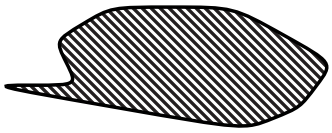
Målning

Säg att vi har ett slutet område i 2D och vill fylla det med en viss färg eller ett visst mönster. Hur gör man då? Alla ritprogram har naturligtvis en metod för det, men hur kan den tänkas se ut? Alla läroböcker brukar ta upp frågan och ägnar ett par sidor åt den.



Låt oss börja med färgfallet. Vi väljer en punkt i området och kan sedan anropa den rekursiva proceduren

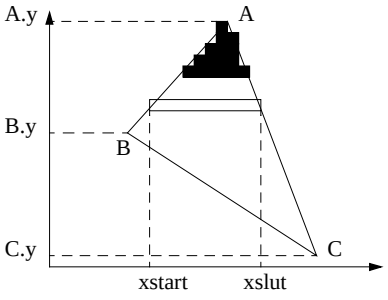
```
void fyll(int x, int y) {
    if (färgen i (x,y) inte är randens färg och inte
        är fyllnadsfärgen) {
        rita punkten (x,y);
        fyll(x+1,y); fyll(x,y+1); fyll(x-1,y); fyll(x,y-1);
    }
}
```

Rekursionsdjupet kan bli stort och metoden är inte så snabb. Den kan effektiviseras på olika sätt.

I mönsterfallet (jag tänker mig någon form av texturmönster som upprepas) kan man arbeta på en kopia av området, som fylls successivt med en genomgående färg. Varje gång en punkt (x,y) i kopian fylls i räknas motsvarigheten i texturen fram (x MOD N etc) och punkten (x,y) i originalet markeras på rätt sätt.

Rastrering

Vi har tittat på rastrering av en linje med Bresenhams algoritm. Även övriga primitiver behöver rastreras. Läroböckerna brukar gå igenom detta i detalj för godtyckliga polygoner och lyckas identifiera en del svårigheter. Men det känns inte särskilt angeläget.
Att rastrera en triangel är däremot problemfritt om än inte trivialt för en nybörjare.



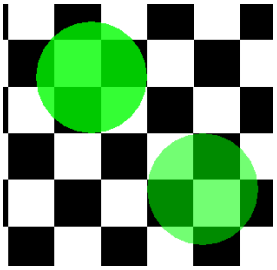
1. Sätt de aktiva kanterna till AB och AC
2. Upprepa för y = A.y, A.y-1,...,C.y
   2.1 Byt aktiva kanter om y=B.y. Räkna ut xstart och xslut. Kan ske med interpolation eller med variant av Bresenham.
   2.2 Räkna även ut färgvärden, djup, texturkoordinater etc för punkterna (xstart,y) och (xslut,y) med någon form av interpolation. Jfr "Från värld till skärm".
   2.3 För x = xstart, xstart+1, ...,xslut
   Räkna ut värden för bildpunkten (x,y) med interpolation. Nu har vi ett fragment i OpenGLs mening. Rita eventuellt (efter bl a djuptest).

Genomskinliga (transparenta) objekt

Genomskinlighet kan, som vi vet från avsnittet 28 om färgblandning i OGL-häftet, åstadkommas genom att vi ritar med ett alfa-värde som är mindre än 1. Det är dock viktigt att de genomskinliga ytorna ritas i rätt ordning, dvs att den närmsta genomskinliga ytan ritas sist.

I praktiken kan det gå till så här:

- Rita alla ogenomskinliga ytor med normalt utnyttjande av djupminnet.
- Se till att de genomskinliga ytorna är sorterade (t ex med BSP) och rita dem i rätt ordning. Använd även nu djupbufferten för att förhindra att en genomskinlig yta som döljs av en ogenomskinlig blir ritad.
  Ett fusksätt enligt OpenGL Prog Guide: Rita de genomskinliga utan sortering, men se till att djupbufferten inte **förändras** av dessa (gör anropet `glDepthMask(GL_FALSE)`; återställning till det normala med `glDepthMask(GL_TRUE)`). Fungerar helt korrekt för t ex enstaka kuber och sfärer. Högra bilden visar två genomskinliga gröna sfärer ritade framför ett schackbräde. Övre sfären ritad utan förändring av djupbufferten, dvs vi ritar två gånger per bildpunkt, undre med förändring av djupbufferten, dvs vi ritar eventuellt bara en gång per pixel. Den senare är något ljusare vilket visar att ritning bara skett en gång per pixel.



