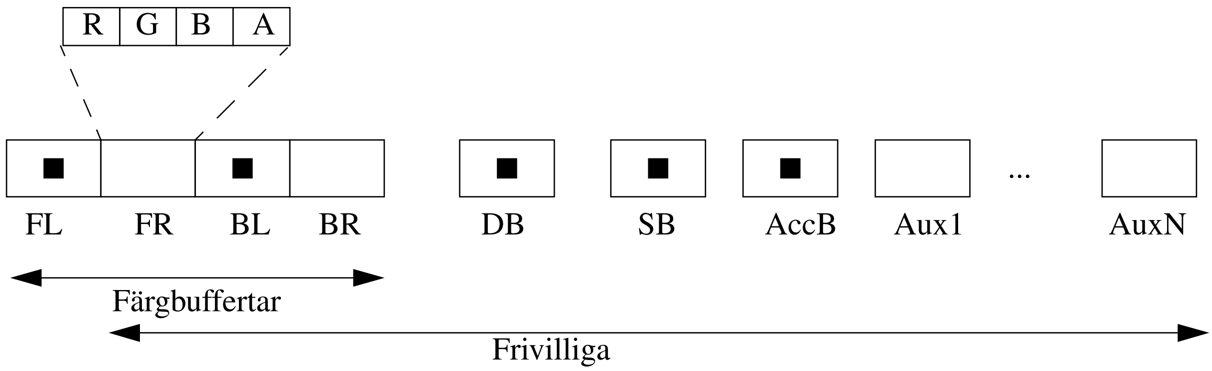
Exempel 8: Ibland vill man låta ett objekt röra sig över en bakgrundsbild utan att denna förstörs. Detta kan göras enligt modellen i följande kodavsnitt som låter en figur glida horisontellt. Vid ritning i Xor-mod "inverteras" de bildpunkter som normalt skulle ritats, genom att befintligt färgvärde och pålagt färgvärde kombineras bitvis med XOR, vilket betyder att efter en andra ritning av samma objekt är bakgrunden oförändrad.

```
void display(void)
{
    int x, y;
    glClearColor(1.0,0.0,0.0,0.0);         // Röd bakgrund
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,0.0,1.0);                // Blå ritfärg
    glEnable(GL_COLOR_LOGIC_OP);
    glLogicOp(GL_XOR);
    for (x=0; x<300; x++) {
          myFigure(x,200);                 //  Rita
          glFlush();                       //  Annars flämt
          for (y=0; y<1000000; y++) x = x; //  Fördröjning, oproffsigt!
          myFigure(x,200);                 //  Sudda
          glFlush();
    }
}
```

Observera att det kan göras även i en 3D-värld (2D-världar är från OpenGLs synpunkt bara ett specialfall). Det finns andra sätt som kan ge liknande funktionalitet: rasterkopiering, lager. ❋

14 Buffertar

De olika buffertar som OpenGL tänker sig visas här:



Varje färgbuffert har delar för rött, grönt och blått (i allmänhet 8 bitar för varje). Dessutom kan det finnas en Alpha-del. Sådan måste på våra Linux-datorer begäras med `GL_ALPHA` i `glutInit-DisplayMode` (se avsnitt 9). Det räcker inte med `GL_RGBA`.

FL = Front Left, BL = Back Left
FR = Front Right, BR = Back Right
DB = Depth Buffer (djupbufferten)
SB = Stencil Buffer, AccB = Accumulation Buffer, Auxi = Extra buffert i

Dock är det inte alls säkert att alla finns i ett aktuellt system. Vi förutsätter att de med svart fyrkant finns. Det finns flera sätt att undersöka hur det står till

```
 // Deklarera två vektorer med vardera 5 element (1 hade räckt)
 GLint a[5];
 GLboolean b[5];
 // Fråga på
 glGetIntegerv(GL_AUX_BUFFERS,a); printf("Antal aux-buffertar %d\n", a[0]);
 glGetIntegerv(GL_DEPTH_BITS,a); printf("Antal bitar per pixel i djupbuffert
%d\n", a[0]);
 glGetIntegerv(GL_RED_BITS,a); printf("Antal röda bitar per pixel  %d\n", a[0]);
 glGetIntegerv(GL_ALPHA_BITS,a); printf("Antal alfabitar per pixel  %d\n", a[0]);
 glGetIntegerv(GL_STENCIL_BITS,a); printf("Antal stencil bitar per pixel  %d\n",
a[0]);
 glGetIntegerv(GL_ACCUM_RED_BITS,a); printf("Antal röda ackumuleringsbitar per
pixel  %d\n", a[0]);
 glGetBooleanv(GL_STEREO,b); printf("Stödjer stereo?%d\n", b[0]);
 glGetBooleanv(GL_DOUBLEBUFFER,b); printf("Stödjer dubbel-buffring?%d\n", b[0]);
```

Resultat

```
Antal aux-buffertar 0
Antal bitar per pixel i djupbuffert 28
Antal röda bitar per pixel  8
Antal alfabitar per pixel  0
Antal stencil bitar per pixel  4
Antal röda ackumuleringsbitar per pixel  16
Stödjer stereo?1                    Jag körde på min arbetsstation som har det
Stödjer dubbel-buffring?1
```

När vi ändå är inne på frågor till OpenGL, så kan det också nämnas att man kan undersöka om en viss funktionalitet är påslagen eller ej. T ex är djuptestning påslagen om `glIsEnabled(GL_DEPTH_TEST)` har värdet 1 (true) och annars ej. Man kan som vi nämnt i ett tidigare avsnitt också läsa av vissa systemmatriser.

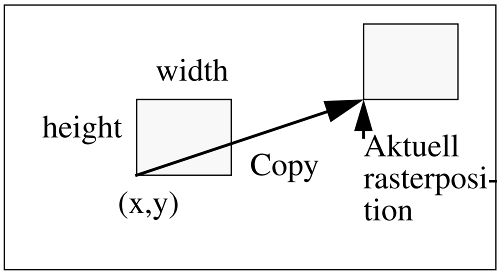
15 Rasterkopiering

Det finns många situationer där man behöver kopiera ett rektangulärt pixelområde från en plats till en annan. I OpenGL kan vi dels kopiera inom aktuell färgbuffert, dels mellan en buffert och primärminnet (dvs en variabel).

För det första fallet har vi

```
void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GL_COLOR);
```

vars funktion framgår av



Aktuell rasterposition sätts med någon av procedurerna

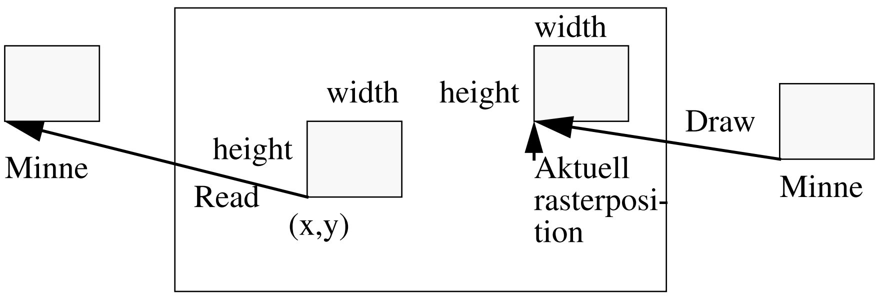
```
void glRasterPos2d(GLdouble x, GLdouble y);
void glRasterPos3d(GLdouble x, GLdouble y, GLdouble z);
```

Parametrarna avser världskoordinater, inte som i *glCopyPixels* och *glReadPixels* (nedan) fönsterkoordinater!

För det andra fallet har vi procedurerna

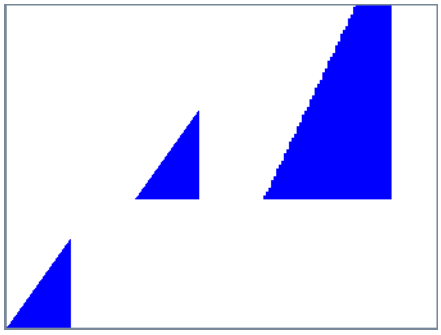
```
void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height,
                  GL_RGB, GL_BYTE, GLvoid *Minne);
void glDrawPixels(GLsizei width,GLsizei height, GL_RGB, GL_BYTE, GLvoid *Minne);
```

som kopierar från färgbufferten till variabeln *Minne* respektive från *Minne* till färgbufferten. *Minne* är i allmänhet en tillräckligt stor matris eller vektor: `GLbyte Minne[10000]` men kan naturligtvis också vara dynamiskt allokerad. Antalet element måste vara minst 3*(antalet bildpunkter).



Värd att nämna är också proceduren `glPixelZoom(GLfloat x_zoom, GLfloat y_zoom)` som ställer in skalningsfaktorerna vid ritning till skärmen. Som standard gäller naturligtvis (1.0,1.0). Det finns en hel del ytterligare detaljer, bl a hur lagring sker, vilket framför allt är viktigt att veta om man utanför grafikmiljön vill definiera ett raster.

Exempel 9: Vi ritar först en blå triangel längst ned till vänster. Därefter kopierar vi och får den som ligger snett ovanför. Till sist läser vi av den ursprungliga, lagrar i variabel och skickar tillbaka med förstoring.



```
void display(void)
{
    int x, y; GLbyte Minne[30000];
    glClearColor(1.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0,0.0,1.0);
    myFigure(0,0);
    // Nästa rad förutsätter att världskoordinater=fönsterkoordinater
    glRasterPos2d(100,100);
    glCopyPixels(0,0,50,70,GL_COLOR);
    glReadPixels(0,0,50,70,GL_RGB,GL_BYTE,Minne);
    glRasterPos2d(200,100); glPixelZoom(2.0,3.0);
    glDrawPixels(50,70,GL_RGB,GL_BYTE,Minne);
    glPixelZoom(1.0,1.0);
}
```
❋

Rasterkopiering kan användas för att visualisera innehållet i vissa andra buffertar. Vi "kopierar" då först med *glReadPixels* från den aktuella bufferten till primärminnet och sedan med *glDrawPixels* vidare till bildminnet. Vissa konverteringar kan bli nödvändiga och utförs automatiskt. Antag att vi har ett fönster om 400x400 och att vi vill visualisera djupminnet. Då behöver vi en minnesvariabel
`GLfloat DJUP_MINNE[400*400];`
Vi läser av djupminnet med (det är den 5:e parametern som bestämmer bufferten)
`glReadPixels(0,0,400,400,GL_DEPTH_COMPONENT,GL_FLOAT,DJUP_MINNE);`
Sedan kan vi med
`glDrawPixels(400,400,GL_LUMINANCE,GL_FLOAT,DJUP_MINNE);`
skicka innehållet vidare till bildminnet. Vi använder flyttalen som luminansvärden, dvs punkter nära betraktaren blir mörka och punkter längre bort ljusare.

16 Mer om interaktion

Vi har något snuddat vid interaktion med hjälp av tangenter och mus. Det finns mycket ytterligare att säga.

Vi har sett (Exempel 5) hur händelser kan knytas till de vanliga tangenterna med *glutKeyboardFunc*. För specialtangenter används i stället `void glutSpecialFunc(void (*func) (int key, int x, int y))`. Obegripligt för annan än C-experten. Men det hela fungerar på samma sätt som för de vanliga, bortsett från att man använder namn som `GLUT_KEY_F1` (funktionstangent), `GLUT_KEY_LEFT` (piltangent) i stället för de vanliga tecknen. Styrning med piltangenter är väl oftast naturligare än det sätt vi hittills använt. Och styrning med mus ofta ännu naturligare.

Och nu något som är väl så viktigt. Under hanteringen av mushändelser måste vi naturligtvis kunna avgöra dels om knappen trycktes ned eller släpptes, dels vilken knapp det rör sig om. I exempel 2 nämnde vi `void myMouse(int btn, int state, int x, int y)`, som blev mustryckshanterare genom att vi skrev `glutMouseFunc(myMouse)`. Den andra parametern kan ha värdet `GLUT_DOWN` resp `GLUT_UP` och den första värdet `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` eller `GLUT_RIGHT_BUTTON`. Vi kan alltså avläsa hur det står till. Man kan också ta hand om musrörelser med musen nedtryckt (*glutMotionFunc*) respektive utan musen nedtryckt (*glutPassiveMotionFunc*).

Exempel 10: Som exempel tittar vi på ett kodavsnitt (ingår i GL_ROTATION.c), som kan användas för att med relativa musrörelser på ett intuitivt sätt styra två parvisa rotationsvinklar. Med vänstra musknappen nedtryckt styr vi rotation i y- och x-led. Med högra i stället i z- och x-led. I huvudprogrammet har vi anropat

```
  glutMouseFunc(myMouse); glutMotionFunc(myMotion);
```

Med *glutPassiveMotionFunc* hade vi bara kunnat styra ett par av vinklar.

