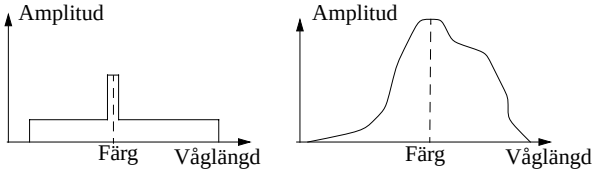
Färg 3(3)

Denna skala är vad regnbågen eller ett brytande prisma visar. Men i praktiken är det vi ser en kombination av flera våglängder. T ex finns inte vitt i färgskalan utan uppkommer genom att flera våglängder blandas. Det kan se ut så här



Den vänstra bilden är idealiserad medan den högra är mera verklighetsförankrad. Ett förenklat sätt att beskriva färgen är att ange den dominerande färgen (markerad i figurerna) (kallas på engelska **Hue**), total intensitet (dvs ytan under kurvan) kallad **Luminans** samt kvoten mellan den dominerande färgens energi och den totala mättnadsgraden (eng. **Saturation**). Detta är vad vi har rattar eller knappar för på en TV. Detta är också grunden för det nämnda HLS-systemet.
Ett gammalt problem är hur man förfar när presentationsutrustningen inte har tillräckligt många färger. Redan en tidning som vill trycka bilder med gråskala drabbas av detta, eftersom trycket antingen är vitt eller svart. Lösningen i just den situationen är att trycka större svarta prickar där bilden är mörk och mindre där den är ljus. Kolla med ett förstoringsglas. I datorvärlden har man väl alltid kunnat reglera intensiteten mera kontinuerligt, men för inte så länge sedan var problemet ändå akut inom datorgrafik. Systemet tillät kanske bara 4 eller 16 olika färger, eventuellt fördefinierade. En efterlikning av tidningsmetoden kallas **halvtoning** (det använda mönstret använder olika antal färgade punkter). En variant för dittring (eng **dithering**).

Format 1(3)

Det finns format för modeller (inkl scener), bilder och animeringar etc. Ett format beskriver hur informationen lagras. Ett format kan vara textbaserat eller binärt, okomprimerat eller komprimerat. Många program har sina egna format och den stora frågan är hur t ex geometriinformation skall kunna utbytas. Detta har varit ett stort problem i CAD-världen.

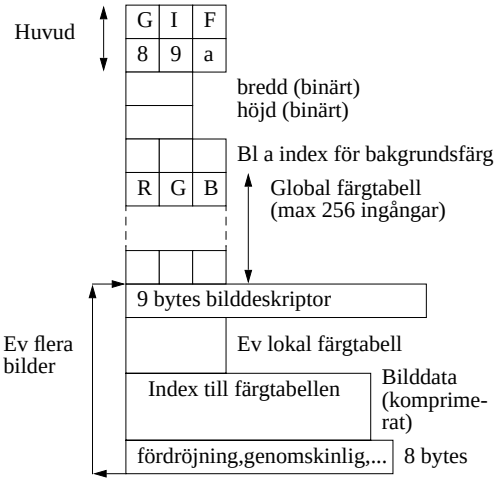
När det gäller modellformat nöjer vi oss med vad vi redan mött. Men det finns många andra, t ex det tidiga `.obj`.

När det gäller bilder är i UNIX-världen olika varianter av formatet PPM (Portable PixMap) vanliga. På nätet dominerar JPEG (Joint Photographic Experts Group) och GIF (Graphics Interchange Format). I PC-världen var det länge BMP (BitMaP) som gällde. För högkvalitativ grafik (hög upplösning), t ex från skannrar, används TIFF (Tag Image File Format). Rit- och bildbehandlingsprogram brukar kunna konvertera mellan några av formaten. JPEG är till skillnad mot de övriga förstörande, i den meningen att det lagrar en approximation till bilden som uppkommer genom att man bl a tar bort höga frekvenser. Ett företag hävdar upphovsrätt till GIF. Som svar på detta skapades det mer avancerade formatet PNG (Portable Network Graphics), som stöds av t ex *Netscape* och *xv*, och en del program, t ex *MATLAB*, sa upp bekantskapen med GIF.

Det mest uppenbara sättet att lagra en (digitaliserad) färgbild vore kanske att lagra RGB-värdena för varje bildpunkt. Vilket skulle betyda 3 bytes/bildpunkt med binär lagring. Detta skulle kunna kallas RGB-format.

Format 2(3)

Låt oss beröra GIF och senare PNG litet mer.

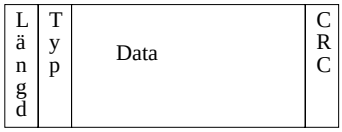


GIF kan visa upp icke-fyrkantiga bilder genom att man låter en färg släppa igenom bakgrundsfärgen (jfr sprites). GIF-animering har vi redan nämnt. Den har sin grund i att en GIF-fil kan lagra flera bilder. GIF uttrycker färger i RGB-systemet, vilket gör att bilden kan se litet olika ut beroende på presentationsutrustning.