Som vanligt är den genomskinlighet som OpenGL åstadkommer halvdan. För perfekt resultat behövs strålföljning.

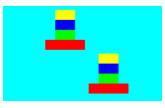
Mera OpenGL: Sprite

Ibland vill man att något litet icke-fyrkantigt objekt (populärt kallat **sprite**) skall röra sig över skärmen. Man kan använda rasterkopiering i kombination med färgblandning (jfr avsnitt 28). Med datatypen `GLbyte` betyder 127 1.0 och 0 0.0. I rastret ser jag därför till att genomskinliga punkter har alfa-värdet 0 och övriga 127.

```
void init {
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
GLbyte M[64]=
    {127,0,0,127, 127,0,0,127, 127,0,0,127, 127,0,0,127,
     0,127,0,0, 0,127,0,127, 0,127,0,127, 0,127,0,0,
     0,0,127,0, 0,0,127,127, 0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127, 127,127,0,127, 127,127,0,0};
int xpos1, xpos2, ypos1, ypos2;
void display(void) {
    glClearColor(0.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2d(xpos1,ypos1);
    glPixelZoom(8.0,8.0);
    glDrawPixels(4,4,GL_RGBA,GL_BYTE,M);
    glRasterPos2d(xpos2,ypos2);
    glDrawPixels(4,4,GL_RGBA,GL_BYTE,M);
    glPixelZoom(1.0,1.0);
    glutSwapBuffers();
}
```

Använd rasterbild (matrisen M)

Gul
Blå
Grön
Röd



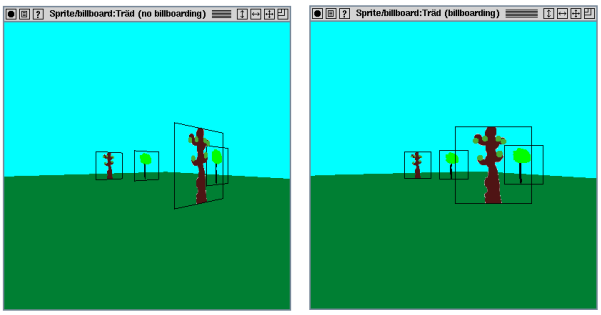
Allmännare med textur. T ex är en känd teknik att rita träd att man ritar två vinkelräta och korsande fyrkanter med en trädtextur där texlar motsvarande trädet har alfavärdet 1 och övriga 0. Mer strax.

Mera OpenGL: Billboarding 1(6)

Betrakta följande bild hämtad från DirectX SDK 9.0 och körd på en föga märkvärdig PC. Man kan röra sig i scenen i realtid. Hur går det till? Vi koncentrerar oss på träden, som är ritade med en teknik kallad billboarding (försvenskat affischering). Träden ritas som rektanglar belagda med en textur. Vi ser till att delar av texturen är genomskinlig genom att förse alla texlar med ett alfa-värde (fjärde komponenten i färgen). T ex 0.0 för genomskinliga och 1.0 för övriga. I scenen nedan används 2-3 olika trädtexturer. Även för marken/himlen används texturering.

Mera OpenGL: Billboarding 2(6)

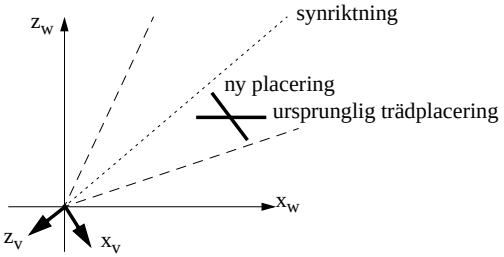
Innan man når fram till fullgod billboarding måste man lösa några problem.



Betrakta bilderna ovan. Scenen är litet enklare och vi har bara två trädtexturer (träd med rund krona respektive knotigt träd). Låt oss placera trädens kvadrater parallella med xy-planet. För tydlighets skull är kvadraternas kanter utritade. Om vi vrider oss runt y-axeln kommer det att se ut som i vänstra figuren och vi inser att i vissa lägen kommer ett träd bara att se ut om ett vertikalt streck. En lösning som antyddes tidigare är att använda två korsande kvadrater för varje träd. En annan är att se till att trädkvadraterna vid rotation följer den så att de hela tiden är vinkelräta mot användaren, se högra bilden. Detta hjälper naturligtvis inte om användaren tar sig före att flyga över träden, för då avslöjas ju fusket.