Koden

```
// Globala musvariabler eftersom myMotion bara får reda på koordinater
int mouseX; int mouseY;
int mouseButton;

void myMotion(int x, int y) {
   if (mouseButton == GLUT_LEFT_BUTTON){
         VinkY = VinkY - (mouseX - x);
         VinkX = VinkX - (mouseY - y);
   } else if (mouseButton == GLUT_RIGHT_BUTTON){
         VinkZ = VinkZ + (mouseX - x);
         VinkX = VinkX - (mouseY - y);
   }
   mouseX = x; mouseY = y;
   glutPostRedisplay();
}
void myMouse(int btn, int state, int x, int y){
      mouseButton = btn;
      mouseX = x; mouseY = y;
}
```

❋

Interaktion med 3D-objekt via en 2D-skärm är inte helt lätt. Det finns många mjukvaru- och hårdvarulösningar utöver de enkla vi berört. GLUT ger visst stöd även för annan hårdvara.

17 Tidsstyrd uppdatering

Hittills har omritningsfrekvensen styrts av datorns hastighet eller manuellt. Det förra är ju inte så lyckat, eftersom ett skeende inte ska passera snabbare bara för att datorn har högre prestanda (däremot har vi inga invändningar mot att återgivningen har högre kvalitet). Till vår hjälp finns i GLUT tidsstyrning, som gör att vi kan få omritning med t ex 10 bilder per sekund (detta förutsätter dock vissa prestanda). Vi byter ut det typiska anropet `glutIdleFunc(idle)` mot `glutTimerFunc(100,tiden,5)`. Den första parametern anger tiden i millisekunder fram till dess att proceduren `tiden` (andra parametern) skall anropas. Proceduren `tiden` kan se ut så här (skrivsatsen enbart för teständamål; `TickCount` anges nedan):

```
void tiden(int value) {
    idle();
    // printf("Tidshändelse %d %d\n", value, TickCount());
    glutTimerFunc(100,tiden,value+1);
}
```

De åtgärder som tidigare utfördes så fort som datorn hinner, kommer nu att utföras som mest var 100:e millisekund. Observera att "väckarklockan" måste sättas på nytt. Parametern till tiden kan som här användas för att räkna antalet anrop el dyl.

I sammanhanget kan också nämnas något om **tidmätning**, vilket är aktuellt när man vill bedöma prestanda. Flera alternativ finns, men jag brukar införa en funktion *TickCount* enligt (fungerar under gcc på Unix såväl som PC), som ger en förfluten (dvs ej ren CPU-tid) tid i millisekunder (utgångstidpunkt året 1970).

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/timeb.h>
unsigned long    TickCount(){
        struct timeb tp;
        ftime(&tp);
        return 1000*(tp.time&0x0000ffff) + tp.millitm;
}
```

Genom att registrera starttid och sluttid kan vi få tiden för ett förlopp.

Exempel 11: Som exempel tittar vi på följande program.

```
> cat GL_TIDEX.c
/***************************************************************
*  Demonstrerar bara tidsstyrning och tidmätning
***************************************************************/
#include <math.h>
#include <GL/glut.h>
...
void display(void) { }
void myReshape(int width, int height) { }
void idle() {
   printf("idle-anrop %d \n", TickCount());
   glutPostRedisplay();
}
// Proceduren tiden som ovan
int main(int argc, char** argv) {
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(400,400);
    glutCreateWindow("Inget skall synas");
    glutReshapeFunc (myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(idle);
    //glutTimerFunc(100,tiden,5);
    glutMainLoop();
}
```

Körning

```
> GL_TIDEX
idle-anrop 54415655
idle-anrop 54415661
idle-anrop 54415663
idle-anrop 54415667
idle-anrop 54415677
idle-anrop 54415678
```

Tidsdifferenserna varierar mellan 1 och 10 ms. Om vi i stället för att anropa `glutIdleFunc` anropar `glutTimerFunc` blir tidsdifferenserna i stället som planerat kring 100 ms.

```
> GL_TIDEX
idle-anrop 54539273
idle-anrop 54539393
idle-anrop 54539503
idle-anrop 54539613
```

18 Menyer med GLUT

GLUT ger stöd för enkla s k uppdykningsmenyer, vilka kan knytas till någon av musknapparna. Dessa menyer kan vara hierarkiska, dvs i sin tur ha menyer.

Exempel 12: Som exempel tittar vi på ett program som när man trycker på vänstra musknappen visar upp menyn


```
> cat GL_MENYER.c
#include <stdlib.h>
#include <GL/glut.h>

/* Uppdateringsrutin: raderar enbart */
void myUpdate(void)
{
    printf("myUpdate called\n");
    glClearColor(1.0,1.0,1.0,0.0);
    glClear(GL_COLOR_BUFFER_BIT);
}
/* Fönsterrutin : behöver inte göra något */
void myReshape(int width, int height)
{
    printf("myReshape called!\n");
}

#define INFO 0
#define QUIT  1
```

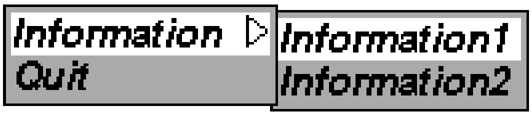
```
/* Menyhanteringsrutin för vänsterknappens meny */
void menuSelectLeft(int itemID)
{
  switch (itemID)
  {
    case INFO:
      printf("Left button - first alternative\n");
      break;
    case QUIT:
      printf("Left button - second alternative\n");
      exit (0);
      break;
  }
}
/* Huvudprogram */
int main(int argc, char** argv)
{
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
   glutInitWindowSize(400,200);
   glutCreateWindow("Menyer med GLUT");
   glutReshapeFunc (myReshape);
   glutDisplayFunc(myUpdate);
   /* De följande fyra raderna skapar menyn */
   glutCreateMenu (menuSelectLeft);
   glutAddMenuEntry ("Information", INFO);
   glutAddMenuEntry ("Quit", QUIT);
   glutAttachMenu (GLUT_LEFT_BUTTON);
   glutMainLoop();
   return 0;
}
```

Till varje meny hör en menyhanteringsfunktion, i det här fallet kallad *menuSelectLeft*, som hanterar val i menyn. Varje menyalternativ identifieras med ett godtyckligt heltal. En meny skapas med funktionen `glutCreateMenu(namnet på menyhanteringsfunktionen)`. Menyalternativen läggs till i tur och ordning med `glutAddMenuEntry(Text, identifieringheltal)`. Slutligen kopplas menyn till t ex vänstra musknappen med `glutAttachMenu(GLUT_LEFT_BUTTON)`.

```
> GL_MENYER
myReshape called!
myUpdate called
myUpdate called
Left button - first alternative
myUpdate called
Left button - second alternative
```

Det är viktigt att uppdateringsproceduren verkligen suddar. I annat fall försvinner inte menyn från skärmen. Vi kan ha en meny för var och en av musknapparna.

Hierarkiska menyer stöds som sagt. Låt oss bara bygga ut ut ovanstående exempel en aning.




```
menunr=glutCreateMenu (menuSelect);
glutAddMenuEntry ("Information1", 7);
glutAddMenuEntry ("Information2", 5);
glutCreateMenu (menuSelectLeft);
glutAddSubMenu("Information",menunr);
glutAddMenuEntry ("Quit", QUIT);
glutAttachMenu (GLUT_LEFT_BUTTON);
```

*

19 Text

Inte så sällan vill man skriva något i ett grafikfönster. OpenGL låter oss inte göra det på ett enkelt sätt, men det gör däremot GLUT. Tecken kan återges på två sätt, som ett raster (bitmap) eller som ett geometriskt objekt uppbyggt av linjer.

I det förra fallet sker en kopiering från ett raster till skärmen och i botten ligger alltså procedurerna för rasterkopiering. Tecknet hamnar plant på skärmen (om vi nu inte använder rastret för texturering). I det andra fallet ritas tecknet som alla andra geometriska objekt (i botten ligger linjeritning) och utsätts därmed för alla aktuella transformationer. Därmed kan vi lätt få text i 3D. Naturligtvis kan man spinna vidare på detta och låta tecknen vara 3D-objekt med all tänkbar glans, men än så länge finns ingen färdig procedur för det utan då får man anstränga sig själv.

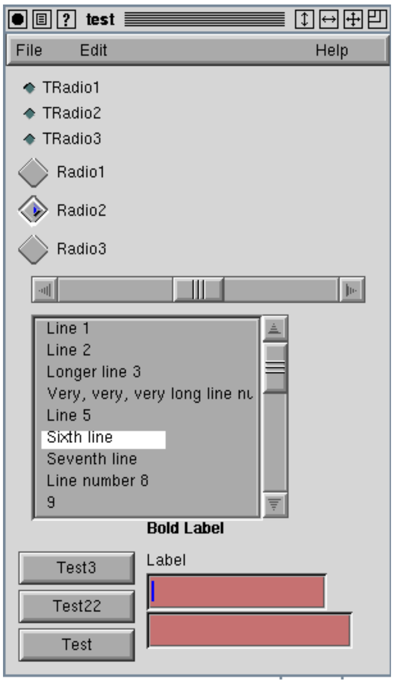
Rastertecken:
1. Placera pennan med `glRasterPos3f(x,y,z)`. Detta behöver bara göras för första tecknet i en följd. Punkten (x,y,z) transformeras som alla andra automatiskt till skärmkoordinater, så det är lätt att få tecken i närheten av ett visst objekt.
2. Anropa `glutBitmapCharacter(GLUT_BITMAP_8_BY_13, tecknet)` för varje tecken i en sträng. Den första parametern anger vilket typsnitt som skall användas. Just detta typsnitt är fixt (dvs alla tecken upptar lika stor plats) och finns även som 9_BY_15, men det finns också proportionella, t ex GLUT_BITMAP_TIMES_ROMAN_10.

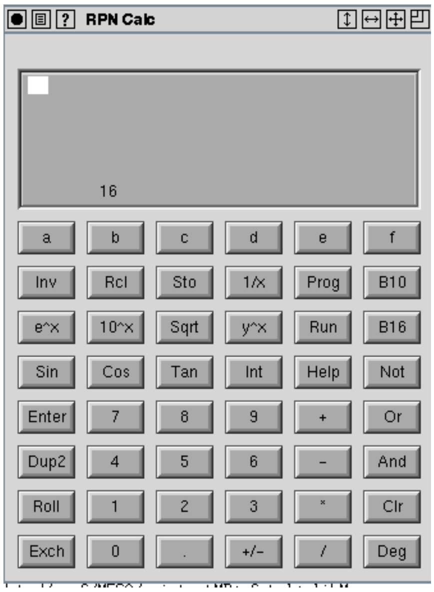
Geometriska tecken:
1. Anropa `glutStrokeCharacter(GLUT_STROKE_ROMAN, tecknet)` för varje tecken i en sträng. Den första parametern anger vilket typsnitt som skall användas. Tecknen i detta typsnitt är 2D-objekt och har höjdkoordinater (y) mellan ungefär -33 och 119, medan breddkoordinaten (x) startar kring 0.0. Vi måste därför i regel skala ned objektet kraftigt. Efter varje ritat tecken sker automatiskt en translation i x-led motsvarande tecknets bredd.

Strängar:
Vanliga GLUT har inget stöd för strängar utan man får själv genomlöpa strängen. GLUT finns emellertid, som nämnts, i en version *freeglut* med strängstöd:
glutBitmapString(typsnittsnamn som ovan, strängen)
glutStrokeString(typsnittsnamn som ovan, strängen)
I kursen används nog normalt GLUT. Om du inkluderar `freeglut.h` i st f `glut.h` och låter bli att skriva `setup_course TDA360` i ett nytt fönster, används *freeglut*.

20 MUI

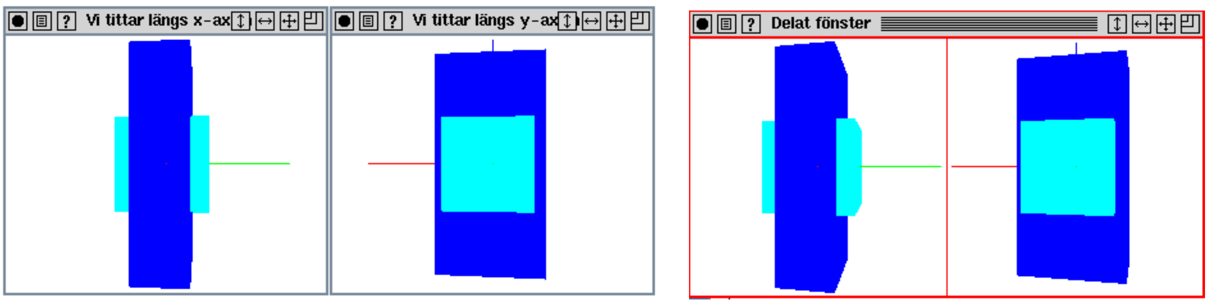
Mera kanske senare. MUI ger möjligheter till mycket bättre användarsnitt än GLUT. Vi nöjer oss här med att visa hur det ser ut i två program, *mui_test* respective *calc*. För övrigt hänvisar vi till inkluderingsfilen *mui.h*.





21 Flera fönster med GLUT

Ibland vill man samtidigt se på en scen från mer än ett håll. Detta är t ex vanligt i modelleringsprogram. Man skulle då kunna rita två olika bilder i ett fönster. Men mera naturligt är nog att ha flera fönster. I figuren nedan visas de två varianter som GLUT erbjuder. Vi har ett objekt sammansatt av två rätblock med dimensionerna 1x0.5x2 (mörkt) och 0.8x0.8x0.8 (ljusare). I vänstra figurens vänstra fönster ser vi längs x-axeln och i det högra längs y-axeln. z-axeln pekar uppåt. I högra figuren har vi till synes ett enda fönster med motsvarande vyer, men i själva verket innehåller det fönstret två underfönster. Kodningen för de två fallen är inte identisk men rätt likartad. Att figurerna inte är alldeles perfekta beror på att objektet snurrar kring z-axeln och att jag fångat rörelsen i flykten.



För att inte denna skrift skall explodera volymmässigt har jag våren 2002 placerat resten av detta grafiskt sett mindre viktiga avsnitt separat. Den intresserade kan hämta det via kurssidan. Exemplen 13 och 14 finns där.

22 Belysning

Nu skall vi ta hänsyn till ljuskällor i vår modell. Detta fungerar på ett vettigt sätt bara om vi har RGBA-mod. OpenGL använder i stort sett den belysningsformel vi diskuterat i kursen. Dvs den intensitet som användaren upplever i en viss punkt bestäms som en summa av omgivningsljus, diffust reflekterat och spegelreflekterat ljus.

I OpenGL Programming Guide står det "*The OpenGL lighting equations are just an approximation, but one that works fairly well and can be computed relatively quickly*". Ingenstans står det vad detta innebär. I praktiken betyder det att belysningsformlerna är approximativa, vilket vi kan stå ut med. Men värre är att OpenGL beräknar belysningen lokalt. OpenGL tar ingen som helst hänsyn till att ett objekt kan hindra ljuset från att nå en annat! Om ljusriktningen sammanfaller med synriktningen, blir resultatet någorlunda, men vid sidobelysning blir resultatet ofta grovt felaktigt.

Låt oss nu se vad som behöver göras (vi kommer här att ignorera att en hel del standardinställningar finns, speciellt som de varierar) för att få det hela att fungera.

- Ljuskällans (ornas) egenskaper måste anges. Man gör detta typiskt i en rutin som anropas innan vi hoppar in i händelsesnurran.
- Materialegenskaperna anges för en eller flera polygoner. Dessa fungerar som **ersättare** för våra färger (satta med rutinen `glColor3f`) och beter sig på likartat sätt, dvs en egenskap gäller till dess att den ändras.
- Normaler i hörnen. Gäller också tills vidare.
- Övrigt

22.1 Ljuskällorna

Vi sätter ljuskällans färg och position med ett anrop av