Format 3(3)

PNG (uttalas ping) har en något utökad funktionalitet jämfört med GIF. Dock saknas animeringsmöjligheterna, dvs det finns bara en bild per fil. Formatet kan hantera såväl färgtabellbaserade bilder som RGB-bilder. Komprimering görs med samma metod (LZ77) som zip och gzip använder. Animering byggande på PNG finns nu som MNG (uttal ming). PNG och MNG stöds av moderna webbläsare.

En PNG-fil inleds med ett huvud om 8 bytes: \211 P N G \r \n \032 \n. Därefter följer ett antal block (på engelska kallade chunks) med strukturen (CRC=Cyclic Redundancy Check, dvs kontrollinformation).



Varje del utom data-delen är 4 bytes lång. Det första blocket skall ha typen IHDR och innehåller bildens bredd och höjd, bildtyp m m. Exempel på andra block

PLTE: Färgtabell
IDAT: Bilddata
CHRM: CIE-kromaticitet (x,y) för vitpunkt, rött, grönt och blått (32 bytes). Se färgavsnittet.
gAMA: Gamma (4 bytes). Intensitet=(angiven intensitet)$^{\gamma}$.

När det gäller animeringar finns bl a formaten AVI, MPEG och MOV (QuickTime) och som sagt MNG.

Effektivisering med OpenGL. Tidmätning. 1(2)

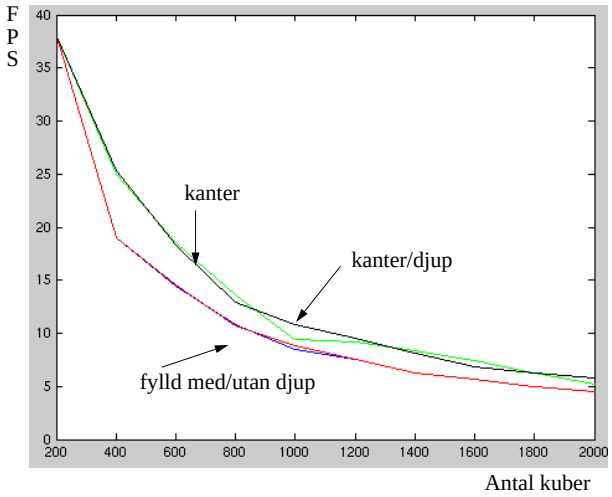
Prestandamätning baseras på tidmätning enligt t ex följande i uppdateringsproceduren. Vi använder funktionen *TickCount* som mäter förfluten tid i millisekunder (OpenGL-häftet, avsnitt 17). I programmet (`TIDER2001_NY.c`) beräknas ett medelvärde för antalet uppdateringar per sekund (FPS=Frames/s) ungefär en gång i sekunden.

```
void display(void) {
    unsigned  long starttime, stoptime, difftime;
    starttime = TickCount();
        glPushMatrix();
        // Radering+gluLookAt+Eventuella transformationer
        // Nu själva ritandet
        myFigure();
        glPopMatrix();
        glutSwapBuffers ();
    stoptime = TickCount();
    Frames = Frames + 1;
    difftime = stoptime - OldTime; // OldTime global
    if (difftime > 1000) {
        printf("MFPS = %f\n", Frames/(0.001*(difftime)));
        Frames = 0; OldTime = stoptime;
    }
}
```

I koden har variabeln *starttime* betydelse bara om man vill mäta tiden för en enstaka omritning.

Effektivisering med OpenGL. Tidmätning. 2(2)

Här visas resultatet av en körning (SUN). Programmet ritar upp ett antal kuber. Diagrammet (gjort med MATLAB) visar antalet uppdateringar per sekund när polygonerna ritas fyllda resp ofyllda, dels med och dels utan djupbufferttest.



Vi ser att antalet kuber som ritas har stor betydelse. Medan djupbuffert eller ej saknar betydelse. Däremot går det litet snabbare om man bara ritar kanterna på sidoytorna. Exempel på MATLAB-kod:

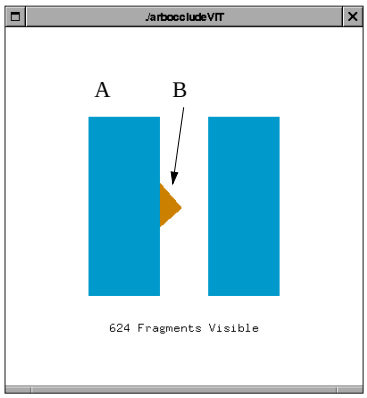
```
>> wire=[38.0,25.3,18.3,12.9,10.8,9.5,8.1,6.8,6.3,5.8];
>> plot(t,wire,'black');
```


Effektivisering med OpenGL

Inom OpenGL finns många möjligheter att effektivisera. Väl så viktigt kan vara att skriva bra C-kod, men det går vi inte in på.

- Remsor och fjädrar. Se OpenGL-häftet, avsnitt 7. I fallet `GL_QUAD_STRIP` behöver vi bara skicka 2 nya hörn per fyrkant i stället för 4 som för `GL_QUADS`. I fallet GL_TRIANGLE_STRIP 1 hörn i stället för tre. I fallet GL_LINE_STRIP bara ett hörn i ställe för två per ny linje. Sparar tid både kommunikations- och transportmässigt.
- Fördröjd ritning (displaylistor). Se OpenGL-häftet, avsnitt 25.
- Polygongallring, som hindrar onödig ritning av frånvända ytor. Redan avhandlat.
- Hörnvektorer (vertex arrays).
- Ocklusionsgallring (eng. occluding). Att ett objekt döljs av ett redan ritat.