Mera OpenGL: Billboarding 3(6)

Det förståelsemässigt enklaste sättet att se till att trädkvadraterna är vända mot observatören, når vi genom att titta på följande figur, där scenen är ritad sedd uppifrån och med ett enda träd. Observatören tittar i en viss synriktning (observatörspositionen saknar betydelse för resonemanget; råkade bli origo i världen).



Trädets centrumpunkt skall vara oförändrad. Trädkvadratens över- och underkant skall vara riktad som vykoordinatsystemets x-axel (x_v). Sidokanterna får förbli vertikala (om synriktningen inte är parallell med horisontalplanet är det rimligare att låta dem vara parallella med y_v-axeln). Eftersom vi lätt tar reda på x_v-vektorn ur modellvymatrisen (som ju kan avläsas i OpenGL), kan vi räkna ut trädkvadratens nya hörn och sedan rita den som en GL_POLYGON.

Mera OpenGL: Billboarding 4(6)

Ett alternativt sätt att resonera är att direkt efter *gluLookAt* ta ut rotationsdelen av modellvy-matrisen (beskriver då hur världskoordinater övergår i vykoordinater och innehåller inga skalningar) och invertera den (vridning åt andra hållet), vilket kan göras så här.

Globalt: `GLfloat mv[16];`
I display:

```
glFloat t;
// Efter gluLookAt
glGetFloatv(GL_MODELVIEW_MATRIX,mv);
// Bara rotationsdelen
mv[12]=0.0; mv[13]=0;mv[14]=0;
// Invertera genom transponering
t = mv[1]; mv[1]=mv[4]; mv[4]=t;
t = mv[2]; mv[2]=mv[8]; mv[8]=t
t = mv[6]; mv[6]=mv[9]; mv[9]=t;
```

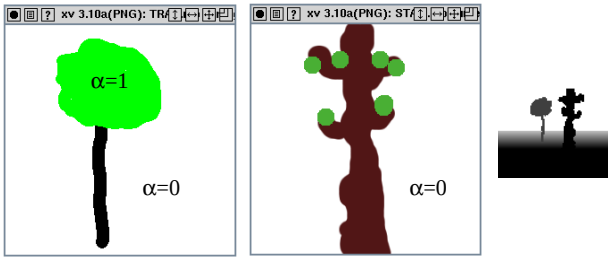
Sedan ritar vi i *display* ett enstaka träd med texturering påslagen:

```
// mittpunktens placering
glTranslatef(x,y,z);
// rotationen
glMultMatrixf(mv);
glBegin(GL_POLYGON);
   glTexCoord2f(0,0); glVertex3f(-0.5,0,  0);
   glTexCoord2f(1,0); glVertex3f(0.5, 0,  0);
   glTexCoord2f(1,1); glVertex3f(0.5, 1.0,0);
   glTexCoord2f(0,1); glVertex3f(-0.5,1.0,0);
glEnd();
```

Mera OpenGL: Billboarding 5(6)

Återstår ett problem. Texturernas genomskinlighet kan vi lösa genom att använda färgblandning (dvs glEnable(GL_BLEND)+glBlendFunc(...)). Men då visar det sig att träd längre bort ibland (beroende på ritordning) döljs av träd framför, vilket beror på att färgblandningen görs efter djuptestet (och djupskrivningen). Lyckligtvis kan man slå på ett annat test, alfa-testet, som görs före djuptestet och använda det i stället för färgblandning.

Vi ser till att den genomskinliga texturen har alfa-värdet 0.0 och att övriga delen har värdet 1.0. Slår på alfa-testet med




```
glEnable(GL_ALPHA_TEST);
```
Dessutom talar vi med
```
glAlphaFunc(GL_GREATER, 0.1);
```
om att fragment med alfa-värde större än 0.1 skall skickas vidare, medan övriga skall ignoreras. Detta gör att bara fragment tillhörande trädkrona och stam skickas vidare och utsätts för djuptest etc. Djupminnet kan se ut som i högra figuren.

Mera OpenGL: Billboarding 6(6)