```
glLightfv(ljuskälla, egenskap, värde i form av en 4-vektor)
```

Ljuskällorna, som kan vara åtminstone åtta till antalet, betecknas `GL_LIGHT0` t o m `GL_LIGHT7`. Egenskaperna med `GL_DIFFUSE`, `GL_SPECULAR`, `GL_AMBIENT` och `GL_POSITION`. Man låter varje ljuskällas ljus sammansättas av en diffus del, som kan reflekteras diffust, och en "spekulär" del, som kan reflekteras koncentrerat. Oftast kan man väl låta dessa delar ha samma innehåll (i kursboken skiljer man dem inte åt). Vidare kan varje ljuskälla ge ett bidrag till det "ambienta" ljuset (omgivningsljuset).

Exempel 15:

```
GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_DIFFUSE, specular);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
```

Man kan inte ange siffervärden direkt i anropet. ❋

När det gäller positionsangivelser så betyder ett värde (x,y,z,0.0) att (x,y,z) är riktningen till ljuskällan och att denna ligger långt bort. Är fjärde koordinaten större än 0 anger (x,y,z) i stället absoluta positionskoordinater i världskoordinatsystemet. Ljuskällorna beter sig som vanliga geometriska objekt och utsätts för de geometriska transformationer (modell-vy-matrisen; projektionsmatrisen påverkar dem inte) som gäller när man med `glLightfv(...,GL_POSITION, ...)` anger positionen. En statisk ljuskälla placeras därför lämpligen direkt efter anropet av `gluLookAt`.

22.2 Material

Tidigare har vi kunnat ange en färg för varje hörn med `glColor3f`. När belysningsalgoritmerna är igång har den proceduren inte någon effekt. I stället anger vi för varje hörn ange en utåtriktad normal med `glNormal3f`. Utifrån belysningsmodellen och bl a materialegenskaperna räknar OpenGL ut färgen. Polygoner karakteriseras ju av en enda normal, men när sådana används för approximation av krökta objekt är det ofta lämpligt att låta hörnnormalerna vara medelvärden av normalerna för de polygoner som möts i hörnet. Det är nog bra att se till att normalerna är normerade (annars bör man slå på `glEnable(GL_NORMALIZE)`).

De rena materialegenskaperna sätts med
`glMaterialfv(`*`vilka polygonsidor som avses`*`, `*`egenskap`*`, `*`värde i form av en 4-vektor`*`).`
Den första parametern kan vara `GL_FRONT`, `GL_BACK` eller `GL_FRONT_AND_BACK`. Oftast har ju utsidan och frånsidan av en polygon (tänk på en husvägg) helt olika egenskaper. Egenskaperna kan vara `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, och `GL_EMISSION`. Om egenskapen är `GL_EMISSION` avses ett strålande objekt och då har övriga storheter ingen betydelse.

Exempel 16: `GLfloat mat_diffuse[] = { 0.0, 1.0, 0.0, 1.0 };`

```
                ....
        glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
```

betyder att följande polygoners framsidor bara reflekterar grönt ljus diffust, dvs de ser gröna ut. Talen anger reflektionsgrad. ❋

Exponenten i spegelreflektionstermen sätter man med
`glMaterialf(`*`vilka polygonsidor som avses`*`, GL_SHININESS, `*`reellt tal`*`).`

22.3 Övrigt

Belysning måste aktiveras allmänt (med `glEnable(GL_LIGHTING)`) och per ljuskälla med t ex `glEnable(GL_LIGHT0)`. Om man tillfälligt vill rita något med färgen bestämd av `glColor3f` kan man slå av belysningsberäkningarna med `glDisable(GL_LIGHTING)`.

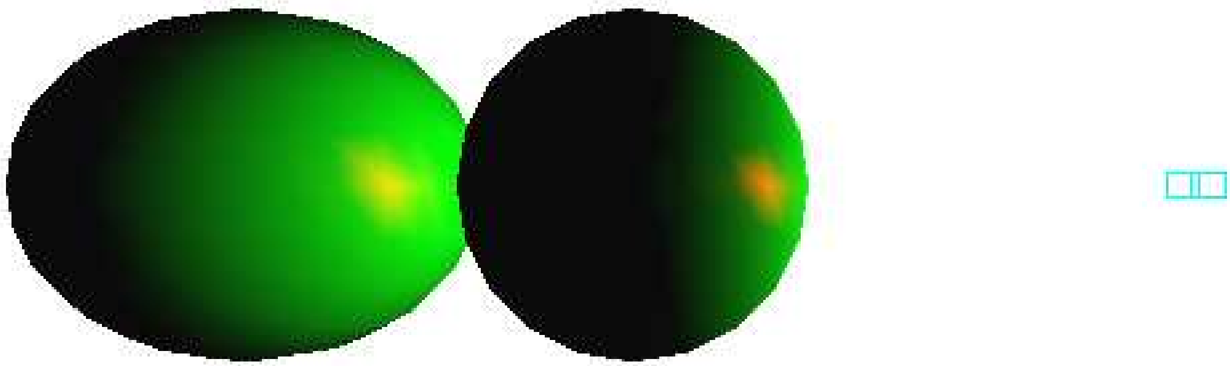
Val av målningsmodell görs med `glShadeModel(`*`modell`*`)`, där *modell* kan vara `GL_FLAT` (konstant målning över polygonen) respektive `GL_SMOOTH` (interpolerad målning enligt Gouraud).

För att få belysningen på både fram- och baksida rätt beräknad måste man göra anropet
`glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE).`
Detta är aktuellt bara då man har öppningar i objekt eller tänkes kunna snitta dem.

Omgivningsljuset kan definieras oberoende av övriga ljuskällor, men vi går inte in på detaljerna.

22.4 Ett exempel

Exempel 17: Följande figur visar ett enhetsklot med mittpunkt i origo och ett annat med mittpunkt i -2 på den negativa x-axeln. Vi tittar i negativ z-riktning med y-axeln uppåtriktad. Kloten belyses av en ljuskälla indikerad av den lilla kuben till höger. I figuren är kuben mycket nära den positiva x-axeln och vi ser tydligt bristen i Open GL när det gäller sidobelysning. Programmet som följer låter användaren snurra på ljuskällan runt y-axeln (med tangenterna 1 och 2). Vidare kan betraktarens position på z-axeln regleras med tangenterna n och m. Reflektionsexponenten kan ändras med tangenterna k och l. Programmet finns som `GL_ANIM.c`.



```
/* GL_ANIM.c något redigerad; finns även i en utbyggd version GL_LJUS.c */
#include <GL/glut.h>
#include <stdlib.h>
// Tre globala storheter som kan ändras interaktivt med det halvdana användar-
// snittet
GLfloat Dist = 3.0, Konc = 50, VinkY = 0;
//Position för ljuskällan
GLfloat light_position[] = { 0.0, 0.0, 2.5, 1.0 };

/*  Permanenta inställningar sätts i init */
void init(void) {
   // Materialegenskaper
   // Bara rött spegelreflekteras, bara grönt diffusreflekteras
   GLfloat mat_specular[] = { 1.0, 0.0, 0.0, 1.0 };
   GLfloat mat_diffuse[] = { 0.0, 1.0, 0.0, 1.0 };
   // Bara blått diffusreflekteras på baksidan
   GLfloat mat_diffuse_back[] = { 0.0, 0.0, 1.0, 1.0 };
   // Ljuskälleegenskaper: vitt spekulärt och diffust ljus
   GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
   glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, specular);
   // Materialegenskaper sätts (uppdelning p g a C:s regler om deklarationer)
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_BACK, GL_DIFFUSE, mat_diffuse_back);
   // Se till att fram- och baksidor ljussätts "korrekt"
   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
   glLightModeli(GL_LIGHT_MODEL_TWO_SIDE,GL_TRUE);
   // Hänsyn till betraktarens position tas. Onödig, ty kostar.
   glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,GL_TRUE);
   glEnable(GL_LIGHTING); glEnable(GL_LIGHT0);
   glClearColor (1.0, 1.0, 1.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glEnable(GL_DEPTH_TEST);
}
```

```
void display(void) {
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();  // Push-Pop så att vi inte får en kumulativ effekt
      gluLookAt(0,0,Dist,0,0,0, 0,1,0); // I display!!!
      // Rotera ljuskällan
      glPushMatrix();
         glRotatef(VinkY,0,1,0);
         glLightfv(GL_LIGHT0, GL_POSITION, light_position);
         // Rita ljuskällan som en grön-blå box med avslagen belysning
         glTranslated (0.0, 0.0, light_position[2]);
         glDisable (GL_LIGHTING);
         glColor3f (0.0, 1.0, 1.0);
         glutWireCube (0.1);
         glEnable (GL_LIGHTING);
      glPopMatrix();
      glMaterialf(GL_FRONT, GL_SHININESS, Konc);
      // En sfär (normalerna beräknas internt i proceduren)
      glutSolidSphere (1.0, 20, 16);
      // En till till vänster om den första
      glTranslatef(-2,0,0);
      glutSolidSphere (1.0, 20, 16);
      glutSwapBuffers();
   glPopMatrix();
}

void reshape (int w, int h) {
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(120, 1, 0.1,100);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y) {
   switch (key) {
      case 'q': exit(0); break;
      case 'm' : Dist = Dist + 0.2;
         display();  // Borde genomgående vara glutPostRedisplay() men labuppg
         break;
      case 'n' : Dist = Dist - 0.2; display(); break;
      case 'k' : Konc = Konc-2; display(); break;
      case 'l' : Konc = Konc+2; display(); break;
      case '1' : VinkY=VinkY-2;  display(); break;
      case '2' : VinkY = VinkY+2; display(); break;
   }
}
```