Effektivisering med OpenGL: Ocklusionsgallring 1(1)



Körning av ett program *arboccludeVIT* (finns i `$DG/DEMOS`) i MESA-distributionen

Ett objekt B döljs ibland av ett (eller flera) redan ritat A. Ritningen av B (eller som i figuren delar av) hindras då av djuptestet. Men vi måste ju skicka B:s geometridata till grafikprocessorn. Eller också i vårt eget program avgöra om det föreligger ocklusion eller inte. Ett nyare alternativ (fr o m 1.5-kärnan) är att låta grafikprocessorn sköta avgörandet. Detta går i princip till så här:

1. Sätt med `glBeginQuery(...)` grafikprocessorn i frågeläge och "rita" B (inget ritande i djupminne eller bildminne, vilket kan fixas med `glDepthMask(GL_FALSE)` och `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE,GL_FALSE)`).
2. Vänta på svar, som talar om hur många fragment som är synliga.
3. Sätt grafikprocessorn i vanligt läge och rita B om det inte döljs helt, dvs om antalet synliga fragment > 0.

Eftersom vi även nu skickar geometridata (t o m eventuellt två gånger) är inget vunnet. Tanken är i stället att man skall bunta ihop objekt och göra ocklusionstestet för begränsningskroppen. Mer om detta snart.

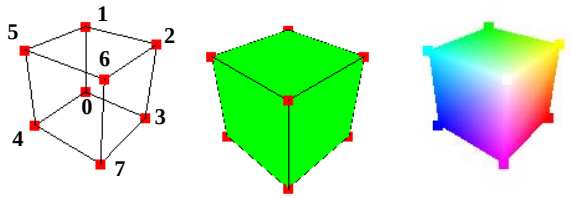
Effektivisering med OpenGL: Hörnvektorer. Ingår ej 2005.

Hittills har vi skickat en hörnpunkt i taget och tillhörande information. T ex

`glVertex3f(7.5,3.4,1.0);` eller `glVertex3fv(P[2]);`

Men man kan också skicka många hörn i ett svep. Detta sparar proceduranrop. Det kan också tänkas reducera kommunikations- och transformationstid om grafikprocessorn "cachar" hörnen.

Detta belyses bäst med ett exempel. Säg att vi vill rita kuben i mellersta figuren. I första hand sidoytorna men även hörnpunkterna. Hörnet 0 ligger i origo och hörnet 6 i (1,1,1). För detaljer se manualblad.



Nedan visar vi koden vid användning av hörnvektorer (eng vertex array) och tillhörande globala variabler. Nyheterna är fetlagda.

```
GLfloat horn[]={0,0,0, 0,1,0, 1,1,0, 1,0,0,
                0,0,1, 0,1,1, 1,1,1, 1,0,1};
GLuint  index[]={0,1,2,3, 4,7,6,5, 7,3,2,6, 6,2,1,5,
                 4,5,1,0, 0,3,7,4};

void display(void) {
    glClearColor(1.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
```



```
    //*** glEnableClientState(GL_COLOR_ARRAY);
    // Röda hörnpunkter och med storleken 8
    glColor3f(1.0,0.0,0.0);
    glPointSize(8.0);
    glVertexPointer(3,GL_FLOAT,3*4,horn);
    //*** glColorPointer(3,GL_FLOAT,3*4,horn);
    glDrawArrays(GL_POINTS,0,8);
    glColor3f(0.0,0.0,0.0);
    //glDrawElements(GL_LINES,24,GL_UNSIGNED_INT,indexkant);
    glColor3f(0.0,1.0,0.0);
    glDrawElements(GL_QUADS,24,GL_UNSIGNED_INT,index);
    glutSwapBuffers();
}
```

I vektorn *horn* finns koordinaterna för de 8 hörnen. I vektorn *index* finns kvadraternas hörn via index till horn-vektorn. Med anropet av *glDrawArrays* skickas alla hörnen till grafikprocessorn och ritas som punkter. Med anropet av *glDrawElements* skickas alla hörn som det refereras till i index-vektorn och hörnen bildar hörn i successiva fyrhörningar som ritas. Man kan här hoppas på att grafikprocessorn bara behöver transformera nya hörn. För att allt detta skall fungera måste man med *glVertexPointer* definera att just vektorn *horn* skall användas och samtidigt ange vissa layout-data. Dessutom måste ett tillstånd sättas med *glEnableClientState*.

Om vi inför

```
GLuint indexkant[]={1,2, 2,6, 6,5, 5,1, 0,3,
          3,7, 7,4, 4,0, 4,5, 0,1, 3,2, 7,6};
```

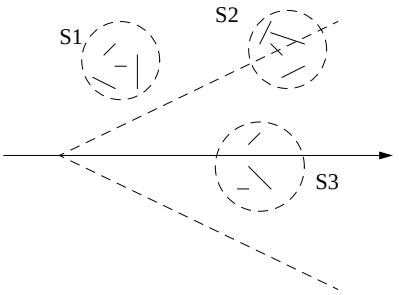
och tar bort kommentartecknen före det näst sista *glDrawElements* kommer även kanterna att ritas. Man kan skicka flera vektorer samtidigt, t ex en vektor med färginformation för hörnen. Då tar man i ovanstående kod bort //*** på två platser. För enkelhets skull använder jag hörndata som färgdata, men det hade gått utmärkt att ha en separat färgvektor. Resultatet blir figuren till höger.