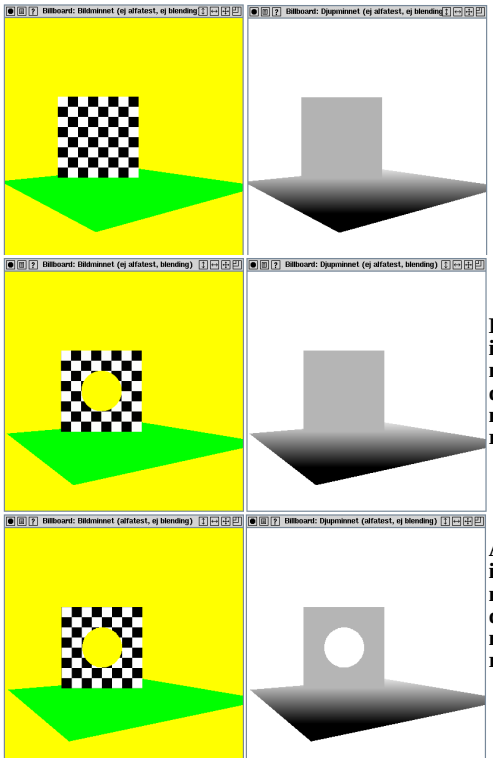
Bilderna på nästa OH har jag tillverkat genom att utöka ett program `billboard.c` gjort av David Blythe och som finns i GLUT-distributionen. Till vänster visas scenen och till höger djupminnet. Schacktexturen har alfavärdet 1 utom i en inre cirkel där värdet är 0. Normal ritning ger översta raden. Ritning med färgblandning mellersta och ritning med alfa-test understa. I det sista fallet återger djupminnet vad som verkligen ritats.

Tekniken kan användas i många andra sammanhang också, t ex i samband med partikelsystem när partiklarna är mer komplicerade än punkter eller trianglar. Det finns varianter av billboarding.

Några praktiska detaljer:

- Vi kan mycket väl läsa in en textur med bara 3 komponenter/texel och sedan ge alla texlar av en viss färg alfa-värdet 0.0 och övriga värdet 1.0. Själv använder jag funktionen *read_texture* från GLUT-distributionens `texture.c` för att läsa in texturer i form av rgb-filer. Har för mig att den automatiskt för in plats för en fjärde komponent om den inte redan finns (eller också har jag ändrat till det).
- Precis som för de vanliga basfärgerna har jag föredragit att tala om alfa-värden mellan 0.0 och 1.0. I praktiken använder vi ju `GLbyte` (C:s `signed char`) eller `GLubyte` (C:s `unsigned char`) för värdena. Härvid motsvarar `(GLbyte)127` värdet 1.0 och `(GLubyte)255` eller `0xff` värdet 1.0 medan 0 motsvarar 0.0.

Mera OpenGL: Blending/alfatest 1



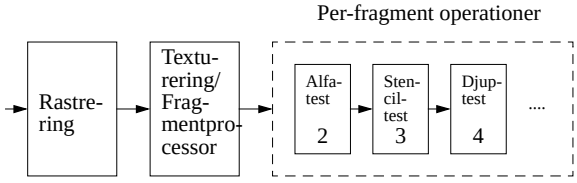
Blending: ingen ritning men djupminnet ändras

Alfatest: ingen ritning och djupminnet ändras ej

Slutet av OpenGL:s rörledning (pipeline)

Vi vet att OpenGL rastrerar, dvs bestämmer vilka bildpunkter som skall ritas och för var och en av dessa djup, färg, texturkoordinater, etc. I OpenGL kallas ett sådant paket med information om en punkt för ett **fragment**. Innan ritning - i bildminnet - sker utförs en mängd ytterligare saker, som i specifikationen kallas *Per-fragment Operations*. Dessa är i tur och ordning (enbart de i fet stil berörs under kursen):

1. Saxtestet (hindrar ritning utanför viss del av fönstret)
2. **Alfatestet** (fragment stoppas p g a sitt alfa-värde)
3. **Stenciltestet**
4. **Djuptestet**
5. **Färgblandning**
6. Dittring
7. **Logiska operatione**r (XOR etc)

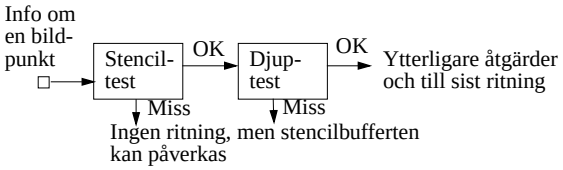


Mera OpenGL: Stencilminne 1(2)

Stencilminnet kan användas på flera sätt. Den vanligaste gäller begränsning av ritning till ett område med godtycklig form. Vi nämnde en annan i samband med BSP.

Stencilminnet kan närmast ses som ett nummerminne. Vi ritar inte i det utan placerar tal i det. Varje gång vi står i begrepp att på vanligt sätt rita i en bildpunkt görs ett test som involverar motsvarande punkt i stencilminnet. Testet kan påverka innehållet i stencilminnespunkten och det kan också förhindra ritning i det vanliga bildminnet.