```
int main(int argc, char** argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);// Programnamnet blir fönstertitel
  init();
  glutDisplayFunc(display);glutReshapeFunc(reshape);glutKeyboardFunc(keyboard);
  glutMainLoop();
}
```

Det blev ett otäckt långt kodstycke, så jag måste avstå från kommentarer här. ❋

23 Texturer

En textur i OpenGL är ett rutnät med MxM punkter, där M är en 2-potens, dvs $M = 2^N$. Rutnätet innehåller typiskt RGB-information med en byte per färgkomponent. En variabel för en sådan textur deklareras enligt

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

Typen *GLubyte* står för "unsigned byte" (motsvarande C:s `unsigned char`) och innebär att innehållet tolkas som heltal mellan 0 och 255. Tidigare (i samband med rasterkopiering) mötte vi nog *GLbyte* som innebär att innehållet tolkas som heltal mellan -128 och 127. Det hade hade varit naturligare att även då använda *GLubyte*, åtminstone om vi skulle stoppa in värden själva i vektorn.

Vi måste naturligtvis ge texturen innehåll. Normalt sker det väl genom att man läser från en fil (inga formatkonverteringsredskap finns dock i OpenGL), men vi kan också göra det direkt i programmet. Följande rader gör t ex att nedre vänstra fjärdedelen i texturen *Image* blir röd.

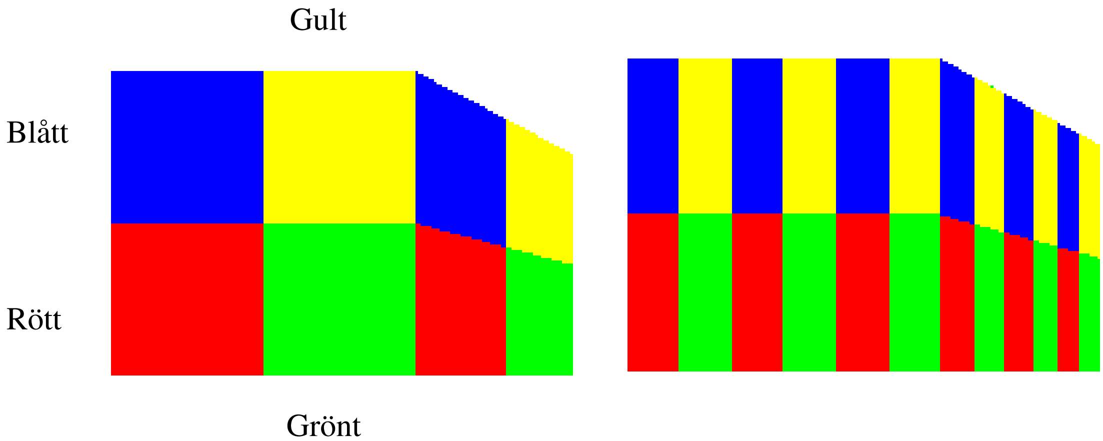
```
for (i = 0; i < TexHeight/2; i++) {
      for (j = 0; j < TexWidth/2; j++) {
         Image[i][j][0] = (GLubyte) 255;
         Image[i][j][1] = (GLubyte) 0; Image[i][j][2] = (GLubyte) 0;
      }
}
```

Innan vi kan använda texturen för ritning måste vi lägga in den i ett texturregister och komplettera den med viss information. För texturregistret använder vi en vektor `GLint texName[ANTAL]` , som rymmer `ANTAL` texturer. Denna initieras med `glGenTextures(ANTAL, texName)`. Vi gör en textur aktuell med ett anrop av `glBindTexture`. När en textur väl är aktuell kan vi lägga in information i den eller rita med den.

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1); // Ev standard
glGenTextures(2, texName);
glBindTexture(GL_TEXTURE_2D, texName[0]);
// Filtreringsegenskaper vid förstoring (MAG) respektive förminskning (MIN),
// GL_LINEAR blir bättre än GL_NEAREST
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

```
// GL_DECAL betyder att texturen skriver över
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
// Koppla mönstret i Image till texturen
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, TexWidth,TexHeight, 0,
             GL_RGB, GL_UNSIGNED_BYTE,Image);
// Ev ytterligare textur
glBindTexture(GL_TEXTURE_2D, texName[1]);
...
glEnable(GL_TEXTURE_2D);
```

Exempel 18: Till vänster i följande figur visas två plank med pålagd textur. Det ena planket är vinkelrätt mot betraktningsrikningen och har nedre högra hörnet i origo (0,0,0). Användaren står på positiva z-axeln och tittar mot origo. Det andra planket är vinklat 45 grader moturs kring y-axeln. Vi ser att texturpåläggningen görs perspektivistiskt korrekt (perspektiv är påslaget med `gluPerspective`). I den högra bilden har vi sett till att texturen upprepas i ena riktningen men inte i den andra.



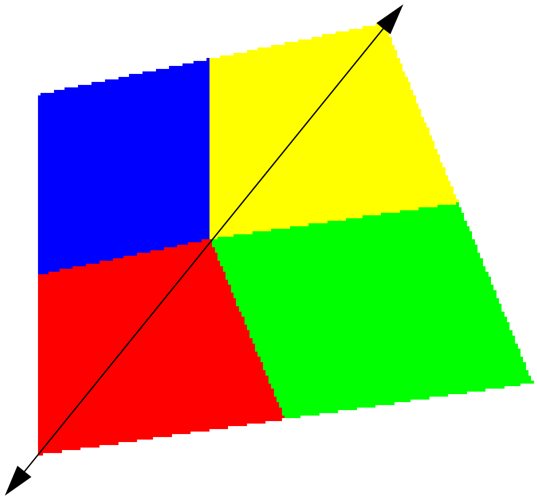
Uppritningskoden för bilden följer. D = 1.0 ger vänstra figuren och D=3.0 den högra. Till varje hörn ger vi en texturkoordinat. Ett värde större än 1.0 innebär att texturen upprepas på nytt (om ej annorlunda avtalats med OpenGL). Programmet finns i sin helhet som `GL_TEXTUR.c`.

```
void fig() {
   glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0, 0.0);
      glTexCoord2f(D, 0.0);   glVertex3f(1.0, 0.0, 0.0);
      glTexCoord2f(D, 1.0);   glVertex3f(1.0, 1.0, 0.0);
      glTexCoord2f(0.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
   glEnd();
}

void display(void) {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(0,0,my_z, 0,0,0, 0,1,0);
   glBindTexture(GL_TEXTURE_2D, texName[0]);
   glPushMatrix();
       glTranslatef(-2,0.0,0.0);glScalef(2,2,1); fig();
   glPopMatrix();
   glPushMatrix();
       glRotatef(45,0,1,0); glScalef(2,2,1); fig();
   glPopMatrix();
```

```
   glutSwapBuffers();
}
```
✻

Som bekant blir det distortioner i texturpåläggningen om man försöker göra något omöjligt. I följande figur har vi försökt att lägga den tidigare texturen på en fyrhörning i xy-planet och tittar på den ortografiskt. Linjär avbildning är som vi vet omöjlig. Den ena diagonalen i texturen bryts också mycket riktigt. Det förefaller som om OpenGL delat fyrhörningen i två trianglar (längs den manuellt markerade diagonalen) och gjort en linjär avbildning av texturen på var och en av dessa.



24 Kurvor och ytor i OpenGL

I OpenGL finns stöd för ritning av **enstaka** kurv- eller ytsegment (eng patch) av Beziertyp med godtyckligt gradtal. I GLU finns stöd för allmänna kurvor och ytor av NURBS-typ.

24.1 Bezierkurvor

Detta avsnitt kan nog förbigås, även om det möjligen underlättar förståelsen av avsnitten om NURBS-objekt.

OpenGL använder s k **evaluerare** för beskrivning och generering av kurvor (och ytor) av Beziertyp. Evaluerare kan implementeras i hårdvara. Vill man arbeta med någon annan typ av splines, får man själv svara för konvertering till Beziertyp.

En evaluerare för generering av punkter längs en kubisk Bezierkurva skapas och aktiveras med

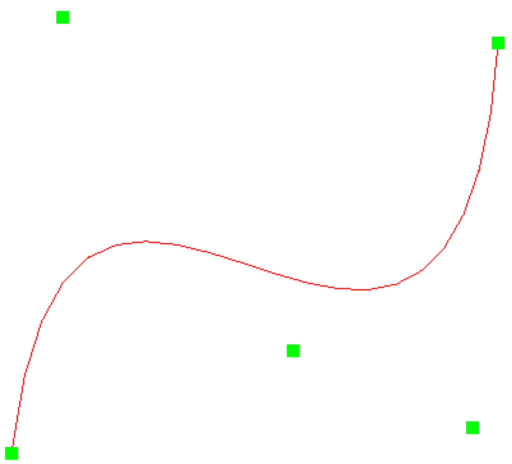
```
glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,Hopp,4,adr till vektor med punkterna):
glEnable(GL_MAP1_VERTEX_3);
```

Konstanten `GL_MAP1_VERTEX_3` säger att det endimensionella parameterintervallet skall avbildas på (ge upphov till) 3-dimensionella punkter (x,y,z). De två följande talen anger parameterintervallet. Talet 4 är gradtalet ökat med 1 (man kan alltså använda godtyckligt gradtal) eller om du så vill antalet punkter. Vektorn kan typiskt vara `GLfloat CtrlPoints[4][D]`, där D är 2 eller 3 beroende på om vi anger ankar- och styrpunkterna som (x,y) eller (x,y,z). Det verkar dock som om man i praktiken måste ange dem på formen (x,y,z), dvs D är 3. Parametern *Hopp* anger avståndet mellan två sådana punkter i vektorn uttryckt i *GLfloats*, dvs är vårt D.

Själva punktgenereringen görs med ett anrop `glEvalCoord1f(t)`, där t är aktuellt parametervärde. Varje sådant anrop motsvarar att vi hade skrivit `glVertex3f(x(t),y(t),z(t))`, där x(t) är den funna approximationen i x-led, etc.

Det finns ett otal tänkbara evaluerare. T ex gör `GL_MAP1_COLOR4` att evalueraren ger ifrån sig `glColor4f(Red(t),Green(t),Blue(t),Alpha(t))`.

Exempel 19: Följande bild på en Bezierkurva av gradtalet 4 (som omväxling) ritades med program-



delarna (hela programmet finns som `GL_BEZIER.c`) nedan

```
GLfloat CtrlPoints[5][3] = {
        {0.1, 0.1, 0.0}, {0.3, 1.8, 0.0}, {1.2, 0.5, 0.0},
        {1.9, 0.2, 0.0}, {2.0, 1.7, 0.0}};
void init(void) {
   glClearColor(1.0, 1.0, 1.0, 0.0);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 5, &CtrlPoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}
void display(void) {
   int i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 20; i++)
         glEvalCoord1f((GLfloat) i/20.0);
   glEnd();
   // Rita styrpunkterna
   glPointSize(6.0);
   glColor3f(0.0, 1.0, 0.0);
   glBegin(GL_POINTS);
      for (i = 0; i < 5; i++)
         glVertex3fv(&CtrlPoints[i][0]);
   glEnd();
   glFlush();
}
```

❋

24.2 NURBS-kurvor

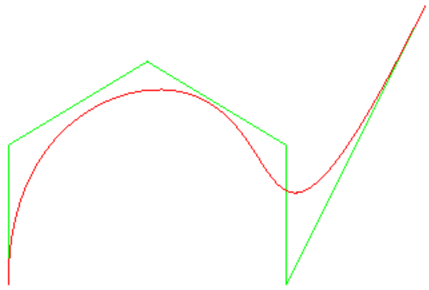
Detta och nästa avsnitt förutsätter att du har skaffat dig en grundläggande förståelse för NURBS på annat håll (t ex föreläsningsmaterial). Detaljer om parametrar etc hittar du i de (delvis svårbegripliga) manualblad som finns för de olika procedurerna.

NURBS-kurvor hanteras av GLU, som gör "tessellering" anpassad för OpenGL:s evaluerare. Nedan producerar vi bara specialfallet kubiska B-splines.

Vi anger styrpunkterna och skarvarna. Antalet skarvar (eng. knots) skall vara 4 högre än antalet styrpunkter. För interpolering i ändpunkterna skall de fyra första skarvarna vara lika liksom de fyra sista.

I ett program som arbetar med NURBS-kurvor eller ytor behöver man ett enda NURBS-objekt, som kan uppfattas som ett dynamiskt arbetsutrymme för OpenGL:s interna arbete. NURBS-objektet skapas genom att man deklarerar en variabel (t ex `theNurb`) som en pekare till typen `GLUnurbsObj` och sedan gör anropet `theNurb = gluNewNurbsRenderer();`

Exempel 20: Kubiska B-Splines-kurvan till de sex styrpunkterna i figuren. Även styrpolygonen utritad.



Programmet finns som `GL_BSPLINES.c` och låter oss dra i och flytta styrpunkter. Vi redovisar bara kurvbitarna här.

Globalt

```
// 6 styrpunkter
GLfloat CtrlPoints[6][3] =
     {{0.25, 0.5, 0.0}, {0.25, 0.75, 0.0},{0.5,0.9, 0.0}, {0.75, 0.75, 0.0},
       {0.75, 0.5, 0.0}, {1.0,1.0, 0.0}};

// Och därmed 10 skarvar (i kubiska fallet), varav de 4 första och de 4 sista
// skall vara lika inbördes
GLfloat Knots[10] = {0.0, 0.0, 0.0, 0.0, 1.0, 2.0,
                     3.0, 3.0, 3.0, 3.0};
GLUnurbsObj *theNurb;
theNurb = gluNewNurbsRenderer(); // Ett NURBS-objekt skapas
```

Uppdateringsproceduren

```
glColor3f(0,1,0);
// Rita styrpolygonen i grönt
glBegin(GL_LINE_STRIP);
   for (i=0; i<6; i++) {
```

```
        glVertex3fv(curvePt[i]);
     }
glEnd();
// Rita kurvan i rött. gluBeginCurve/gluEndCurve i st f glBegin.
glColor3f(1.0,0.0,0.0);
gluBeginCurve(theNurb);
   // gluNurbsCurve i st f glVertex
   gluNurbsCurve (theNurb, 10, Knots, 3,
          &CtrlPoints[0][0], 4, GL_MAP1_VERTEX_3);
gluEndCurve(theNurb);
```

Proceduren `gluNurbsCurve` har huvudet