Effektiviseringar i applikationen

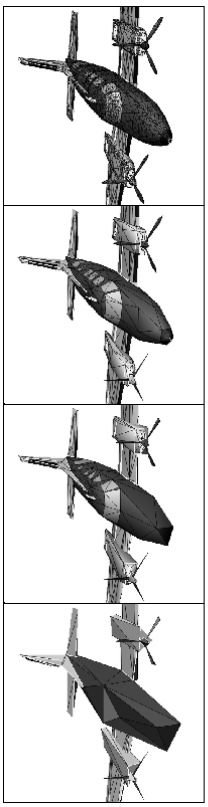
Det finns åtskilliga principiella effektiviseringar som tills vidare bara kan göras i det egna grafikprogrammet. De två viktigaste är kanske **synpyramidgallring** (eng. view frustum culling) och **detaljnivåer** (LOD=Levels of detail).

synpyramidgallring

OpenGL ser ju till att det som ligger utanför den stympade synpyramiden klipps bort. Men detta sker efter det att motsvarande hörn skickats till grafikprocessorn och transformerats, dvs en del arbete har utförts i onödan. Om objekten - i figuren symboliserade med streck - buntas ihop i begränsningsobjekt kan det löna sig att göra egna test i förväg. I figuren finns tre sådana begränsningsobjekt sfärerna S1, S2 och S3.



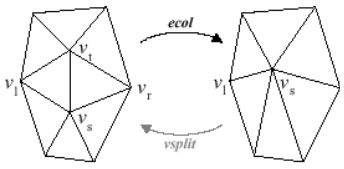
I figurens fall kan vi utan vidare förkasta objekten i S1, dvs vi reducerar antalet hörn som skickas med cirka 30%. Se även "Från värld till skärm", avsnitt 9.



Detaljnivåer

Objekt som befinner sig långt ifrån användaren behöver inte ritas med samma nogrannhet som när de är nära. En teknik är därför att ha ett objekt i flera olika upplösningar. Tyvärr störs betraktaren när övergång sker mellan två nivåer.

En lösning på det problemet är **progressiva nät** som infördes av Hugues Hoppe. Man startar med en fin modell och reducerar successivt denna genom kantkollapser (se fig nedan) till grövre och slutar med en grov M_0. Genom att spara information om hur reduceringen gått till kan de finare återskapas genom hörndelning:M_0->M_1->...->M_n=M



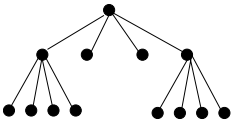
I figuren till vänster är n=6698. Man kan också skapa en mjuk övergång mellan de olika stegen (geometrisk morfing). Enligt Hoppe sparas minne på att utgå från M_0 snarare än M_n.

Kvadrär- och oktalträd 1(2)

Vi skall nu se på ett par “automatiska” sätt att bunta ihop objekt, vilket är en förutsättning för att kunna göra synpyramidgallring och ocklusionsgallring på stora scener.

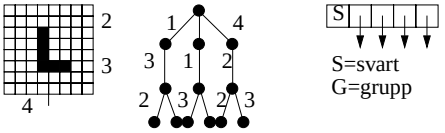
Vi har i denna kurs mött binära träd i samband med BSP. Kvadrärträd (eng. quadtree) och oktal träd (eng. octal tree) är också vanliga i datorgrafik. De kan användas för att öka hastigheten vid uppritandet (och vid kollisionskontroll) och i vissa fall även för att spara minne.

Ett **kvadrärträd** är ett träd med som mest 4 barn per nod.



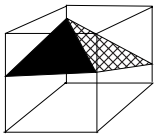
Används typiskt för 2D-kvadrater successivt halverade. I ett **oktalträd** har noderna som mest 8 barn. Används typiskt för kuber successivt halverade.

Exempel: En svart-vit 8x8-bild där de flesta punkterna är svarta kan med fördel representeras med ett kvadrärträd. En tidig rasterskärm från Tektronix (70-talet) använde f ö den tekniken för bildminnet.



Kvadrär- och oktalträd 2(2)

För en digital terrängmodell bestående av

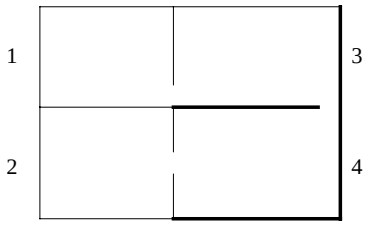


dvs höjder i ett rutnät, säg nxn st (n gärna en 2-potens) passar ett kvadrärträd bra. För varje nod lagrar vi bl a max- och minhöjd och kan med rekursion göra synpyramid- eller ocklusionsgallring på de enskilda blocken. Om vi t ex står så att terrängen inte alls är synlig räcker det med att testa rotnoden.