Djuptestet är det kända, dvs om bildpunktens djup är mindre än det som är lagrat i djupminnet blir resultatet OK (det går att utforma testet annorlunda, men vi berör inte det).



Följande behöver göras.

Mera OpenGL: Stencilminne 2(2)

Nollställ stencilminnet: `glClear (GL_STENCIL_BUFFER_BIT);`

Slå på stenciltestet: `glEnable (GL_STENCIL_TEST);`

Definiera stenciltestet och det värde som eventuellt skall placeras i minnet:

```
glStencilFunc (villkor för OK,
               heltaligt referensvärde, 0xffffffff);
```

Villkoret kan bl a vara `GL_ALWAYS`, `GL_LESS` och `GL_EQUAL`. I det första fallet ger testet alltid resultatet OK. I det andra fallet blir det OK om referensvärdet är mindre än talet i stencilminnet.

Tala om hur stencilminnet skall påverkas vid olika testresultat:

```
glStencilOp(åtgärd när stenciltestet ger Miss,
            åtgärd när djuptestet ger Miss,
            dito när djuptestet ger OK);
```

Värdena kan vara bl a `GL_KEEP` (ändra inte), `GL_REPLACE` (ersätt värdet med det som definierats i glStencilFunc) och `GL_INCR` (öka värdet med 1).

Exempel 1: Räkna antalet ritningar per punkt

```
glStencilFunc(GL_ALWAYS, vad som helst, 0xffffffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
```

Exempel 2: Markera ett område i stencilminnet med 1:or.

```
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
Rita området (man kan förhindra att något syns på skärmen)
```

Exempel 3: Rita något på skärmen men begränsat till området bestämt av stencilminnet (se förra ex). T ex ett runt förstoringsglas.

```
glStencilFunc(GL_EQUAL, 1, 0xffffffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
Rita.
```

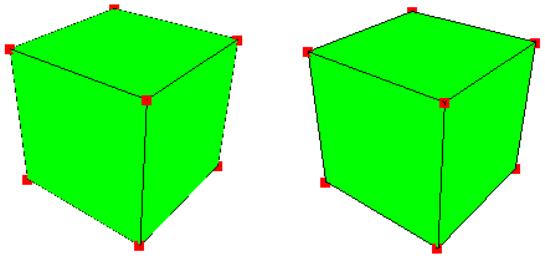

Mera OpenGL: PolygonMode och PolygonOffset

Ibland vill man att kanterna på de ritade polygonerna skall synas. Man kan då som vi gjorde i något tidigare exempel rita kanterna som linjer (GL_LINES). Men behändigare är att byta polygonritningsmod med (nämns som hastigast i OpenGL-häftets avsnitt 7)

```
                GL_FRONT_AND_BACK, GL_FILL
glPolygonMode(GL_FRONT,           GL_LINE  )
                GL_BACK,          GL_POINT
```

till GL_FRONT_AND_BACK och GL_LINE från de i fetstil markerade standardvärdena.

Men resultatet blir normalt inte riktigt som vi önskar. Se vänstra figuren. Vissa av kantlinjerna verkar streckade eller prickade. Orsaken är djupminnestekniken. Punkterna på ett objekt ritas som standard bara om de är närmre betraktaren än vad som redan ritats. Det betyder att punkterna på kantlinjerna (även om de teoretiskt skulle sig skilja sig en aning från polygonernas inre) på grund av djuprepresentationen inte säkert ritas korrekt (man kan med `glDepthFunc` ställas om till bl a "närmre eller lika", men resultatet blir ändå likartat).



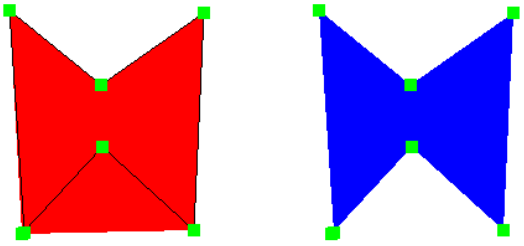
Men vi kan under linjedragningen se till att alla djupvärden minskas något (precis så mycket att de skiljer sig från det tidigare djupvärdet), dvs att punkterna kommer närmare oss. Vi gör detta med en *glEnable + glPolygonOffset*.

```
glEnable(GL_POLYGON_OFFSET_LINE);
glPolygonOffset(-1.0,-1.0);
glColor3f(0.0,0.0,0.0);
ritakub();
```

Resultatet blir den högra figuren, som är som den skall.