```
void gluNurbsCurve (GLUnurbs* nurb, GLint knotCount, GLfloat *knots,
GLint stride, GLfloat *control, GLint order, GLenum type);
```

och i vårt anrop

```
gluNurbsCurve(theNurb,10,Knots,3,&CtrlPoints[0][0],4,GL_MAP1_VERTEX_3);
```

anger följaktligen 10 antalet skarvar (antalet styrpunkter+4), Knots och &CtrlPoints[0][0] adresserna till skarvvektorn respektive styrpunktsvektorn, 3 utrymmet i *GLfloats* mellan två styrpunkter i vektorn, 4 ordningen (gradtalet+1) och slutligen `GL_MAP1_VERTEX_3` att parametervärdena skall generera tredimensionella punkter.

Det är lätt att lägga till samtidig generering av färg (motsvarande `glColor4f`) och texturkoordinater (motsvarande `glTexCoord1f`) genom att införa punktvektorer för dessa och sedan bara göra extra anrop av `gluNurbsCurve` före det tidigare (med sista parametern `GL_MAP1_COLOR_4` respektive `GL_MAP1_TEXTURE_COORD_1`). Även normaler klaras så. Nu är ju dock texturer och normaler inte så intressanta för kurvor, men samma teknik duger för ytor.

Exempelvis kan vi åstadkomma färgväxling längs kurvan i vårt exempel genom att införa vektorn

```
GLfloat curveCol[6][4] = {{1,0,0,0}, {0,0,1,0},{0,1,0,0}, {1,1,0,0},
                          {0,1,1,0},{0,1,0,0}};
```

och före det tidigare anropet av `gluNurbsCurve` lägga

```
gluNurbsCurve(theNurb,10,curveKnots,4,&curvCol[0][0],4,GL_MAP1_COLOR_4);
```

Kurvan kommer då att börja i rött och sluta i grönt.

Det kan inträffa en del fel i samband med arbetet med NURBS. För att få någon form av vettig felutskrift kan man registrera en callback-rutin för NURBS-objektet. Detta görs direkt efter det att skapats på följande sätt:

```
gluNurbsCallback(theNurb, GLU_ERROR, nurbsError);
```

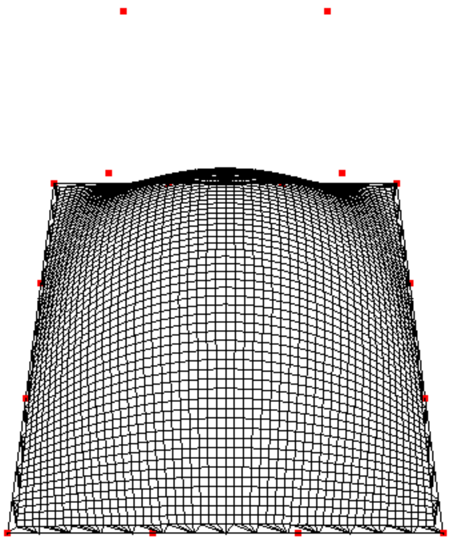
Vid fel kommer proceduren `nurbsError`, som vi måste skriva själva, att anropas. Den kan se ut så här och gör om en felkod till en upplysande sträng och skriver ut den:

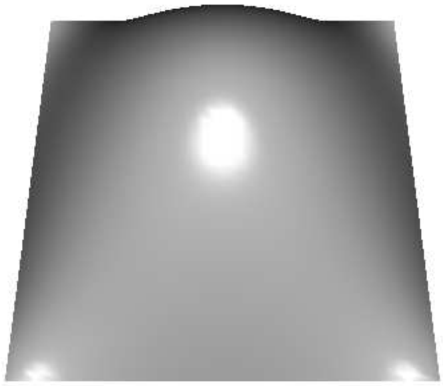
```
void nurbsError(GLenum errorCode) {
   printf ("Nurbs Error: %s\n", gluErrorString(errorCode));
   exit (0);
}
```

När det gäller allmänna NURBS räcker det att införa styrpunkter med fyra koordinater och senare anropa `gluNurbsCurve` med GL_MAP1_VERTEX_4.

24.3 NURBS-ytor

Exempel 21: Ett exempel (`surface.c`) hämtat från röda boken.





Globalt

```
GLfloat ctlpoints[4][4][3];
void init_surface(void){
   int u, v;
   for (u = 0; u < 4; u++) {
      for (v = 0; v < 4; v++) {
         ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
         ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);
         if ( (u == 1 || u == 2) && (v == 1 || v == 2))
            ctlpoints[u][v][2] = 3.0;
         else ctlpoints[u][v][2] = -3.0;
      }
   }
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
   // Högra bilden
   // gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
   // Vänstra
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_OUTLINE_POLYGON);
   gluNurbsCallback(theNurb, GLU_ERROR, nurbsError);
}
```

I uppdateringsproceduren

```
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb,
                   8, knots, 8, knots,
                   4 * 3, 3, &ctlpoints[0][0][0],
                   4, 4, GL_MAP2_VERTEX_3);
   gluEndSurface(theNurb);
```

25 Fördröjd ritning

I de flesta grafiksystem kan man arbeta i dels en **omedelbar** (eng. immediate) mod, dels en **fördröjd** (eng. retained) mod. Hittills har vi bara ägnat oss åt den förra. Den fördröjda moden kan ha många olika innebörder, men den allmänna principen är att man bygger upp en lista av grafikkommandon, som man senare kan få utförda genom att referera till listan. Avsikten är att öka effektiviteten. En sådan lista kan i viss mån kompileras och vidare kan i en klient-server-miljö (som OpenGL ju är) listan och dess data lagras på serversidan, vilket ger vinst åtminstone om samma lista anropas ofta. Speciellt kan texturer lagras i grafikminnet. Det är dock svårt att i förväg bedöma i vilka situationer som vinst garanterat uppnås.

I OpenGL används **"display"-listor** för fördröjd mod. Man refererar till en lista med ett positivt heltal (!), som man förstås kan namnge med `#define` eller `enum`. Det finns redskap som hindrar en från att oavsiktligt använda samma heltal två gånger men det tar vi inte upp.

En lista konstrueras (kan ligga i en initieringsrutin) enligt modellen

```
glNewList(23, GL_COMPILE);
     // Följd av vanliga grafikkommandon, t ex
     glTranslatef(...); glRotate(...);
     glBegin(GL_POLYGON);
       for (i=0; i<10; i++) {
           glColor3fv(Col[i]);
           glVertex3fv(Point[i]);
       }
     glEnd();
glEndList();
```

En lista anropas enligt modellen

```
glRotatef(...);
glCallList(23);    // Om listan har numret 23
```

En lista kan anropa andra listor, vilket kan användas för att bygga upp en hierarkisk struktur. Listorna kan inte styras med parametrar eller genom ändring av globala storheter, utan de attribut som anges vid uppbyggnaden av listan gäller även vid uppritandet. I ovanstående exempel är det alltså ingen idé att ändra i vektorn `Point`. Däremot utsätts alla geometriska objekt för kombinationen av de transformationer som anges i definitionen och de som anges vid uppritandet. En optimering som görs i listan i vårt exempel är att de två geometriska transformationerna (eller snarare deras matriser) slås ihop till en enda en gång för alla vid listans skapande.

26 Selektering

Hur pekar vi ut objekt i 3D? OpenGL erbjuder två olika metoder.

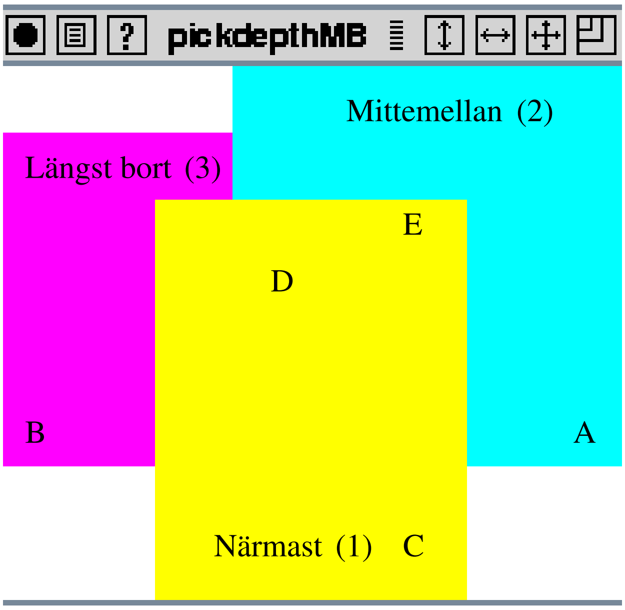
26.1 Med gluUnproject

Kommer i ett eget avsnitt längre fram.

26.2 Med selekteringsmod

Följande exempel har jag - i stort sett - lånat från en OpenGL-bok.

Exempel 20: Vi betraktar tre rektanglar som ligger på olika avstånd från betraktaren och delvis



överlappar i xy-led. Låt oss först se på en körning (programmet finns som `GL_PICKDEPTH_MB`) där jag klickat på i tur och ordning punkterna A, D och E. För varje klick får vi reda på antalet överlappande objekt och sedan för varje objekt minsta och största z-avstånd (i det normaliserade koordinatsystemet) samt objektets namn i form av ett tal.

```
> GL_PICKDEPTH_MB
Antal träffar = 1
  zlow: 1; zhigh: 1 Objektets namn: 2
Antal träffar = 3
  zlow: 0.333333; zhigh: 0.333333 Objektets namn: 1
  zlow: 1; zhigh: 1 Objektets namn: 2
  zlow: 1.66667; zhigh: 1.66667 Objektets namn: 3
Antal träffar = 2
  zlow: 0.333333; zhigh: 0.333333 Objektets namn: 1
  zlow: 1; zhigh: 1 Objektets namn: 2
```

Uppritningen av objekten görs på vanligt sätt, men vi parametriserar motsvarande procedur så att den dels kan anropas i normal ritningsmod (`GL_RENDER`), dels i selekteringsmod (`GL_SELECT`). I det senare fallet sker ingen ritning, men objekten förknippas med namn i form av heltal.

```
void drawRects(GLenum mode){
   if (mode == GL_SELECT) glLoadName(1);
   glBegin(GL_QUADS);
      glColor3f(1.0, 1.0, 0.0);
      glVertex3i(2, 0, 0); glVertex3i(2, 6, 0);
      glVertex3i(6, 6, 0); glVertex3i(6, 0, 0);
   glEnd();
   if (mode == GL_SELECT) glLoadName(2);
   ...
}
```

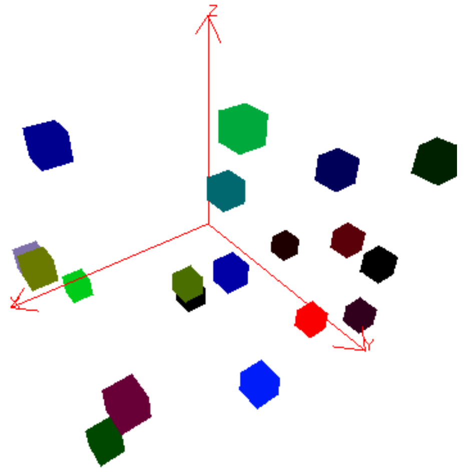
Uppdateringsproceduren anropar denna med `drawRects(GL_RENDER)`. När val via musproceduren sker går man in i följande procedur, där vi först med `gluPickMatrix` "begränsar synfältet" till en 5x5-kvadrat runt muspositionen. Detta resulterar i en matris som multipliceras med den vanliga projektionsmatrisen.

```
#define BUFSIZE 512
void pickRects(int button, int state, int x, int y){
   GLuint selectBuf[BUFSIZE];
   GLint hits;
   GLint viewport[4];
   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN) return;
   glGetIntegerv(GL_VIEWPORT, viewport);
   glSelectBuffer(BUFSIZE, selectBuf);
   glRenderMode(GL_SELECT);       // selekteringsmod
   glInitNames();                 // teknisk detalj
   glPushName(0);                 // ännu mera teknisk detalj
   glMatrixMode(GL_PROJECTION);
   glPushMatrix();                 // spara normal projektionsmatris
      glLoadIdentity();
      //create 5x5 pixel picking region near cursor
      gluPickMatrix((GLdouble) x,
                  (GLdouble) (viewport[3] - y),
                  5.0, 5.0, viewport);
      glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);   // samma som vid vanlig ritning
      // modellvy-matrisen satt till enhetsmatrisen i reshape,dvs behöver ej anges
      drawRects(GL_SELECT);
   glPopMatrix();                   // återställ normal projektionsmatris
   glFlush();
   hits = glRenderMode(GL_RENDER);  // återställ till normal ritningsmod
   processHits(hits, selectBuf);
}
```

I slutet av proceduren återställer vi till normal ritmod med `hits=glRenderMode(GL_RENDER)` och får då reda på antalet träffar. Informationen om träffarna har samlats i en buffert som vi bearbetar med följande procedur. Den är något överarbetad för vår situation och jag avstår från kommentarer.

```
void processHits(GLint hits, GLuint buffer[]) {
   unsigned int i, j;
   GLuint names, *ptr;
   printf("Antal träffar = %d\n", hits);
   ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) {  // För varje träff
      names = *ptr; ptr++;
      printf("  zlow: %g;", (float) *ptr/0x7fffffff); ptr++;
      printf(" zhigh: %g", (float) *ptr/0x7fffffff); ptr++;
      printf(" Objektets namn: ");
      for (j = 0; j < names; j++) {  // För varje namn
         printf("%d ", *ptr); ptr++;
      }
      printf("\n");
   }
}
```

Exempel 23: Ett roligare och mera praktiskt exempel är programmet `GL_VALKUBER.c`. Här klickar man med musen på en kub och får reda på numret på den och eventuella bakomliggande.



27 Stereo

Exempel 24: Situationen i figuren ovan lämpar sig utmärkt för stereo, så låt oss titta litet på ett program för det (det finns som `GL_STEREO.c`). Ändringarna är rätt obetydliga. I huvudrutinen `main` lägger vi till moden `GLUT_STERO`:

```
    glutInitDisplayMode(GLUT_STEREO | GLUT_DOUBLE | GLUT_RGB |
                        GLUT_DEPTH);
```

I proceduren för storleksförändring av fönstret tar vi inte med observatörens placering.

```
void myReshape(int width, int height){
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(55.0,((GLfloat)width)/((GLfloat)height),1.0,5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

I omritningsproceduren ritar vi två gånger, ena gången en bild för vänstra ögat, andra gången en bild för högra.

```
void display(void){
    glPushMatrix();                    // Spara modell-vy-matris = enhetsmatris
       glClearColor(1.0,1.0,1.0,1.0);
       glDrawBuffer(GL_BACK);
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
       // Vänstra ögats bild
       gluLookAt(-Eye,0.0,Position,0.0,0.0,0.0,0.0,1.0,0.0);
       glRotatef(VinkX,1.0,0.0,0.0);     // Scenen kan vridas med piltangenterna
       glRotatef(VinkZ,0.0,0.0,1.0);
       glDrawBuffer(GL_BACK_LEFT);       // Välj var ritning skall ske
       myFigure();
    glPopMatrix();                     // Hämta modell-vy-matris = enhetsmatris
```

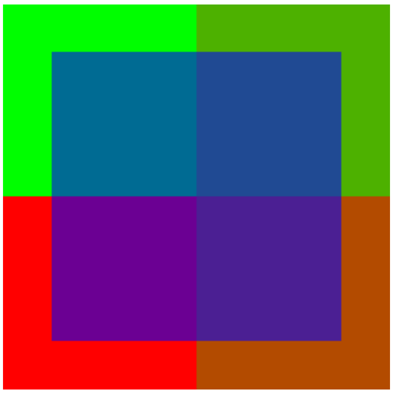
```
    // Högra ögats bild
    glPushMatrix();
       gluLookAt(Eye,0.0,Position,0.0,0.0,0.0,0.0,1.0,0.0);
       glRotatef(VinkX,1.0,0.0,0.0);
       glRotatef(VinkZ,0.0,0.0,1.0);
       glDrawBuffer(GL_BACK_RIGHT);
       myFigure();
    glPopMatrix();
    glutSwapBuffers();
}
```

Jag satte `Eye` (halva ögonavståndet) till 0.15 i programmet, men det lämpliga värdet beror på den aktuella scenen, fönsterstorleken och avståndet till skärmen. I programmet finns möjligheten att med tangenterna 3 och 4 ändra det. Vi tittar mot samma punkt i båda fallen (detta är inte alldeles korrekt stereoseende, men ger ett rätt acceptabelt resultat).

28 Färgblandning

Vi har sett att färger kan anges med fyra storheter, den fjärde brukar kallas alfa-värdet. Mer avancerade system än våra har alfaplan, dvs en eller flera bitar per bildpunkt i vilka alfavärdet lagras. Alfavärdet kan bl a användas för färgblandning, t ex för polygoner som släpper igenom en del av färgen från tidigare ritade ytor.

Exempel 25: I figuren ser vi en delvis genomskinlig mittruta, som släpper igenom delar av de fyra mindre rutorna (torde framgå dåligt i svartvitt, men programmet finns som `GL_ALPHA.c`). Dessa fyra kan anses vara ritade med vissa färger, men i praktiken är de åstadkomna genom att rött och grönt blandats i olika proportioner).



Färgblandning måste aktiveras och dessutom måste vi bestämma en regel för hur blandningen skall gå till. Detta kan t ex göras i en initieringsprocedur

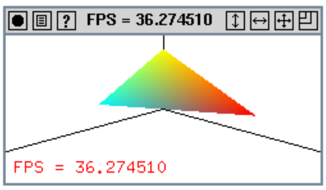
```
void init(void) {
   glEnable (GL_BLEND);                 // Aktivering
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  // Komm nedan
   glShadeModel (GL_FLAT);
   glClearColor (1.0, 0.0, 0.0, 0.0);
}
```

Med vårt anrop av `glBlendFunc` begär vi att varje färgkomponent t ex R skall bestämmas så här: R = alfa * ritfärgen + (1-alfa)*(färgen hos det redan ritade), där alfa är alfa-värdet vid ritning. Ritfärgen kan komma från `glColor` eller från en ljusberäkning. Nedan återger vi uppdateringsproceduren `display` som utnyttjar en procedur `drawSquare` för ritning av kvadrater.

```
// Ritar en enhetskvadrat med nedre vänstra hörnet i (x,y,z)
void drawSquare(float x, float y, float z){
   glPushMatrix();
      glTranslatef(x,y,z);
      glBegin (GL_QUADS);
         glVertex3f(0.0, 0.0, 0.0);
         glVertex3f(1.0, 0.0, 0.0);
         glVertex3f(1.0, 1.0, 0.0);
         glVertex3f(0.0, 1.0, 0.0);
      glEnd();
   glPopMatrix();
}


void display(void) {
   // Sudda (rött)
   glClear(GL_COLOR_BUFFER_BIT);
   // Rita över så vi får totalt fyra småkvadrater (grön färg, olika alfa)
   glColor4f(0.0,1.0,0.0,0.3); drawSquare(1,0,0);
   glColor4f(0.0,1.0,0.0,0.7); drawSquare(1,1,0);
   glColor4f(0.0,1.0,0.0,1.0); drawSquare(0,1,0);
   // Rita den genomskinliga mittkvadraten (blå ritfärg)
   glColor4f(0.0,0.0,1.0,ALPHA);  // Alfa-värdet kan ändras med tangent
   glPushMatrix();
      glTranslatef(0.25,0.25,0.0); glScalef(1.5,1.5,0.0); drawSquare(0,0,0);
   glPopMatrix();
   glFlush();
}
```

29 FPS och annan information



Ofta (speciellt under utvecklingsarbetet) kan det vara intressant att fortlöpande få reda på hur många bildrutor som produceras per sekund. På engelska blir det begreppet **FPS** (Frames Per Second). Även annan liknande information, som t ex antalet ritade polygoner per bild, kan vara av intresse. Dessa data kan naturligtvis skrivas i ett terminalfönster eller till en fil. Behagligare blir det kanske om de kommer i ett separat fönster. Det kan dock vara bra nog att skriva dem inuti i grafikfönstret eller som rubrik. Låt oss titta litet på de två sista alternativen. Antag tills vidare att den reella variabeln `fps` innehåller vad som önskas (sist i avsnittet visar vi hur FPS kan beräknas i omritningsproceduren). Vi börjar med att producera en textsträng (deklarationen `char infotext[255]` ger plats för en sträng med åtskilliga tecken) med ("utskrift mot sträng" i stället för normal utskrift)

```
sprintf(infotext, "FPS= %f", fps);
```

och ändrar sedan vid behov fönsterrubriken med

```
glutSetWindowTitle(infotext);
```

Vill vi skriva direkt i fönstret, t ex med början i position (5,15) i OpenGLs heltalskoordinatsystem, så måste vi först byta koordinatsystem och slå av eventuell belysning etc genom att anropa en procedur av typen *ortho* nedan, varpå utskrift kan ske med (se avsnitt 19)

```
char c; int j = 0;
glColor3f(1.0,0.0,0.0); glRasterPos2f( 5, 15 );
while ((c=infotext[j])!='\0') {
   glutBitmapCharacter(GLUT_BITMAP_8_BY_13,c); j = j + 1;
}
```

Därefter återgår vi till det ursprungliga koordinatsystemet med en procedur liknande *perspective* nedan.

```
void ortho(void) {
        glDisable(GL_DEPTH_TEST);
        // Slå av belysning etc
        // Spara projektionsmatrisen och ändra sedan
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        glOrtho(0, SCREEN_WIDTH, 0, SCREEN_HEIGHT, -1, 1 );
        // Spara ev modell-vymatrisen och ändra sedan
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();
}
void perspective(void) {
        // Återställ transformationsmatriserna
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();
        // Slå på ljus etc
        glEnable(GL_DEPTH_TEST);
}
```

Till sist något om beräkningen av FPS. Vi kan göra det genom att avläsa tiden i början av omritningsproceduren med

```
unsigned long time = TickCount(); // se avsnitt 17
```

och på slutet med

```
unsigned long time_per_frame = TickCount() - time;
fps = 1000.0/time_per_frame;
```

Ett stabilare värde får man genom att mäta över en längre tidsperiod, t ex en sekund eller ett antal bildrutor. Då måste vi ha globala variabler

```
int antal_bildrutor = 0;
unsigned long previous_time = 0;
```

I varje anrop av omritningsproceduren gör vi så här:

```
antal_bildrutor=antal_bildrutor+1; time = TickCount();
// Har det gått minst en sekund
if ( time - previous_time >= 1000 )   {
      fps =  antal_bildrutor * 1000.0/ (time - previous_time);
      antal_bildrutor = 0;previous_time = time;
}
```

Se till exempel programmet `GL_2002_1_TID.c` (för att få omritning rotera med r).

30 Felhantering

Så dags att komma med detta nu kanske du tycker. Men de vanligaste felen i laborationsarbetet är av annan karaktär.
OpenGL sätter ibland en felflagga och lagrar ett felvärde. Som standard ignoreras den felaktiga operationen. Vi kan läsa av genom att anropa glGetError(), som ger ett heltal. Om det är skilt från noll har ett fel inträffat. Namn för felvärdena finns definierade i gl.h:

```
#define GL_NO_ERROR                         0
#define GL_INVALID_ENUM                     0x0500
#define GL_INVALID_VALUE                    0x0501
#define GL_INVALID_OPERATION                0x0502
#define GL_STACK_OVERFLOW                   0x0503
#define GL_STACK_UNDERFLOW                  0x0504
#define GL_OUT_OF_MEMORY                     0x0505
```

Exempel:

```
glBegin(GL_LINE);  // I st f det riktiga GL_LINES
    ...
glEnd();
...
printf("FELKOD= %x\n", glGetError()); //hexadecimal utskrift
// printf("FELKOD= %s\n", gluErrorString(glGetError()));
```

ger `FELKOD= 500` (dvs felaktigt parametervärde till `glBegin`). Ersätts skrivsatsen av den bortkommenterade får man det bättre `FELKOD= invalid enumerant`. Utelämnat `glBegin(...);` skulle ge `FELKOD= 502` (felaktig operation) liksom gl-satser (de flesta) som inte får förekomma mellan `glBegin` och `glEnd` (t ex `glGetError`, `glGetFloatv(GL_CURRENT_NORMAL, a)`). ❋

Ett anrop av `glGetError` nollställer felflaggan/värdet. Vid upprepade fel före anrop av `glGetError` lagras bara det första felet, dvs läs av ofta för att hitta rätt ställe.
Lämpligen villkoras utskriften enligt

```
if ((e = glGetError()) != 0) printf("FELKOD= %s\n", gluErrorString(e));
```

Det är en god idé att ha en sådan rad åtminstone i slutet av omritningsproceduren. Härigenom upptäcks bl a det inte ovanliga nybörjarfelet `glEnd` i st f `glEnd()`.

31 Selektering med gluUnProject

GLU-metoden `gluUnproject` nämns i avsnitt 26.1. I 26.2 handlar selektering om val av ett visst objekt. Det är tekniskt tungt i OpenGL och vi behandlar det i kursen summariskt. Men ibland vill man peka på ett objekt och få reda på vad motsvarande punkt på objektet har för koordinater (i modellkoordinatsystemet eller ev världskoordinatsystemet). Här kommer `gluUnproject` till vår hjälp. Den omvandlar nämligen en muskoordinat till t ex modellkoordinater.

Exempel: Vi modellerar jordklotet med en snurrande texturerad enhetssfär (texturen tas ej med i koden nedan). Vi vill med musen kunna peka på olika platser på jorden och få reda på motsvarande koordinater i någon form (här nöjer vi oss med (x,y,z)). T ex skall vi för "nordpolen" få (0,0,1) oberoende av hur roterad sfären är. Kör vi programmet och klickar på nordpolen får vi

```
> GL_UNPROJECT
Modellkoord: (0.016598, -0.008299, 0.992942)
```

Samma resultat om vi först snurrar på sfären (med piltangenterna). Trycker vi utanför sfären, får man något av följande typ.

```
Du pekade utanför
```

Programmet (exklusive kod för piltangenterna och de vanliga include)

```
// Flera av dessa globala variabler kan ligga lokalt i mouse
// Modellkoordinater från gluUnproject
GLdouble wx=0, wy=0, wz=0;
// Vektorer för tranformationsmatriser
GLint viewport[4];
GLdouble mvmatrix[16], projmatrix[16];
// Rotationsvinklar
GLdouble VinkX = 0, VinkY = 0, VinkZ = 0;

void InitGL(GLvoid) {
     glClearColor(1.0f, 1.0f, 1.0f, 1.0f);   // Vit bakgrund
     glEnable(GL_DEPTH_TEST);
     glColor3f(1,0,0);   // Röd ritfärg
}

void update() {
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glLoadIdentity();
     gluLookAt(0,0,5, 0,0,0, 0,1,0);
     glRotatef(VinkZ, 0,0,1); glRotatef(VinkY, 0,1,0); glRotatef(VinkX, 1,0,0);
     glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
     glutWireSphere(1,10,5);
     glutSwapBuffers();
}

void reshape(int width, int height) {
      glViewport(0,0,width, height);
      glMatrixMode(GL_PROJECTION); glLoadIdentity();
      gluPerspective(45.0, ((GLfloat)width)/height, 0.1f, 10.0f);
      glMatrixMode(GL_MODELVIEW);
}

void mouse(int button, int state, int x, int y) {
   GLint realy;  /*  OpenGL y coordinate position  */
   GLfloat zbuff;
   if (button==GLUT_LEFT_BUTTON) {
      if (state == GLUT_DOWN) {
          glGetIntegerv (GL_VIEWPORT, viewport);
          glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);
          /*  viewport[3] är fönsterhöjden */
          realy = viewport[3] - (GLint) y - 1;

          // Räkna ut z via djupminnet
          glReadPixels(x,realy,1,1,GL_DEPTH_COMPONENT, GL_FLOAT, &zbuff);
          gluUnProject ((GLdouble)x, (GLdouble)realy, zbuff,
             mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
          if (zbuff>0.9999) printf("Du pekade utanför\n");
          else printf ("Modellkoord: (%f, %f, %f)\n", wx, wy, wz);
          glutPostRedisplay();
      }
   }
}
```

```
int main(int argc, char** argv) {
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA |GLUT_DEPTH);
   glutInitWindowSize(400,400);
   glutCreateWindow("Peka på plats");
   InitGL();
   glutReshapeFunc(reshape);
   glutDisplayFunc(update);
   glutKeyboardFunc(key);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

Programmet tar fram modellvymatrisen (i *update* eftersom den ändras fortlöpande) och projektionsmatrisen (i *mouse*). Genom att transformera punktens normaliserade koordinater med inverserna av dessa matriser kan *gluUnProject* få fram koordinater i modellkoordinatsystemet. Nu känner vi inte den normaliserade z-koordinaten (djupet). Men vi tar den från djupbufferten. Programmet tar fram träffpunkten på ytterligare ett sätt, som inte finns med i listningen och som inte är generellt. För detaljerad beskrivning av *gluUnProject* se manualblad.

32 Open Inventor

Förmodligen har du vid det här laget upptäckt att OpenGL är rätt primitivt. Det finns t ex inte några färdiga redskap för att läsa in eller skriva scenbeskrivningar. Och selektering är mer än lovligt omständligt. Open Inventor är en C++-produkt som löser ett antal problem och inför det viktiga begreppet scengraf. Open Inventor finns numera som en kommersiell produkt (som utvecklas) från företaget TGS. Dessutom finns det som sagt i en stillastående "open-source" version (dock har man inte frisläppt texten till den bok *The Inventor Mentor* som beskriver användningen).

På kurssidan hoppas jag att det finns litet ytterligare information om Open Inventor.

För att kunna använda Open Inventor på våra Linux-datorer måste du använda lägst g++ version 3.2 (nuvarande är 3.4). Under Windows behövs lägst Microsoft Visual C++ 6.0 eller g++ version 3.2. Detta beror på att den interna namngivningen av C++-metoder skiljer sig åt mellan olika kompilatorer och olika versioner (så är det inte i C)!. Den nyfikne får vara beredd på ofullständigheter och överraskningar.

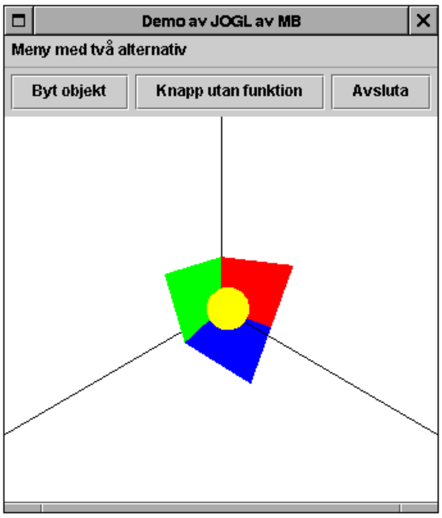
33 Java och OpenGL

I dessa dagar är det oundvikligt att ställa sig frågan om man från Java kan komma åt OpenGL. Skälet skulle väl främst vara att få tillgång till 3D-grafik utan att behöva arbeta i t ex C, eftersom 2D-grafik finns redan i Java. Men det skälet har egentligen ingen bärighet längre ty det finns en utvidgning Java3D, som ger 3D-grafik på en högre nivå (bl a scengrafer) än OpenGL. Enda kruxet är att Java3D känns trögt, trots att det i sin tur utnyttjar OpenGL och därmed eventuell kraftfull grafikhårdvara. Så kanske frågan ändå är befogad en tid framåt. Och många har säkert lagt ned möda på egna lösningar. Problemet är inte enkelt. Bl a p g a händelsehantering, grafik och GUI.

Dessa rader vänder sig till den som är intresserad av att kunna producera OpenGL-grafik med ett Java-program. Vi har under ett par år använt ett fritt system *OpenGL for Java*, ursprungligen sluttillverkat som ett examensarbete i Tyskland av Sven Goethel. Under år 2003 insåg Sun emellertid behovet (man skyller på spelindustrin som vill utnyttja finesser som inte finns i Java2D och Java3D) av ett sådant system och har skapat JOGL för ändamålet. JOGL finns för Windows, Linux, Solaris på Sun och MacOS. Består av ren javakod och s k native kod för var och en av plattformarna. JOGL-funktionaliteten kommer nog att ingå i Java så småningom, men är ännu inte färdigutvecklad. Via webbsidan `https://jogl.dev.java.net/` når man dokumentation och programvara (binärer och källkod). Just nu gäller version *JSR-231 beta03* (feb 2006). För den som läst Java som enda programspråk, kan detta system vara ett bra sätt att närma sig OpenGL och grafikprogrammering. JOGL känns renare än det tidigare använda systemet och stöder hela OpenGL 1.5/2.0, men tyvärr inte hela GLU. På kursens webbsida finns en länk till API-dokumentationen.

För att underlätta har jag som vanligt gjort i ordning kommandoprocedurer. Säg att ditt Java-program heter `Prog.java`. Då kompilerar man på Linux-datorerna med `JOGL_COMPILE Prog` och kör med `JOGL_RUN Prog`. Detta om du är inloggad på grafikkontot. Kommandoprocedurerna ger några systemvariabler rätt värden och utför sedan sin uppgift. I Windows kompilerar man med
`javac -classpath .;"C:\Program Files\JOGL\jogl.jar" Prog.java`.

Låt oss nu diskutera hur man kodar i anslutning till det program vars körning visas till höger (på en Sun-dator). Kodningen är troligen inte gjord på det bästa sättet, utan jag har känt mig nöjd när något börjat fungera.



Programkoden heter `JOGL1_MB.java` och finns i katalogen `$DG/JAVA/DEMO`. Om du flyttar dig dit kan du köra det med `JOGL_RUN JOGL1_MB` och kan naturligtvis kopiera koden som du vill. Koden resulterar vid kompilering i flera klassfiler (.class).

Programmet visar ett par objekt som snurrar kring den uppåtriktade y-axeln. Man kan växla mellan två olika objektuppsättningar med bl a en av de övre knapparna. För att mus och tangenter skall ha effekt i ritpanelen måste den vara aktiv, dvs kanske måste man göra ett inledande klick där. När vi nu övergår till att diskutera koden kommer jag inte att beskriva de enskilda metodernas uppgifter, eftersom de är samma som i t ex C. Du hänvisas alltså tyvärr till huvudhäftet.

Användarsnittet (knappar och interaktion etc) är gjort i Java medan grafiken produceras av OpenGL men är beordrad från Java, dvs all programkod är skriven i Java. Java fungerar således som ersättare till GLUT (GLUTs objektprocedurer kan fortfarande vara av intresse).

Programmet utgörs av tre klasser. Huvudklassen är *JOGL1_MB*, som svarar för igångsättning. Klassen *MyFrame* bygger upp användarsnittet och har hand om mus- och tangenthanteringen. Den tredje klassen *MyGLEventListener* beskriver den animerade grafiken. All datorgrafik ryms i instansmetoderna *reshape* och *display* i *MyGLEventListener* och det är nästan bara här ändringar behöver göras om man vill sätta ihop ett program med annan grafik. Dessa namn är givna! Vi kan alltså inte som i C ha t ex *storleksbyte* och *rita* i stället. Som du redan vet kan java-program struktureras på många olika sätt. En del demoprogram för JOGL väljer att arbeta med en enda klass som har interna klasser.

Eftersom klassen *MyGLEventListener* är den intressantaste börjar vi där. Den ser grovt ut så här:

```
class MyGLEventListener implements GLEventListener {
     private GL gl;
     private GLU glu = new GLU();
     private GLUT glut = new GLUT();   // Enbart om GLUT
     public boolean OBJEKT1=true, ANIM = true;
     private double vinkel = 0.0;
     // Fyra metoder i gränssnittet GLEventListener som måste implementeras.
     // Vi anropar dem aldrig själva. Detaljer längre fram.
     public void init(GLAutoDrawable drawable) {...}
     public void reshape(GLAutoDrawable drawable, int x, int y,
                         int width, int height) {...}
     public void display(GLAutoDrawable drawable) {...}
     public void displayChanged (GLAutoDrawable drawable,
                   boolean modeChanged, boolean deviceChanged) {}
     // Ytterligare problemspecifika metoder, se längre fram
}
```

I klassen *MyGLEventListener* finns variablerna *gl*, *glu* och *glut* och *drawable* av klasserna *GL*, *GLU*, *GLUT* respektive *GLDrawable*. Dessa kommer att kopplas till motsvarande objekt i init-metoden. Objekten *gl* , *glu* och *glut* används för att komma åt OpenGL-rutiner, GLU-rutiner respektive GLUT-rutiner. De flesta GLUT-rutiner har ersatts av de interaktionsmöjligheter som ligger i Java, men vill man komma åt dem som har med färdiga geometriska objekt att göra får man själv skapa ett objekt *glut* (se koden).

Låt oss först kommentera *reshape*, som anropas automatiskt när fönstret ändrar storlek. Den ser ut så här:

```
public void reshape(GLAutoDrawable drawable, int x, int y,
                    int width, int height) {
    double h = (double)height / (double)width;
    gl.glViewport(0,0,width,height);
    gl.glMatrixMode(GL.GL_PROJECTION);
    gl.glLoadIdentity();
    glu.gluPerspective(120.0, 1.0/h, 1.0,5.0);
}
```

Noterbart är att de tre första parametrarna som man i praktiken har begränsad nytta av inte finns med i den tidigare C-versionen. Vi sätter här upp projektionsförhållandena på samma sätt som i t ex Exempel 5. Samma typ av namn, men gl-namn är metoder hos objektet *gl* och glu-namn hos *glu*. OpenGL-konstanter som i C heter t ex `GL_PROJECTION` nås nu som klasskonstanter, t ex `GL.GL_PROJECTION` (i Java 5 kan man slippa ifrån `GL.`).

Vi övergår till *display*, som anropas automatiskt när fönstret behöver ritas om. Om vi inte vill ha animeringsbeteendet kan vi sätta variabeln *ANIM* till **`false`**, vilket hindrar att vinkeln ökas inför varje omritning.

```
public void display(GLAutoDrawable drawable)  {
    if (ANIM) vinkel = vinkel + 0.5;
    gl.glClearColor(1.0f,1.0f,1.0f,1.0f);
    gl.glClear(GL.GL_COLOR_BUFFER_BIT | GL.GL_DEPTH_BUFFER_BIT);
    // Betraktarens position etc
    // Kan placeras i reshape om observatören - som här - orörlig
    gl.glMatrixMode(GL_MODELVIEW);
    gl.glLoadIdentity();
    glu.gluLookAt(2.0,2.0,2.0, 0.0,0.0,0.0,0.0,1.0,0.0);
```

```
        // Den egentliga ritningen
        koordinataxlar();
        gl.glRotated(vinkel, 0.0, 1.0, 0.0);
        if (OBJEKT1) triangel(); else tre_kubsidor(0.0,0.0,0.0);
        gl.glColor3d(1,1,0);
        gl.glTranslated(0.5,0.5,0.5);
        glut.glutSolidSphere(0.5,12,12);
    }
```

Byte av bildminne görs automatiskt. F ö ser det ut som i C. Vi suddar och sätter upp betraktningsförhållanden. Sedan ritas koordinataxlar (se kod nedan) och beroende på värdet hos OBJEKT1 en roterad triangel eller en roterad del av en kub. Slutligen med hjälp av GLUT en först translaterad och sedan roterad solid sfär.

Nu följer instansmetoderna *koordinataxlar* och *triangel* i *Rityta*. För metoden *tre_kubsidor* hänvisas du till programfilen `JOGL1_MB.java`.