För en allmännare 3D-scen där vi kan ha flera objekt med samma xy-värden, passar oktalträd. Vi slår in hela scenen i en kub, som sedan successivt halveras. Delkuber som inte innehåller objekt ignoreras. Delkuber som innehåller objekt testas rekursivt vid synpyramid- eller ocklusionsgallring.

PVS. Portaler

I gallringsmetoderna sänder vi en begränsad del av scenen till grafikprocessorn. Denna uppsättning av scenens alla polygoner kallas PVS (potentially visible set) och varierar beroende på var betraktaren befinner sig. Hittills har vi bestämt den dynamiskt under exekveringen. Man skulle också kunna tänka sig att man i förväg beräknade PVS-er för de tänkbara positionerna. Mest omtalat är detta nog för scener med rum (celler) och portaler (dörrar och fönster), vilka förekommer i spel och arkitektprogram.



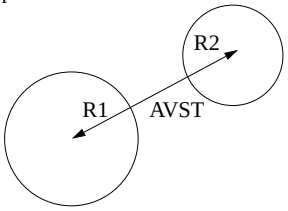
Låt oss först anta att rummen är tomma på föremål. Från rum 2 kan vi som mest se rummets väggar samt vissa av väggarna i rum 4 (fetlagda) och en del av en vägg i rum 3. Motsvarande för övriga rum. Om rummen inte är tomma utökas naturligtvis PVS-arna. Det viktiga är att PVS-arna i princip kan beräknas i förväg och att vi sedan kan välja PVS efter position. Det gäller naturligtvis att som vanligt lagra informationen på ett vettigt sätt, men det kan vi inte gå in på.

Kollisionsdetektering

I scener med rörliga objekt är det för realismens skull viktigt att upptäcka kollisioner och att se till att lämplig fysisk åtgärd vidtages då. Eftersom man “samplar” rörelsen (tiden), kan man ibland missa en aning. Området är aktuellt i bl a spel och robotkinematik.

I en scen med N objekt innebär fullständig kollisionsdetektering att N^2 tester skall göras för varje bildruta. Men ofta är de rörliga objekten få, dvs testantalet kan reduceras till O(N). I allmänna fallet behövs hierarkier och enkla begränsande objekt i form av t ex sfärer eller rätblock. Med rätt datastruktur kan man hitta O(NlogN)-algoritmer.

Enklaste fallet: Sfär mot sfär. Vi får kollision när avståndet AVST mellan de två mittpunkterna är lika med summan av radierna.



Någorlunda enkelt fall: Axelparallella rätblock (AABB=Axis Aligned Bounding Boxes).

Svårare fall: Andra objekt. Flera beräkningsgeometriska problem. Men fallet med snett belägna rätblock (OBB=Oriented Bounding Box) också väl utrett.

Stereografik 1(5)

Vi har nu sett på en mängd olika metoder att få höggradig realism. Emellertid fattas något, nämligen riktigt 3D-intryck. Perspektivtransformationen ger t ex bara en partiell djupkänsla (depth cueing). Denna brist har man i andra sammanhang försökt råda bot på ända sedan människan började kunna producera kopior av verkligheten i form av framför allt foton. Och man har trott sig ha lösningar som skulle bli kommersiella framgångar. Nu har problemet uppmärksammats i datorvärlden. Grundidén i alla lösningar är att producera en bild för vardera ögat och att se till att respektive öga ser bara sin bild. Seriösa intressenter hittar man inom områden som CAD och vetenskaplig visualisering. Men kommersiellt är troligen privatmarknaden (datorspel och TV) väl så betydelsefull. Vi nämner ett par tekniker: färgade glasögon respektive blinkande glasögon. Sensationella system för stereografik dyker upp med en viss regelbundenhet.

Röd-blå glasögon

Man tillverkar en enda bild, som innehåller det vänstra ögats bild ritad i blått (typiskt) och det högra ögats ritad i rött. Observatören utrustas med enkla glasögon, med röd plast för vänstra ögat och blå plast för det andra. Om vi ritar mot en vitt bakgrund, kommer det vänstra ögat att uppfatta bakgrunden som röd och det som är ritat i rött (högra ögats bild) sammanfaller med bakgrunden. Det som ritats med blått, syns emellertid som svart eftersom motsvarande ljus inte släpps igenom. På motsvarande vis uppfattar höger öga det röda som svart mot en blå bakgrund. Härigenom ser vardera ögat enbart sin egen bild. Metoden klarar naturligtvis bara svart-vita bilder, men de kan ha gråskala. Den användes på 50- och 60-talet för framställning av 3D-film, men blev inte någon större succé, kanske delvis beroende på att en annan filmteknik samtidigt kom i allmänt bruk: färgfilmen. Och när

Stereografik 2(5)

svensk television för några år sedan visade ett par av dem, blev det knappast någon längtan efter mer. Tidningar och bokverk använde samma teknik men den fungerade sällan mera än som nyhetens behag. Med en dator och färgskärm är det lätt att tillverka bilder av denna typ utgående från formlerna nedan.

Blinkande glasögon

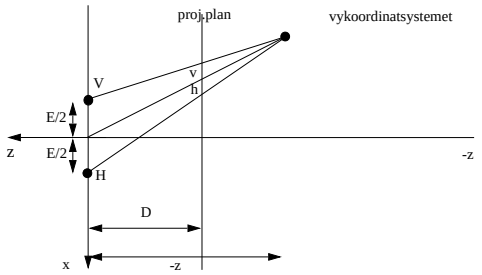
Man visar bilder med dubbla bildfrekvensen, varvid varannan bild är avsedd för vänstra respektive högra ögat. Observatören måste blinka med sina ögon synkroniserat med bildvisningen, vilket torde vara omöjligt, så därför får hon/han ta på sig speciella glasögon i stället. Numera används "blinkande" glasögon av LCD-typ. Synkroniseringen sköts med trådförbindelse eller IR-ljus. Resultatet kan bli mycket bra! (Jag har bara sett det i arbetsstationsmiljö). Många grafikkort har stöd för sådana glasögon. Får jag tippa (1993) så kommer denna teknik att "tvinga" sig på oss så fort som HDTV (högupplösnings-TV) blivit vardagsmat om några år. Tekniken har använts inom filmindustrin och på s k speldatorer (med 3D). I t ex TIME, Nr 16, 1990, berättas om en sådan biograf i Japan. Effekten förstärks där av att filmduken är globformad och jättelik. Göteborg har ju skaffat sig en egen sådan sevärdhet.

Ett par andra sätt som jag kanske nämner, men som inte är värda OH-utrymme: Pulfricheffekten och SIRDS (Single Image Random Dot Stereogram

Enkla stereoskop, föregångare till View Master, var populära redan i fotografins barndom. Man monterade då två bilder i en ställning och försökte få vardera ögat att fokusera på sin bild.

Stereografik: Formler 3(5)

Låt oss börja med att ta fram den erforderliga matematiska formeln utifrån figuren nedan. Hittills har observatören befunnit sig i origo i vykoordinatsystemet. Men nu måste vi särskilja de två ögonen och placerar dem lämpligen symmetriskt kring x=0. Ögonavståndet E är i allmänhet normalt 6-7 cm. En punkt på ett objekt kommer vid projektionen att resultera i två punkter på projektionsplanet, en för vardera ögat. Dessa kommer av symmetriskäl att ligga lika långt ifrån den tidigare enda projektionspunkten, dvs i figuren är v = h.



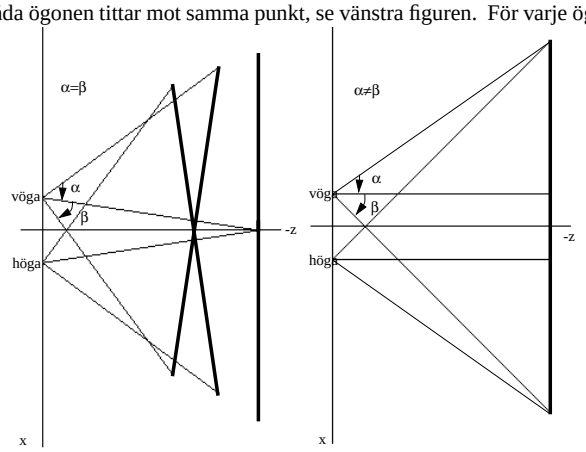
Vi kan beräkna v med likformighet: v förhåller sig till E/2 som -z-D till -z, dvs

$$v = \frac{E}{2}\left(1 + \frac{D}{z}\right)$$

Vi behöver nu bara minska och öka alla x-koordinater med v för att få vänster respektive höger ögas bild. De andra koordinaterna y och z påverkas inte. Vi noterar att stereoseparationen 2v ökar när objektpunktens z-koordinat avtar och är som mest E/2 (när z = -∞).

Stereografik: OpenGL 4(5)

Ett sätt avhandlas i "Introduktion till OpenGL". Det bygger på att båda ögonen tittar mot samma punkt, se vänstra figuren. För varje öga

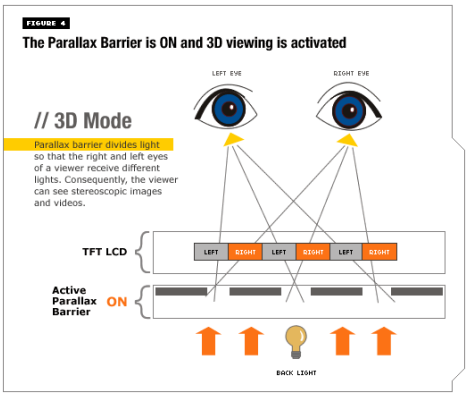


sätter vi med `gluLookAt` upp en symmetrisk synpyramid. Vi projicerar därmed på två olika plan, som jag för tydlighets skull ritat ut något närmare än det egentliga projektionsplanet. Ett korrektare sätt är det som visas i högra figuren. De båda ögonen tittar mot olika punkter. Men det gör också att synpyramiden blir osymmetrisk, vilket gör att `gluLookAt` inte räcker till, utan man måste i stället utnyttja en mera grundläggande procedur `glFrustum`. Vill du ha en utttömmande diskussion kring detta och belysande program, så titta på t ex Paul Bourkes webbsida `http://www.swin.edu.au/astronomy/pbourke/stereographics`. Paul Bourke har under en följd av år gjort i ordning mycket material kring datorgrafik.

Stereografik: Sharps autostereoskop 5(5)

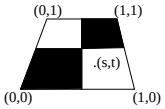
Nu lämnar vi “vetenskap och beprövad erfarenhet”. Hösten 2004 började företaget Sharp sälja en LCD-skärm som inte kräver speciella glasögon. Ett par år tidigare hade man introducerat en mini-version för mobiltelefoner, vilken påstås ha varit en framgång i Japan. Skärmens vanliga LCD-skikt innehåller en bild för vardera ögat. Mellan bakgrundsbelysningen och detta skikt finns en “parallax barrier” av LCD-typ som på något sätt delar upp ljuset så att en del passerar vänster ögas bildpunkter och den andra höger ögas. Man inser att det förmodligen gäller att placera ansiktet på rätt ställe.

Bilden hämtad från `http://www.sharp3d.com/technology/howsharp3dworks/`

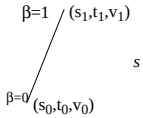


Texturkoordinater m m vid rastreringen

Det som är en linje i vår värld fortsätter att vara en linje i projektionskoordinatsystemet och på vår skärm. Det innebär att vi (OpenGL) bara behöver transformera ändpunkterna och sedan kan generera linjen på skärmen. Motsvarande gäller trianglar och polygoner. Men säg att vi har något som varierar linjärt utmed en linje eller över en triangel i den verkliga världen, t ex djup, färg eller en textur (precisare en texturkoordinat). Som framgår av exemplet räcker det inte att linjärin-



terpolera texturkoordinater om vi vill ha perspektivistiskt korrekt texturering (bilden är korrekt), eftersom objekt skall "krympa" när avståndet till betraktaren ökar. I avsnitt 10 i “Från värld till skärm” reds det ut hur man(grafikkretsen) måste gå tillväga, men vi låter det avsnittet utgå och sammanfattar bara resultatet. Texturkoordinaterna är kända i hörnen, t ex för en linje (s_0,t_0), (s_1,t_1). Men vi kan som sagt inte beräkna dem i en godtycklig punkt på skärmen med vanlig linjär interpolation. Däremot med (v_i är djupet, β är linjeparametern)



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

som innebär en flyttalsdivision per ny bildpunkt och texturkoordinat. Flyttalsdivisioner har hittills alltid varit extremt dyra i förhållande till andra operationer.

Fysikmotorer

Sådana behövs för realistiska rörelseförlopp. Tre som använts kommersiellt kommer från företagen **Havok**, **Novodex** och **Meqon** (svenskt). Under kursens gång har de två sista köpts upp av ett amerikanskt företag **Ageia**, som har annonserat en fysikprocessor (PPU= Physics Processing Unit), som skall påskynda fysikrelaterade beräkningar.
Ett fritt litet enklare system heter **Open Dynamics Engine** (ODE). Presenteras på `www.ode.org` med “ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.”
Vid föreläsning F17 förevisade jag ett demoprogram byggt på en fysikmotor kallad **Newton Game Dynamics** (`www.newtondynamics.com`). På en nyare Windows-dator än min lyser den demon med full glans (kör $DG/PC/NEWTON/newtonplayground/NewtonPlayGround.exe och använd musen). Det visade sig också att min misstanke om att företaget upphört var inkorrekt; länkarna var bara tillfälligt ur funktion. Newton-biblioteket finns för nedladdning till Windows, Linux och Mac. Licensbestämmelserna verkar något oklara. Jag har installerat det under Linux. I distributionen ingick några enklare tutorial-program, som finns färdiga för körning i mapparna $DG/LINUX/NEWTON/newtonSDK/samples/tutorial_*. I avsaknad av källkod går det nog inte att köra de fristående mer avancerade demoprogrammen. Om kvaliteten på Newton uttalar jag mig inte.
Varje sådant här bibliotek för med sig en massa nya metod/funtionsnamn, vilket är en orsak att vi inte går in djupare på dem.

Selektering med gluUnproject 1(4)

GLU-metoden `gluUnproject` nämns i nästan kränkande ordalag i avsnitt 26 i OpenGL-häftet. I det handlar selektering om val av ett visst objekt. Vi går igenom det summariskt. Men ibland vill man peka på ett objekt och få reda på vad motsvarande punkt på objektet har för koordinater (i modellkoordinatsystemet eller ev världskoordinatsystemet). Här kommer `gluUnproject` till vår hjälp. Den omvandlar nämligen en muskoordinat till t ex modellkoordinater.

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

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

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

```
Du pekade utanför
```

Programmet (exklusive kod för piltangenterna och de vanliga include)

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

Selektering med gluUnproject 2-3(4)

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

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

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

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

          // Räkna ut z via djupminnet
```



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

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

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

Selektering med gluUnproject 4(4)

Utdrag ur manualblad

```
NAME
gluUnProject - map window coordinates to object coordinates
C SPECIFICATION
GLint gluUnProject(
   GLdouble winX, GLdouble winY, GLdouble winZ,
   const GLdouble *model, const GLdouble *proj,
   const GLint *view,
   GLdouble* objX, GLdouble* objY, GLdouble* objZ )
PARAMETERS
  winX, winY, winZ
    Specify the window coordinates to be mapped.
  model
    Specifies the modelview matrix  (as  from glGetDoublev).
  proj
    Specifies the projection matrix (as  from  glGetDoublev).
  view
    Specifies the viewport (as from a glGetIntegerv call).
  objX, objY, objZ
    Returns the computed object coordinates.
```


Mera OpenGL

Det finns massor av ytterligare guldkorn i OpenGL.

- Ackumuleringsminne. Kan användas för att få rörelseoskärpa och utjämnade bilder.
- Dimma (`man glFog` så får du reda på hur lätt det är att sprida sådan)

Hårdvaruutveckling

Utvecklingen var ett tag mycket rask.

Hösten 1999 presenterades den första grafikprocessor för konsumentmarknaden med inbyggt stöd för transformationer och belysningsberäkningar GeForce. Det var något som inte ens SUNs dåtida grafikkort Creator3D hade (Elite3D har det).

Den grafikprocessor GeForce 3 som NVIDIA började leverera i april 2001 bestod av 57 miljoner transistorer (grovt sett samma som Pentium 4) och uppgavs kunna utföra 76 miljarder flyttalsoperationer på en sekund. Det är mycket mer än vad en vanlig processor klarar (kanske 1 miljard) och låter mycket. Den principiellt viktigaste nyheten var att man själv kunde programmera delar av beteendet (hörn- och fragmentprogrammering). Under år 2002 kom GeForce 4 (OBS! GeForce 4MX är egentligen en GeForce 2) med ännu högre komplexitet och bättre prestanda, liksom nya modeller med likartat funktionssätt från främst konkurrenten ATI (Radeon).

Därefter har det främst handlat om prestandaökning. Och att man funnit nya användningsområden för de olika tilläggen.

Sista bilden. Vad har delats ut?

Allt med OH finns på nätet via kurssidan. OH och övrigt kan fås av mig, så länge jag har restexemplar. Eventuellt kan jag lägga ut någon småskrift tillfälligt.

- Introduktion till OpenGL, 54 sidor. Kostar.
- Datorgrafik OH 1-335. Finns på kurssidan.
- Ett modelleringsprogram - Blender. 8 sidor. Finns på kurssidan.
- Ett annat modelleringsprogram - Art Of Illusion. 2 sidor.
- Från värld till skärm, 20 sidor.
- Kurv- och ytapproximation med polynom, 16 sidor
- PostScript - en introduktion, 12 sidor.
- Fraktaler och kaos, 12 sidor.
- Bildbehandling, 16 sidor (enbart för GU)

Totalt 124 sidor + 335 OH + 16 sidor (GU)

Feltryck

OH 257 Man måste begära Alpha-buffert, som finns på modernare kort, med GLUT_ALPHA i glutInitDisplayMode.

Tyvärr säkert fler.

Labgodkännande: Sista officiella torsdagen den 13 oktober, vilket meddelades i början av kursen. Några få kan få redovisa lab 3 under tentamensperioden. Ännu färre i första läsveckan, då jag dock förmodligen är rätt jäktad. I båda dessa fall efter kontakt med mig.

Inför tentan 1(2)

Tentans utformning: c:a 1/3 kodning, 2/3 teori. Svaren på de större teori-frågorna skall vara förklarande och begripliga för en kamrat som inte läst kursen.

Kursen har presenterats i form av OH, demonstrationer, laborationer, OpenGL-kod och småskrifter.

Demonstrationer och laborationer
Har haft som syfte att belysa annat material. Dvs inga krav på minneskunskaper.

OpenGL-kod
Du skall på ett korrekt sätt kunna tillämpa det material som finns i OpenGl-häftet (tillåtet hjälpmedel). Kodning sker i det språk som du själv valt i kursen. Smärre språkfel ger ej avdrag. När det gäller funktioner/metoder som bara tagits upp på OH, krävs inga minneskunskaper, utan i förekommande fall ges tillräcklig hjälp på tentan.

Småskrifterna
Från värld ...: Du skall kunna återge och praktiskt tillämpa materialet i avsnitten 1-4 och 8-9. Avsnittet 10 ingår bara som OH 326. Avsnitt 11 inte alls. Övriga avsnitt är exemplifierande.
Kurvor och ytor:Du skall kunna återge och praktiskt tillämpa materialet i avsnitten 1, 3-12.
Fraktaler: Du skall kunna beskriva de begrepp som införs i häftet. Avsnitt 8 ingår inte.
Postscript: Utdelat men ej diskuterat, så det får väl utgå, om det inte genom ett under blir tid över på F18.

Inför tentan 2(2)

Bildbehandling (enbart GU): Häftet ingår i sin helhet. Du bör känna till begreppet fourierkoefficient, men behöver inte kunna några formler för dem.

OH
Du skall kunna beskriva de begrepp som successivt införs. Du skall kunna återge och praktiskt tillämpa de resonemang som förs kring bl a: rastering av linjer, 2D- och 3D-transformationer, dolda ytor, navigering, fotorealism OH 92-112, uppdelningsmetoder OH 123-124, Perlin-brus OH 190-196, vertex- och fragment OH 231-239, OH 245-252 (ev program blir enkelt), marscherande kuber, billboarding, skuggor, Följande OH kan du lugnt hoppa förbi: 54-61, 130-131,167-169, 241-244, 268-269, 296-297, 313-314.