Mera OpenGL. Tessellering 1(2)

OpenGL klarar konvexa polygoner (glBegin(GL_POLYGON)), men inte (alla) konkava polygoner och inte "polygoner med hål". För att klara av allmännare fall finns tessellering (eng. tesselate = lägga mosaik). Detta är en "vetenskap" i sig, som dessutom visat sig vara plattformsberoende. Här tar vi bara upp en liten del av det hela



Till vänster har vi sex punkter som skall utgöra hörn i en polygon. Ordningen framgår av de heldragna linjerna. Punkterna har vi samlat i en vektor

```
GLdouble P[50][3];//ej mitt favoritsätt, men gillas av OpenGL
```

och ritningen har skett med

```
glBegin(GL_POLYGON);
      for (i=0; i<Nr_Of_Points; i++) {
         glVertex2d(P[i][0], P[i][1]);
      }
glEnd();
```

Polygonen är misslyckad eftersom resultatet skulle vara som till höger.

Mera OpenGL. Tessellering 2(2)

Tesselleringskoden (med för enbart fullständighets skull) är tekniskt lik NURBS-kod.

```
GLUtesselator *theTess;   // ett tesselleringsobjekt
```

Man gör diverse initieringar

```
theTess = gluNewTess();
gluTessCallback(theTess,GLU_TESS_VERTEX,
     glVertex2dv);
gluTessCallback(theTess,GLU_TESS_BEGIN, glBegin);
gluTessCallback(theTess,GLU_TESS_END, glEnd);
gluTessCallback(theTess,GLU_TESS_ERROR, tessError);
gluTessCallback(theTess,GLU_TESS_COMBINE, combineCB);
```

Man ritar

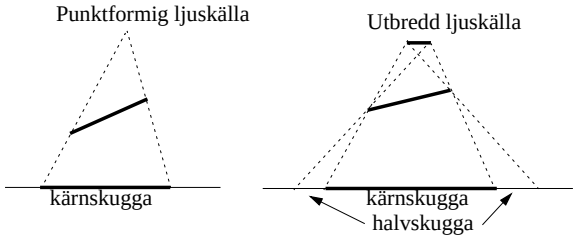
```
void Tessellera() {
      int i;
      glColor3f(0.0,0.0,1.0);
      gluTessBeginPolygon(theTess, NULL);
           gluTessBeginContour(theTess);
           for (i=0; i<Nr_Of_Points; i++) {
              gluTessVertex(theTess, P[i], P[i]);
           }
           gluTessEndContour(theTess);
      gluTessEndPolygon(theTess);
      glFlush();
}
```

Man kan behöva användarskrivna "callbacks".

```
void combineCB(GLdouble coord[3],GLdouble
  *vertexData[4], GLfloat w[4],GLdouble **dataOut) {
    GLdouble *vertex;
    vertex=(GLdouble *)malloc(3*sizeof(GLdouble));
    vertex[0]=coord[0]; vertex[1]=coord[1];
    vertex[2]=coord[2]; *dataOut = vertex;
}
```


Skuggor 1(5)

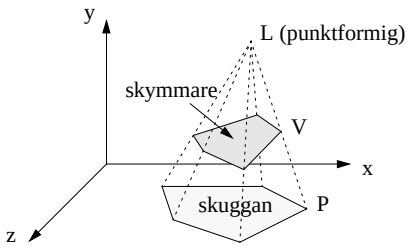
Får vi naturligtvis gratis i strålföljningsprogram, men som alltid vill vi att det skall gå fort. Punktformiga ljuskällor ger ren kärnskugga, medan utbredda ljuskällor ger mjuka skuggor bestående av en kärnskugga (eng. umbra), som inte nås av något ljus från ljuskällan och omgivande halvskugga (eng. prenumbra).



Det finns många metoder för snabb skugg-generering, men ännu ingen given segrare. Oftast blir kodningen något omständlig. Vi belyser bara området översiktligt och förbigår de flesta komplikationerna. En utmärkt sammanfattning av skuggor gjordes av Mark Kilgard vid The Game Developers Conference, 2000. Hittas via NVIDIAs sidor. Objekten i scenen måste delas upp i skymmare (eng. occluder) , som ger upphov till skuggor, och mottagare (eng. receiver), på vilka skuggor bildas.

Skuggor på plana ytor 2(5)

Enklast är det när mottagarna är plana ytor. I figuren tittar vi på ett sådant fall, där mottagaren är planet y=0 (eller möjligen y=y_0)



Vi kan gå tillväga så här:

1. Räkna ut skuggpolygonen (se nedan)
2. Rita mottagande plan. Om ej oändligt (t ex bord) fyll även motsvarande platser i stencilbufferten med 1:or.
3. Rita skuggpolygonen med offset (glPolygonOffset(GL_POLYGON_OFFSET_FILL) eller eventuellt avslaget djuptest och kanske i kombination med färgblandning. Rita bara där stencilbufferten har 1:or.