```
public void koordinataxlar() {
    gl.glBegin(GL.GL_LINES);
       gl.glColor3d(0.0,0.0,0.0);
       gl.glVertex3d(0.0,0.0,0.0); gl.glVertex3d(4.0,0.0,0.0); // x-axeln
       gl.glVertex3d(0.0,0.0,0.0); gl.glVertex3d(0.0,4.0,0.0);
       gl.glVertex3d(0.0,0.0,0.0); gl.glVertex3d(0.0,0.0,4.0);
    gl.glEnd();
}
public void triangel() {
    gl.glBegin(GL.GL_POLYGON);
       // Olika färg i varje hörn är skoj
       gl.glColor3d(1.0,0.0,0.0); gl.glVertex3d(-3.0,0.0,0.0);
       gl.glColor3d(1.0,1.0,0.0); gl.glVertex3d(0.0,3.0,0.0);
       gl.glColor3d(0.0,1.0,1.0); gl.glVertex3d(3.0,0.0,0.0);
    gl.glEnd();
}
```

Jag hoppas att du ser hur likt C det är. Litet mer skrivarbete förstås. Det är bekvämast att genomgående använda **double**-metoder (d) hellre än **float**-metoder (f), eftersom konstanter annars måste skrivas på formen 1.0f p g a Javas konverteringsregler.

Så till resten av klassen *MyGLEventListener*. Metoden *init* anropas automatiskt när OpenGL kommit igång. Vi ser där till att de fyra variablerna *gl*, *glu*, *glut* och *drawable* initieras på rätt sätt. Därefter görs diverse initieringar. Dubbelbuffring är automatiskt påslaget.

```
    public void init(GLAutoDrawable drawable) {
        gl = drawable.getGL();
        drawable.setGL(new DebugGL(gl)); // Kan uteslutas; se anm.
        // Om vi har mus- och tangentlyssnare, se nedan
        drawable.addMouseListener(ml);
        drawable.addKeyListener(kl);
        gl.glEnable(GL.GL_DEPTH_TEST);
        // Eventuella andra initieringar
        // Eventuella display-listor etc
    }
```

Den fjärde av de obligatoriska metoderna *displayChanged* implementerar vi som en tom metod, vilket gjordes redan i det inledande skelettet.

Om vi vill ha mus- och tangenthantering kan vi i *MyGLEventListener* definiera lyssnare, t ex

```
        private MouseListener ml = new MouseAdapter() {
             public void mousePressed(MouseEvent e) {
                 System.out.println("Nu tryckte du" + e.getX() + " " + e.getY());
             }
        };
        private KeyListener kl = new KeyAdapter() {
             public void keyTyped(KeyEvent e) {
                  System.out.println("Tangenten " + e.getKeyChar() + " tryckt");
                  if (e.getKeyChar() == 'a') ANIM = !ANIM;
                  if (e.getKeyChar() == 'o') OBJEKT1 = !OBJEKT1;
             }
        };
```

Vi kan inte addera dessa till klassen själv (som inte har någon add-metod), men däremot till *drawable* (som i praktiken är den rityta - se nedan - som används för OpenGL) i *init*.

Och till slut resten av programmet.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.media.opengl.*;          // OpenGL
import javax.media.opengl.glu.GLU;    // GLU
import com.sun.opengl.util.*;          // GLUT m m

class MyGLEventListener implements GLEventListener {
         // Redovisad tidigare
}

// Klass för igångsättning
public class  JOGL1_MB {
        public static void main(String[] args) {
                MyFrame f = new MyFrame();
                System.out.println(Version.getVersion()); // Ger JOGL-version
        }
}
// Klass för användarsnittet
class MyFrame extends JFrame implements ActionListener {
        // Diverse komponenter
        private JButton   bytKnapp = new JButton("Byt objekt");
        private JButton   knapp = new JButton("Knapp utan funktion");
        private JButton   slutKnapp = new JButton("Avsluta");
        private JMenuBar  menyrad = new JMenuBar();
        private JMenu     enMeny = new JMenu("Meny med två alternativ");
        private JMenuItem bytObjekt = new JMenuItem("Växla");
        private JMenuItem avsluta = new JMenuItem("Avsluta");
        private Animator animator;
        // GLCanvas  canvas;  om annan animeringsteknik
        private MyGLEventListener glevents;
        public MyFrame() {
               // Menybiten
               enMeny.getPopupMenu().setLightWeightPopupEnabled(false); // Se
               setJMenuBar(menyrad);
               menyrad.add(enMeny);
               enMeny.add(bytObjekt); enMeny.add(avsluta);
```

```
            bytObjekt.addActionListener(this);
            avsluta.addActionListener(this);
            // En yta för OpenGL-grafik.
            GLCanvas canvas = new GLCanvas();
            glevents = new MyGLEventListener();
            canvas.addGLEventListener(glevents);
            // Skapa en panel och lägg i den tre egna knappar, standardlayout
            JPanel p = new JPanel();
            p.add(bytKnapp); p.add(knapp); p.add(slutKnapp);
            bytKnapp.addActionListener(this);
            knapp.addActionListener(this);
            slutKnapp.addActionListener(this);
            canvas.setSize(new Dimension(300,300));
            add("Center", canvas); add("North", p);
            setTitle("Demo av JOGL av MB");
            setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);//Ej perf
            pack();
            setVisible(true);
            // Åstadkom ständig omritning av OpenGL-ytan
            animator = new Animator(canvas);
            animator.start();
        }

        public void actionPerformed(ActionEvent e) {
            Object obj = e.getSource();
            if (e.getSource() == bytKnapp) {
                glevents.OBJEKT1 = !glevents.OBJEKT1;
            } else if (e.getSource() == knapp) {
                System.out.println("Knappen utan funktion");
            } else if ((obj == slutKnapp) || (obj == avsluta)) {
                animator.stop(); System.exit(0);
            } else if (obj == bytObjekt) glevents.OBJEKT1 = !glevents.OBJEKT1;
        }
}
```

Två saker är nya här. Dels skapar vi en yta för OpenGL-grafiken av klassen *GLCanvas*. För att kunna anpassa ytan till vad vi vill rita inför man ett objekt *glevents* av vår klass *MyGLEventListener* och kopplar det som lyssnare till ritytan. Den andra nyheten är att vi kan åstadkomma ständiga omritningar av ritytan genom att införa och starta ett *Animator*-objekt, vilket görs sist i *main*-metoden.

Ett annat exempel i `$DG/JAVA/DEMO/GEARS` heter `Gears.java` och kan köras med `JOGL_RUN Gears`. I det exemplet finns belysning och man kan med musen påverka föremålen. Det är hämtat från webb-platsen för *JOGL*.

Om man gör flera Java-program av denna typ är det lämpligast att som jag skapa en katalog för varje program. Alternativt kan man namnge klassen *MyGLEventListener* olika i programmen. Annars måste man kompilera om inför varje ny körning.

Vill man ha ritning under musrörelse - som i exempel 2 - blir det litet trassligare. Det är inte aktuellt i laborationerna. Jag har emellertid gjort en Java-variant `JOGL2_MB.java` av `GL_ENKEL2.c` (exempel 2), som kan användas som alternativ modell i Lab 1 av dem som väljer Java (modellen i exemplet ovan räcker som sagt för laborationsuppgifterna). Finns i kurskontots `$DG/JAVA/DEMO/JOGL2` och via webbsidan. Vi måste nu samla på oss de streck som bygger upp kurvan, vilket kan ske i en vektor eller vektorlista. För övrigt behöver vi bara byta ut tangentlyssnaren mot en *MouseMotionListener*. Omritning sker så fort *Animator*-objektet begär det. I stället för ett sådant skulle vi kunna anropa `drawable.display()` på rätt ställen i muslyssnarna. Vi får nu även korrekt uppdatering, vilket den ursprungliga `GL_ENKEL2.c` inte tillhandahåller. Observera att den använda tekniken måste användas även i ren Java om man skall vara säker på att det skall bli rätt.

Anmärkningar:

- I stället för *GLCanvas* som är en s k tungviktskomponent (AWT), kan man ha *GLJPanel* som är en lättviktskomponent (Swing), men då utnyttjas f n grafikhårdvaran sämre vilket leder till avsevärt lägre prestanda. Fördelen är att menyn inte döljs av ritytan. Å andra sidan verkar det finnas en del andra buggar (t ex blir färgerna fel). *GLCanvas* och *GLJPanel* implementerar gränssnittet *GLAutoDrawable* men är underklass till *Canvas* (föregångare till *JPanel*) respektive *JPanel*.
- Utan raden
  `enMeny.getPopupMenu().setLightWeightPopupEnabled(false);`
  döljs menyn av ritytan (vilket beror på blandningen av Swing och AWT).
- Man kan få rörelse utan att använda *Animator*. I stället anropas instansmetoden `display()` i *GLDrawable* direkt, som i sin tur anropar `display(...)` i *GLEventListener* (dvs i *MyGLEventListener* ovan). Fördelen är att man därmed själv kan bestämma animeringshastigheten. Dock är detta sätt inte alldeles tydligt dokumenterat och kan eventuellt kräva ytterligare åtgärder. I vårt exempel kan vi således få animering med raderna
  ```
  while (true) {
        f.canvas.display();
  }
  ```
  i slutet av *main*-metoden (*canvas* måste då deklareras som oprivat instansvariabel i st f som lokal variabel).
- Utan `animator.stop()` blir det ett felmeddelande i PC-miljön.
- Raden `drawable.setGL(new DebugGL(gl));` i init-metoden kan uteslutas och prestanda ökas då något. Raden gör att man automatiskt får den typ av felkontroll som beskrivs i avsnitt 32.
- JOGL för med sig en komplikation jämfört med C. Matriser lagras i Java på ett helt annat sätt än i C. En matris med 10 rader och 5 element i varje deklareras i C `m[10][5]` och kan kommer åt tredje radens andra element med `m[3][2]`. När vi i Java vill kommunicera med OpenGL-metoder med matriser, så får vi använda vektorer i stället och ange gränsen till 10*5 och kommer åt samma element med `m[3*5+2]`.
- Javas datatyp för värden i en byte heter **`byte`** och klarar heltal mellan -128 och 127. Javas datatyp **`char`** upptar ju två bytes.

34 Avslutningsord

Som titeln anger har detta varit en introduktion till OpenGL. Grafikbiblioteket kan mycket mer men det finns också saker som man kanske hellre gör rätt fristående från OpenGL, t ex strålföljning. Under kursens gång hoppas jag att vi hinner med att exemplifiera ytterligare en del finesser i OpenGL.

35 Litteratur och andra referenser för OpenGL

Segal, M./Akeley, K.: The OpenGL Graphics System: A Specification (version 2.0)
Hittas via `www.opengl.org`. För att vara ett standarddokument exceptionellt trevligt och lättläst. Dokumentationen är dock hårt copyrightad av SGI, vilket gör det omöjligt att komma åt den annat än via nätadressen ovan.

Neider, J et al: OpenGL Programming Guide, Addison-Wesley, 1993 (3 ed 1999).
Standardverket när det gäller OpenGL. Första upplagan finns via nätet (se kurssidan).

Hill, F.S: Computer Graphics Using OpenGL (2nd ed), Prentice Hall, 2000
En bok som tar upp både OpenGL och GLUT och samtidigt är en lärobok i datorgrafik.

Angel, E: Interactive Computer Graphics, Addison-Wesley, 1997 (3 ed 2003)
En bok som tar upp både OpenGL och GLUT och försöker lära ut datorgrafik. Första upplagan var rätt ytlig och förkastades vid KTH efter ett försök. Den nya upplagan är bättre, men kan inte tävla med Hills bok.

Wright, R./Sweet, M.: OpenGL Superbible, Waite Group Press, 1996
En populär bok - att döma av kön på Chalmers bibliotek. Använder inte GLUT utan primitivare versioner (som gäller enbart PC-miljön). Därmed av begränsat värde (tycker jag). Författarna har dessutom missuppfattat matrishantering (möjligen medvetet). Denna upplaga kan nu läsas via nätet (se kurssidan). Tredje upplagan (juni 2004) finns att köpa.

Fullständiga och läsbara specifikationer finns på nätet även till GLU och GLUT på såväl PostScript- som HTML-form.

Manualsidor på HTML-format kommer man åt direkt (utan att besöka SGI) via kursens hemsida `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/datorgrafik`, som innehåller en hel del annat smått och gott. Manualblad når man också via det vanliga man-kommandot, t ex om du vill veta alla detaljer om glBegin: `man glBegin`

36 Innehåll