För att räkna ut ett hörn P, utnyttjar vi som ofta förr linjens parameterframställning P = L + t(V-L). Eftersom vi känner $P_y=y_0$, blir $t=(y_0-L_y)/(V_y-L_y)$, som insatt i ekvationerna för x och y ger (y_0=0 nu) $P_x=(L_xV_y-L_yV_x)/(V_y-L_y)$ och $P_z=(L_zV_y-L_yV_z)/(V_y-L_y)$. Lätt att generalisera till godtyckligt plan. Kan skrivas på matrisspråk. Skuggan beror enbart på ljuskällan och skymmaren. Skuggpolygonen behöver därför bara räknas om när dessa rör på sig.

Exempel: Programmet *projshadow.c* i GLUT-distributionen.

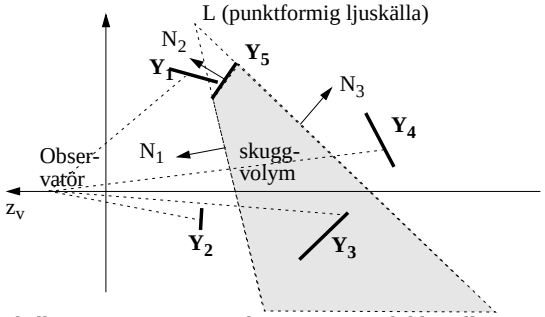
Skuggor: Med texturkarta 3(5)

Här börjar man med att rita scenens skymmare sedda från ljuskällan. Ger en bild som kan sparas som en textur (man kan rita direkt). Sedan ritas scenens mottagare med texturkartan (s k projektiv texturering som OpenGL har stöd för). Därefter skymmarna. Tekniken liknar alltså den som kan användas vid ljussättning i spel.

Exempel: Programmet *projtex.c* i GLUT-distributionen demonstrerar projektion av textur, men inte skuggdelen.

Skuggor: Skuggvolymer 4(5)

Vid sidan av skuggminne (eng. shadow map, som vi inte tar upp) är detta en huvudmetod, som dock kan utformas praktiskt på ett otal sätt. Jag tar upp en listig version som nog hörrör från SGI-kretsar och som kanske var en motivering till stencilminnet. I figuren nedan har vi en



ljuskälla L. Ytan Y_5 är en skymmare. Den bildar tillsammans med strålarna från ljuskällan en halvoändlig avhuggen sned pyramid, som kallas **skuggvolymen**. Enbart objekt helt eller delvis i skuggvolymen

skuggas av Y_5. I figurens fall ligger enbart Y_3 i skugga. För en enstaka bildpunkt ligger motsvarande punkt på den träffade ytan i skugga om strålen från observatören och “genom bildpunkten“ skär skuggvolymens sidoytor precis en gång. Med flera ljuskällor/skymmare i stället ett udda antal gånger.
Algoritm utan praktiska detaljer (1 och 4 kan efter viss modifiering kastas om):
1.Rita scenen normalt.
2.“Rita i stencilminnet“ skuggvolymens framsidor, dvs de som har normalen vänd mot observatören genom att öka motsvarande platser i stencilminnet med 1. Använd djuptest men ändra inte i djupminnet.
3.“Rita i stencilminnet“ skuggvolymens baksidor, dvs de som har normalen vänd från observatören genom att minska motsvarande platser i stencilminnet med 1. Använd djuptest men ändra inte i djupminnet.
4. Rita scenen igen (med t ex enbart omgivningsljus) med djuptestet glDepthFunc(GL_LEQUAL), men enbart om motsvarande stencilpunkt är 1.
Punkterna 1-3 gör att bildpunkten för strålen som träffar Y_4 har stencilvärdet 0, eftersom skuggvolymens fram- och baksida "ritats". Bildpunkten för strålen träffande Y_3 har däremot stencilvärdet 1 eftersom bara skuggvolymens framsida "ritats". I steg 4 kommer därför den senare bildpunkten att förmörkas.
Exempel: Programmen *shadowfun.c* ($DG/EXEMPEL_GLUT/advanced/shadowfun) och *dinoshade.c* i GLUT-distributionen. På nästa sida visas skärmdumpar från körningar av programmet *shadowfun.c* i två olika miljöer, vilket visar på ett av flera problem med denna algoritm. Genom att använda *glPolygonOffset* blir resultatet korrekt i båda miljöerna.

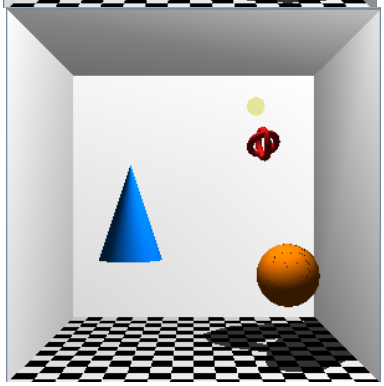
Skuggor: Varför är det på detta viset? 5(5)

Mark Kilgards program shadowfun kört på min PC eller efter ändring av MB på Sun

Ljuskällan är den gula sfär som på bilden syns nära de två ringarna

Samma program kört på Sun: ingen skugga på högra väggen eller på klotet (bara enstaka skuggprickar syns; det mörka på undersidan är belysningsmodellens förtjänst).

Orsak: Man ritar t ex golvet två gånger. Första gången normalt. Andra gången bara den del som ligger i skugga. P g a beräkningsfel klarar inte djupbufferten av att göra rätt.



```
glPolygonOffset(0.0,-1.0); glEnable(GL_POLYGON_OFFSET_FILL)
// Rita polygon GL_POLYGON, GL_QUADS etc
glDisable(GL_POLYGON_OFFSET_FILL);
```


Färg 1(3)

Varje datorgrafikbok brukar ägna ett kapitel åt detta. Ett av de bättre finns i Hills bok. Vi nöjer oss med mycket mindre.
Två huvudproblem: Färgval och färgbeskrivning. Båda är rejält komplicerade.

Färgval. Alla färger passar inte ihop. Den här diskussionen gäller inte så mycket vårt slag av 3D-grafik (med verklighetskontakt) utan främst informationsmaterial av olika slag. Ett utdrag ur en undersökning gjord av Tektronix för många år sedan med ett antal försökspersoner:

Bakgrund	Tunna linjer och text BRA	Tunna linjer och text DÅLIGT	Tjocka linjer och ytor BRA	Tjocka linjer och ytor DÅLIGT
Vit	Blått (94%) Svart (63%) Rött (25%)	Gult (100%) Cyan (94%)	Svart (69%) Blått (63%) Rött (31%)	Gult (94%) Cyan (75%)
Svart	Vitt (75%) Gult (63%)	Blått (87%) Rött (37%)	Gult (69%) Vitt (50%)	Blått (81%) Magenta(31%)
Röd	Gult (75%) Vitt (56%) Svart (44%)	Magenta(81%) Blått (44%)	Svart (50%) Gult (44%) Vitt (44%)	Magenta(69%) Blått (50%)

Beskrivning av färg. RGB är inte ett objektivt format. Det synintryck bilden ger beror naturligtvis på betraktningsförhållandena men än värre på egenskaper hos t ex den skärm som visar upp bilden. Bl a skärminställningarna och lysmaterialet har betydelse.
Det finns en objektiv färgstandard (som kan verifieras mätningsmässigt) kallad CIE-standarden (CIE=Commission Internationale de l’Eclairage) med koordinater XYZ. De normaliserade värdena x=X/(X+Y+Z) och y=Y/(X+Y+Z) representerar kromaticiteten. Man kan göra RGB-systemet mera enhetsoberoende genom att utöka

Färg 2(3)

det med två tal (vitpunkt och gamma) för den skärm som används. Härigenom kan man nämligen transformera mellan RGB och CIE. Vi avstår från alla detaljer. PNG och PDF använder denna teknik.

RGB-systemet lider också av att det inte är ett särskilt naturligt system. Det är svårt att förutse vilken färg som beskrivs.

Ett möjligen användaranpassat system är HLS-systemet som innebär att man uttrycker färgen i en dominerande färgton, en ljushet (luminans) och en mättnadsgrad. Mer om den fysiska bakgrunden senare.

RGB-systemet är ett additivt system som passar vid emitterande strålning, t ex skärmar.

CMY(K)-systemet (med basfärgerna Cyan, Magenta och Gult, K=keycolor som vanligen är blacK)) är i stället ett subtraktivt system.

Det finns många andra färgsystem, t ex används ett av färghandeln. I tryckbranschen har länge Pantone varit rådande. Här beskrivs färger med hjälp av omsorgsfullt tryckta färgkartor. En del ritprogram utnyttjar detta.

Litet fysik. Från skolan har vi lärt oss: