Texturering i JOGL (Ex. 18)

De viktigaste ändringarna beror på

- att matriselement i Java inte säkert lagras i sekvens (vilket OpenGL antar)
- att Java saknar motsvarighet till `GLubyte` (C:s `unsigned char`) för tal 0-255.

Vi får därför använda en vanlig vektor respektive datatypen **`byte`** för tal -128 till 127. Härvid betyder 0 0.0 och 127 1.0 (i min version av JOGL verkar det som om -128 också blir 0.0 och -1 blir 1.0, men det är nog en bugg; alla negativa tal borde bli 0.0).

Variabler i *MyGLEventListener*:

```
private const int TexWidth= 4;
private const int TexHeight=4;
private byte[] Image = new byte[TexHeight*Tex-
Width*3];
int[] texName = new int[2];
```

Initieringarna görs i *init*. I stället för t ex

```
Image[i,j,1] = (GLubyte)255;
```

får vi skriva

```
Image[i*TexWidth*3+j*3+1] =  127;
```

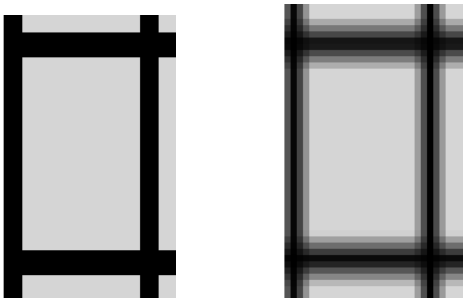
Slutligen byter vi i anropet av *glTexImage* den parameter beskriver texelns lagring i *Image* `GL_UNSIGNED_BYTE` mot `GL_BYTE`, dvs

```
gl.glTexImage2D(GL.GL_TEXTURE_2D, 0, GL.GL_RGB,
      TexWidth, TexHeight, 0, GL.GL_RGB, GL.GL_BYTE, Image);
```


Texturering - förstoring 1(2)

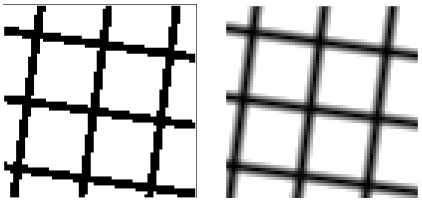
Problem: Till varje bildpunkt hör texturkoordinatvärden (t ex via mittpunkten). Det verkar då naturligt att som texel välja den som i någon mening ligger närmast dessa värden. Detta är också vad som sker då man har filtreringsegenskapen GL_NEAREST. Men vid uppförstoring, dvs då en texel berör flera bildpunkter (inträffar då **betraktaren är nära den texturerade ytan**) kan det skapa oönskad taggighet. Det gäller speciellt om texturerna utgörs av bilder med kraftig kontrast. Med GL_LINEAR kommer i stället en linjär interpolation med hjälp av fyra texlar intill den närmsta att äga rum, vilket ger en utjämnad bild.

Exempel: Vi har en rutnätstextur där varje linje är en texel bred. Vi lägger den på en yta så att varje texel täcker 4x4 bildpunkter. Till vänster visas resultatet med resultatet GL_NEAREST, till höger med GL_LINEAR.



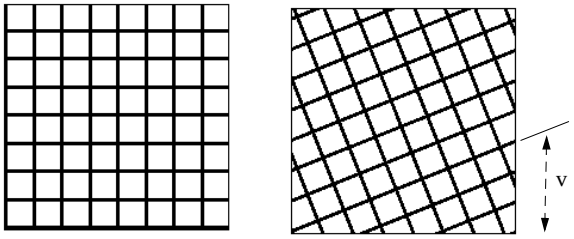
Texturering - förstoring 2(2)

Kanske tycker du att den vänstra bilden (dvs GL_NEAREST) ser bäst ut. Nyttan med GL_LINEAR i förhållande till GL_NEAREST ser vi om linjerna inte är axelparallella



I de flesta fall är en utjämning av typen till höger mest tilltalande för ögat, men det finns situationer då man vill bevara de skarpa övergångarna. För krökta ytor är det inte självklart hur matematiken bakom ser ut. Vi nöjer oss med att konstatera att det brukar fungera bra.

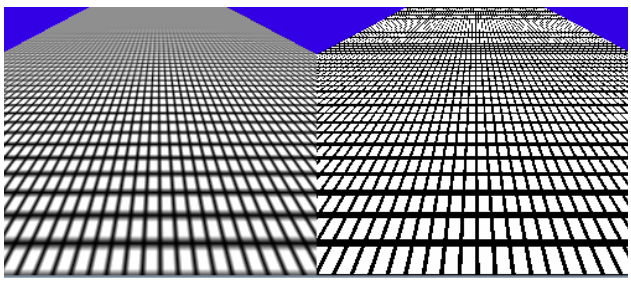
Texturering - placering av texturen



I högra figuren har vi vridit texturen vinkeln v innan den lagts på kvadraten, vilket t ex kan ske med andra texturkoordinater.

Texturering - förminskning

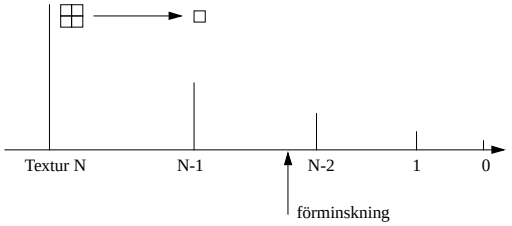
Problem: Vid förminskning kommer flera texlar att höra till samma bildpunkt. Detta inträffar **när vi tittar på en texturerad yta på stort avstånd**. Vilken texel skall då få bestämma? Vi kan ta den närmsta (GL_NEAREST), men detta blir i verkligheten ett närmast slumpmässigt val och skapar brister, speciellt om användaren rör sig i en scen. GL_LINEAR reducerar problemet bara en aning. Om bildpunkten täcker åtskilliga texlar, så är det enda vettiga att bilda ett medelvärde av alla dessas värden, men det blir kostsamt. I högra delen av figuren, ser vi ett fall utan att några åtgärder vidtagits. Till vänster har man använt s k mipmapping, som ger ett för ögat mycket mera tilltalande resultat. Bilden ritad med en modifierad variant av programmet `mipmap.c` i GLUT-distributionen.



Demo: Från S3 (öppna filen /users/course/TDA360/S3/index.htm med någon webbläsare; det sista exemplet visar mipmaps).

Mipmapping

Man bildar en hel följd av texturkartor med utgångspunkt från den ursprungliga. Om den första har dimensionen MxM ($M=2^N$) så har nästa dimensionen M/2xM/2 osv. Den sista består av en enda texel. Totalt har vi N+1 texturer. Vid halveringen slås fyra texlar ihop till en ny med medelvärdesbildning.



Vid användningen av texturkartorna räknas det fram ett ungefärligt mått på förminskningen och man går in mellan de omgivande texturkartorna och gör (bi)linjär interpolation dels inom dessa, dels mellan dem. Benämningen trilinjär interpolation står för detta.

Ordet mip kommer av latinets "Multum in parvo", som betyder mycket på ett litet utrymme. Utrymmesmässigt krävs ungefär 33 % mer än utan mipmapping.

OpenGL (enklast via GLU) stödjer mipmapping. Och de förändringar som behövs är små. Anropet av glTexImage2D ersätts av

```
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB,
    TexWidth, TexHeight, GL_RGB, GL_UNSIGNED_BYTE,
    Image);
```

och filtreringsegenskapen vid förminskning sätts med

```
glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR);
```

Det är allt! (Det finns ett par andra varianter.)

Tekannan och andra färdiga GLUT-objekt 1(2)

Litet inkonsekvent! Dessa kan delas in i plana (*glutSolidCube* m fl), andragradsytor (*glutSolidSphere* m fl) och allmänt krökta (*glutSolidTeapot*). De senare är baserade på B-splines el dyl, varigenom normaler och texturkoordinater kan genereras (med lämpliga tillstånd). Andragradsytorna är baserade på Quadrics-objekt, som vi inte tagit upp här.

	glutSolidCube &Co, glutSolidTorus &Co	**glutSolidSphere &Co**	**glutSolidTeapot &Co (FINNS EJ I JOGL nu)**
Normaler	Ja	Ja	Ja (GL_AUTO_NORMAL satt)
Texturkoordinater	Nej (man kan med möda ändra i källkoden)	Nej (man kan lätt ändra i källkoden)	Ja, lämpligt tillstånd är satt i koden
Belysning	Ja	Ja	Ja, om vi ändrar på begreppet framsida, dvs om `glFaceMode(GL_CW)`, det normala är GL_CCW.

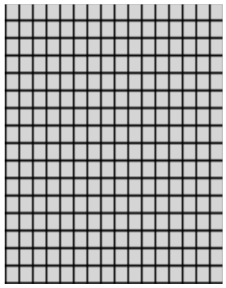
Källkoden till *glutSolidSphere* modifierad så att texturkoordinater genereras

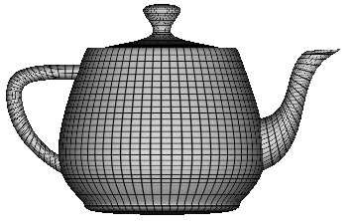
```
void glutSolidSphereMB(GLdouble radius, GLint slices,
                       GLint stacks) {
   gluQuadricDrawStyle(quadObj, GLU_FILL);
   gluQuadricNormals(quadObj, GLU_SMOOTH);
   gluQuadricTexture(quadObj,GL_TRUE); // default GL_FALSE
   gluSphere(quadObj, radius, slices, stacks);
}
```

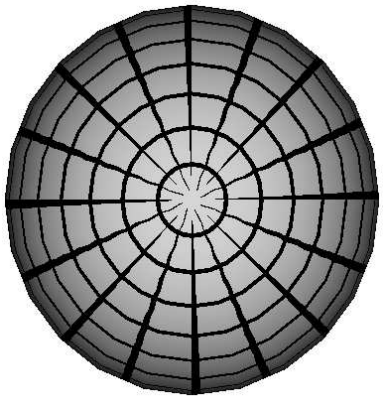
Man måste ha deklarerat `static GLUquadric *quadObj;` och före användning initierat `quadObj = gluNewQuadric();`

Tekannan och andra färdiga GLUT-objekt 2(2)

En rutnätstextur pålagd en kvadrat, en tekanna och en sfär ritad med den modifierade *glutSolidSphereMB*.







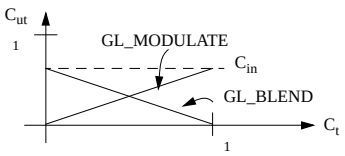
I z-led (uppifrån) Enl källkoden till gluSphere:
If texturing is turned on (with gluQuadricTexture), then texture coordinates are generated so that t ranges from 0.0 at z=-radius to 1.0 at z=radius (t increases linearly along longitudinal lines), and s ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

Texturer och ljus

I OpenGL ligger ljusberäkningen före texturpåläggningen, vilket är naturligt eftersom beräkningen gäller hörn och inte bildpunkter, men det är inte invändningsfritt. Det finns fyra storheter som påverkar hur ljusvärdena och texturvärdena kombineras. Vi skall bara beskriva dem i huvuddrag.

Vi har nämnt att GL_DECAL och GL_REPLACE bara lägger på den aktuella texelns färg och alltså struntar i belysningsvärden. Vårt hopp står då till GL_MODULATE och GL_BLEND. Låt C stå för någon av de tre grundfärgerna R, G och B. Låt C_{in} vara värde enligt belysningsmodellen, C_t texelns intensitet och C_{ut} resultatet. Då gäller för (viss skillnad om RGBA)

GL_MODULATE: $C_{ut} = C_{in}C_t$
GL_BLEND: $C_{ut} = C_{in}(1-C_t)$ (egentligen $C_{ut} = C_{in}(1-C_t)+C_tC_c$, där C_c är förvald till 0, men kan ändras med *glTexEnv*)



Eftersom alla intensiteter ligger mellan 0 och 1, kommer texturen att förmörkas vid användning av GL_MODULATE (om inte $C_{in} = 1$). Likaledes förmörkas ljusvärdet (om inte $C_t=1$). Men det värdet måste ändå bli vår favorit. Det hade varit önskvärt med fler kombinationsmöjligheter (jämför med andra kommande situationer).

Automatisk texturkoordinatgenerering

I princip har vi hittills anget texturkoordinater med någon version av *glTexCoord*. Men det finns också några mer eller mindre väl uttänkta sätt att få texturkoordinaterna automatgenererade. Den första texturkoordinaten kallas i OpenGL *s* och den andra *t* (det finns dessutom *r* och *q*). Man måste då

1. Tala om hur respektive texturkoordinat skall genereras
2. Aktivera genereringen av respektive texturkoordinat. Eventuellt explicit angivna texturkoordinater ignoreras.

Hur respektive texturkoordinat skall genereras

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR)
          GL_T,                     GL_EYE_LINEAR
          GL_R                      GL_SPHERE_MAP
                                    GL_NORMAL_MAP_ARB
                                    GL_REFLECTION_MAP_ARB
```

De två sista alternativen för tredje parametern är utvidgningar som införts i OpenGL 1.3 och som fungerar på grafikkort med bl a GeForce-processorer och nu även Sun. De tre sista (beskrivs i detalj senare) kräver att man genererar tvådimensionella texturkoordinater (`GL_S` och `GL_T`), medan för de två första det räcker med endimensionella (`GL_S`). I dessa två fall blir texturkoordinaten avståndet från den aktuella punkten till ett plan angivet antingen i modellkoordinater (`GL_OBJECT_LINEAR`) eller vykoordinater (`GL_EYE_LINEAR`).

Aktivering

T ex av s-koordinaten: `glEnable(GL_TEXTURE_GEN_S)`
Dessutom måste vi ha aktiverat texturering med `glEnable(GL_TEXTURE_1D)`(bara *s*) eller `glEnable(GL_TEXTURE_2D)`(*s* och *t*)

Avståndsstyrd texturkoordinatgenerering 1(2)

Det handlar om `GL_OBJECT_LINEAR` respektive `GL_EYE_LINEAR`.

Planet beskrivs med ett anrop av formen

```
glTexGenfv(GL_S, GL_OBJECT_PLANE,vektorvariabel)
```

respektive

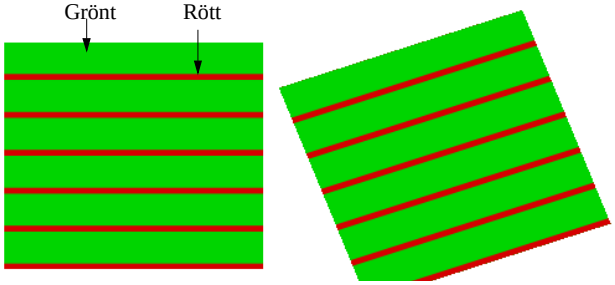
```
glTexGenfv(GL_S, GL_EYE_PLANE,vektorvariabel)
```

där vektorvariabeln innehåller de fyra koefficienterna i planets ekvation F(x,y,z) = Ax+By+Cz+D = 0. Som vanligt ger F(x,y,z) avståndet från (x,y,z) till planet om normalen (A,B,C) är normerad.

Exempel: Vi har en 1D-textur bestående av 32 texlar, där de första fyra är röda och resterande är gröna. Vi lägger den på en fyrkant vinkelrät mot planet y=0. Vi låter texturkoordinaten genereras som avståndet till planet y = 0 i **modellkoordinatsystemet**, dvs

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGenfv(GL_S, GL_OBJECT_PLANE,plane);
```

med `static GLfloat plane[] = {0.0, 1.0, 0.0, 0.0};`. Om vi vrider fyrkanten (högra figuren) följer texturen som väntat modellen.

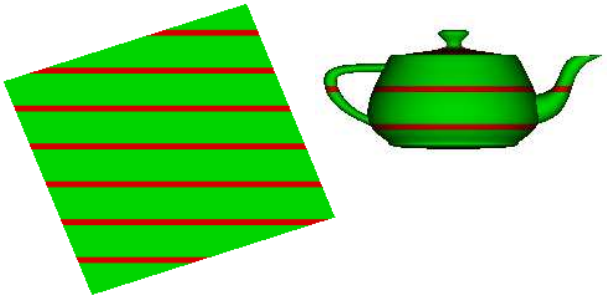


Avståndsstyrd texturkoordinatgenerering 2(2)

Om vi i stället låter texturkoordinaten genereras som avståndet till planet y = 0 i **vykoordinatsystemet**, dvs

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,GL_EYE_LINEAR);
glTexGenfv(GL_S, GL_EYE_PLANE,plane);
```

med `plane` som förut, så ligger texturen fast i vykoordinatsystemet och följer inte modellen (vänstra figuren nedan).



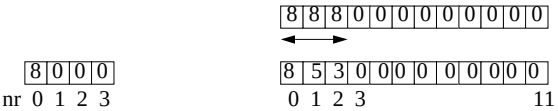
Man kan naturligtvis lika gärna lägga texturen på ett allmännare objekt, t ex den välkända tekannan (högra figuren).

Texturkoordinaterna genereras efter transformationen till vykoordinater. I fallet `GL_OBJECT_LINEAR`/`GL_OBJECT_PLANE` transformerar därför OpenGL koefficienterna till vykoordinatsystemet på det sätt som vi beskrivit i “Från värld ...”, avsnitt 8.

1D-textur

En sådan är

Låt oss för att en aning illustrera den matematiska skillnaden mellan `GL_NEAREST` och `GL_LINEAR` anta att svart har värdet 8 och vitt värdet 0 (det riktigare är ju 0 och 255 eller 0.0 och 1.0). Då utgörs texturen av vänstra figuren nedan. Om vi applicerar den på en horisontell linje med 12 bildpunkter så skulle `GL_NEAREST` ge resultatet överst till höger, medan `GL_LINEAR` ger ungefär undre figuren.



Matematiken bakom skulle kunna vara som följer. Om antalet texlar i texturen är T och antalet bildpunkter är B, så har vi följande samband mellan texelnummer *s* och bildpunktsnummer *x*:
s = Tx/B, dvs i figurens fall s = x/3. Om vi betraktar *s* som ett reellt tal s = i + a, där *i* är heltalsdelen och *a* bråkdelen, har vi följande möjliga medelvärdesbildning (t_i = intensitet i texel *i*)

intensitet i bildpunkt x = $(1-a)t_i + at_{i+1}$.

som ger värdena i figuren.

```
x   s  i  a  intensitet
0      0  0  8
1      0 1/3 16/3≈ 5
2      0 2/3 8/3≈3
```

Textur från fil 1(4)

I allmänhet läser man väl texturkartan från en fil. Om filen bara skall uppfattas som en följd av bytes (vilket gäller t ex textfiler) är det lätt att skriva en läsprocedur. T ex för det byte-baserade formatet PPM P6 (`$DG/EXEMPEL_MB/ReadPPM_P6_Texture.c`)

```
GLubyte* ReadTexture(char* filename, int* width,
                     int* height) {
   int i,j, w, h; GLubyte *ptr, *start;
   // Open file
   FILE *texfile = fopen(filename,"r");
   if (!texfile) { printf("No such file\n"); exit(1)}
   // Ignore first row P6
   for (i=1; i<=3; i++) getc(texfile);
   // Read width and height
   fscanf(texfile, "%d %d", &w, &h);
   *width = w; *height = h;
   ptr = malloc(w*h*3); start = ptr;
   // Ignore last line
   for (i=1; i<=5; i++) getc(texfile);
   // Read the texture
   for (i=0; i<h; i++){
        for (j=0; j<w; j++){
             *ptr=(GLubyte)getc(texfile); ptr++;
             *ptr=(GLubyte)getc(texfile); ptr++;
             *ptr=(GLubyte)getc(texfile); ptr++;
        }
   }
   printf("Texturen läst\n") fclose(texfile);
   return start;
}
```


Textur från fil 2(4)

PPM-formatet (Portable PixMap) utgörs av en följd av bytes, som om texturen är 256x256 inleds med 15 bytes (tecken) (\n är ett tecken, tecknet för nyrad)

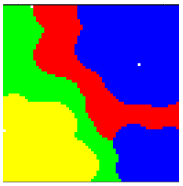
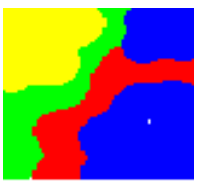
`P6\n256 256\n255\n`

varpå följer 3 bytes med RGB för översta vänstra punkten och sedan motsvarande för övriga punkter rad för rad. Första 256 avser bildbredden, andra bildhöjden. Avslutande 255 anger maximalt färgvärde. Det kan finnas varianter (t ex lägger *xv* in en kommentarrad om cirka 60 tecken efter P6-raden. Du kan titta på inledningen av en PPM-fil med ett vanligt redigeringspreogram. Inleds filen med P4 eller P5, så är det fråga om variantformat PBM resp PGM.

Funktionen *ReadTexture* används typiskt så här

```
GLubyte* Image;
GLint texwidth, texh;
Image = ReadTexture("brick.ppm", &texwidth, &texh);
printf("Texturen %d x %d\n", texwidth, texh);
```

Texturer lagras nedifrån i OpenGL, dvs en textur som i bildbehandlingsprogram som *xpaint* eller *xv* ser ut som till vänster blir med vår funktion upp och nedvänd som till höger.

Detta kan vi lätt bota, vilket vi visar i en motsvarande Java-metod.

Textur från fil 3(4)

För enkelhets skull antar jag att texturstorleken är känd och att vi har en global variabel *Image* där texturen skall placeras.

```
void readImage(String filnamn) {
   try {
      FileInputStream fin = new FileInputStream(filnamn);
      int i, p = 0, j, r = 1;
      byte b;
      // Assuming size nxn with 10<n<100
      // Skips the 13 characters P6\n64 64\n255\n
      for (j=1; j<=13; j++) { i = fin.read(); b = (byte)i; }
          while ((i = fin.read()) != -1) {
              b = (byte)(i/2);
              Image[(TexHeight-r)*TexWidth*3+p]= b; p = p+1;
              if (p==3*TexWidth) { p = 0; r=r+1;}
              //Image[p] = b; p = p+1;
           }
       }
       catch (FileNotFoundException e) { System.exit(1); }
       catch (IOException e) { System.exit(1); }
}
```

Javas `(byte)i` omvandlar heltal enligt

i	0	127	128	255
(byte)i	0	127	-128	-1

vilket gör att såväl 255 som 127 ger full intensitet (åtminstone i JOGL enl tidigare), vilket naturligtvis inte är så lyckat. Därför `i/2` i koden.

Textur från fil 4(4)

Det finns många olika sätt - format - att lagra en bild som t ex en texturkarta på. Vi återkommer till det i slutet av kursen. Bildbehandlingsprogrammet *xv* (under Linux/UNIX) kan användas för att konvertera mellan de vanligaste. *GIMP* (för Linux/UNIX eller PC) klarar också det. Formatet `.ppm` (Portable PixMap) är transportabelt, dvs kan läsas med program skrivet i datoroberoende källkod (t ex med den ovan).
I GLUT-distributionen ingår en procedur (finns i filen `progs/advanced/texture.c`) för läsning av `.rgb`- och `.bw`-format, vilka härrör från Silicon Graphics. Det senare enbart för svartvitt.

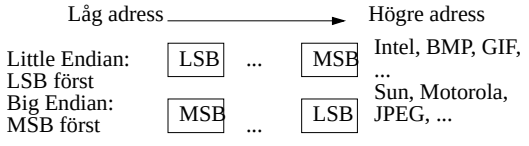
```
unsigned * read_texture(char *name,
           int *width, int *height, int *comps)
```

Koden är utformad så att den fungerar oberoende av dator. Typisk användning

```
int texwidth, texheight, texcomps; GLubyte *image;
image=read_texture("brick.rgb",
              &texwidth,&texheight,&texcomps);
```

Vi får reda på texturens bredd och höjd, liksom antalet bytes per texel (3=GL_RGB, 4=GL_RGBA). Behöver alltså inte införa någon matris! Även ett antal bilder på något av dessa två format medföljer.

Innehåller filen heltal eller flyttal på binärform blir det besvärligare åtminstone om man vill ha datoroberoende kod. Förklaringen är att olika datorer/format lagrar bytarna i tal i olika ordning (Sun: Big Endian, PC: Little Endian). Kod på nätet är oftast utformad för PC-miljön.



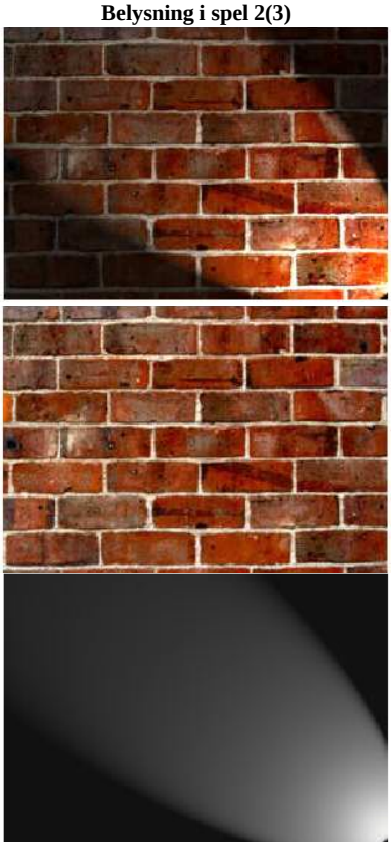
Belysning i spel 1(3)

Säg att vi belyser en tegelvägg med en lampa. Då vill vi ha ett resultat liknande det i översta figuren på nästa OH (hämtad från NVIDIAs promotionsmaterial). Kan vi få det med OpenGL? I första ansatsen räknar man ut ljusvärden i väggens fyra hörn. Även om dessa skulle bli de korrekta, så ger en efterföljande interpolering (Gouraud) ett föga realistiskt resultat. För att få något bättre måste vi dela in väggen i flera mindre fyrkanter och göra ljusberäkning för var och en av dessa. Finns något annat sätt?

Inspirerade av framgångarna med texturer införde man i spelet Quake **ljuskartor** (eng. light map), som är texturkartor med ljusmönster. Längst ned på nästa OH hittar vi en sådan. Ljuskartan byggs upp i förväg (kan vara handtillverkad eller framräknad med stor omsorg). Vid ritningen kombineras ljuskartan med tegeltexturen (texel för texel och därmed pixel för pixel), se andra figuren, innan ritning sker. Med denna teknik kan övertygande effekter åstadkommas. I figurens fall har kartvärdena multiplicerats med varandra, men man kan också tänka sig operationer som + eller någon allmän blandning.

Detta har i sin tur lett till ett begrepp som **multi-texturering**, som innebär att den slutliga texturen bestäms av ett antal olika. För att få snabbhet har de senare grafikkorten allt bättre hårdvara för arbete med sådan texturering. Man har också infört ytterligare möjligheter till automatisk texturgenerering som gör att man med få ljuskartor även kan hantera dynamiska lampor. NVIDIA talar nu om per-pixel-lighting i ställer per-vertex-lighting som är det vanliga i OpenGL. Och nog har man lyckats skapa många imponerande ting. Men det krävs att man har ett grafikkort som stöder dessa utvidgningar av OpenGL.

Belysning i spel 2(3)



Belysning i spel 3(3)

Har man sagt A och B, skall man kanske även säga C även om jag går händelserna i förväg. Här följer det gamla sättet att lägga på ljuskartor.

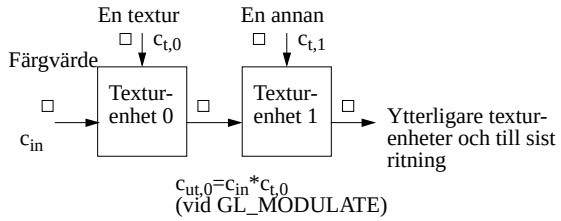
```
// Först ritar vi kvadraten med tegel utan ljus
// Talet 5 ger repetition av texturen
glBindTexture(GL_TEXTURE_2D, texName[tegelnr]);
glColor3f(1.0, 1.0, 1.0); //Bara om vi använder GL_MODULATE,
                          //lägrevärden skulle förmörka
glBegin(GL_QUADS);
     glTexCoord2f(0, 0); glVertex2f(-1, -1);
     glTexCoord2f(5, 0); glVertex2f(1, -1);
     glTexCoord2f(5, 5); glVertex2f(1,  1);
     glTexCoordf(0, 5);  glVertex2f(-1,  1);
glEnd();

// Sedan är det dags att byta till ljuskartan och se till att
// värdena blandas på lämpligt sätt med det redan ritade
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO, GL_SRC_COLOR); // Ger tegel*ljuskarta
glDepthFunc(GL_LEQUAL); // Ser till att det ritas även om
     // vi redan ritat på just det avståndet; finns
     // andra sätt
glBindTexture(GL_TEXTURE_2D, texName[ljusnr]);
// Plats för extranummer
//Och så ritar vi kvadraten med enbart ljuskartan
glColor3f(1.0, 1.0, 1.0); // Onödig upprepning
glBegin(GL_QUADS);
     glTexCoord2f(0, 0); glVertex2f(-1, -1);
     glTexCoord2f(1, 0); glVertex2f(1, -1);
     glTexCoord2f(1, 1); glVertex2f(1,  1);
     glTexCoord2f(0, 1); glVertex2f(-1,  1);
glEnd();
```

Extranummer(låter oss flytta ljustexturen**)**:

```
glMatrixMode(GL_TEXTURE);glPushMatrix();glTranslatef(tx, ty, tz);
... och efter glEnd(): glPopMatrix(); glMatrixMode(GL_MODEL_VIEW);
```


Multitexturering 1(2)

Detta är ett exempel på en utvidgning som kommit in i specifikationen för OpenGL 1.3 (aug 2001). Från våren 2004 kan vi använda det på våra SUN-ar. Syftet är att kunna använda flera texturer samtidigt utan att som vi tidigare (OH om belysning i spel) gjorde skicka samma geometri flera gånger. Man har infört ett antal texturenheter (minst två) numrerade 0 och uppåt, som har sina egna egenskaper (eget påläggningssätt, egen texturmatris, eget igång-tillstånd etc). Dessa verkar i följd, vilket med modern teknologi och lämpligt matningssätt innebär att de inte kostar något tidsmässigt. Så här fungerar det:



Den tidigare koden (se OH 163) förenklas. Vi placerar geometrihanteringen i en separat procedur `wall` (suffixen `ARB` och `_ARB` kan tas bort).

```
void wall() {
    glBegin(GL_QUADS);
      // Första hörnet, ett texturkoordinatpar för
      // tegelväggen (texturenhet 0), ett annat för
      // ljuskartan (texturenhet 1)
      glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0, 0);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0, 0);
      glVertex2f(-1, -1);
```

Multitexturering 2(2)

```
      // Övriga hörn, 5 ger upprepning av tegelmönstret
      glMultiTexCoord2fARB(GL_TEXTURE0_ARB,5, 0);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1, 0);
      glVertex2f(1, -1);
      glMultiTexCoord2fARB(GL_TEXTURE0_ARB,5, 5);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1, 1);
      glVertex2f(1,  1);
      glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0, 5);
      glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0, 1);
      glVertex2f(-1,  1);
    glEnd();

}
```

Och i omritningsproceduren `display()`:

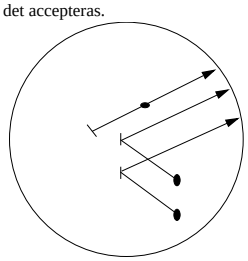
```
  // Med glActiveTexture bestämmer vi vilken texturenhet
  // vars tillstånd skall förändras
  // Först texturenhet 0 som får ha hand om tegelmönstret
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glBindTexture(GL_TEXTURE_2D, texName[tegelnummer]);
  glEnable(GL_TEXTURE_2D);
  // Sedan texturenhet 1 som hanterar ljuskartan
  glActiveTextureARB(GL_TEXTURE1_ARB);
  glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
  glBindTexture(GL_TEXTURE_2D, texName[ljusnr]);
  glEnable(GL_TEXTURE_2D);
  // Extranumret
  glMatrixMode(GL_TEXTURE);
  glPushMatrix();
  glTranslatef(tx, ty, tz);
  wall();
  glPopMatrix(); // texturmatrisen
  glDisable(GL_TEXTURE_2D);
  glActiveTextureARB(GL_TEXTURE0_ARB);
  glDisable(GL_TEXTURE_2D);
```


Environment mapping (omgivningsavbildning)

Syfte: Återge omgivningens reflektion i ett reflekterande föremål utan att ta till strålföljning.

Metod: Skapa en projicerad platt (2D) bild av omgivningen. Lagra den som en textur. När vi vill rita objektet så låter vi reflektionsvektorn ge upphov till texturkoordinater och applicerar texturen (som vanligt bildpunkt för bildpunkt).

Tanken bakom: Om det reflekterande objektet är litet och långt ifrån omgivningen så bestämmer reflektionsvektorn vad som syns i en punkt. Objektet får inte heller reflektera sig självt, dvs det måste vara konvext. Även om förutsättningarna inte är uppfyllda kan resultatet bli sådant att det accepteras.



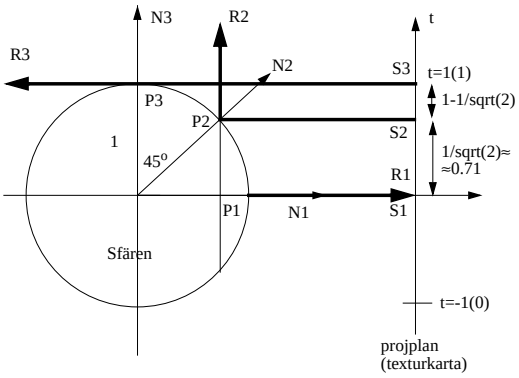
Det finns ett antal olika sätt att göra projiceringen.
1. OpenGL (och hårdvaran) ger stöd för omgivningsavbildning med sfär. Några OH beskriver detta sätt, men dessa kan till största delen överhoppas eftersom vi från hösten 2004 kommer åt omgivningsavbildning med kub (se nästa punkt).
2. OpenGL 1.3 (och någorlunda moderna grafikkort) ger stöd för dito med kub, som är en betydligt bättre metod.
3. En 5-6 år gammal metod med paraboloider är lovande och kanske ges hårdvarustöd (finns delvis). Vi tar ej upp den mera här.

Man kan ha ett antal invändningar mot såväl metoden i sig som de de enskilda projiceringssätten.

Demo: $DG/DEMOS/CUBEMAP/cubemapMB -i eller äldre `glut-3.7/progs/spheremap/glsmap` eller `glut-3.7/progs/advanced97/usespheremap` eller`ADV97_usespheremapMB2` eller `ADV99_spheremapMB`+data

Omgivningsavbildning med sfär 1(3)

Det finns flera sätt att plana ut en sfärisk yta. T ex kan man utgå från en vanlig parameterframställning av en sfär. Ett mindre känt sätt är det som beskrivs nedan och har stöd i OpenGL (och av grafikkort).

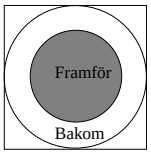


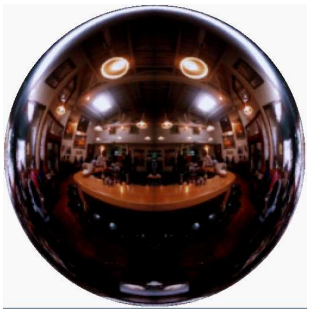
Den sfäriska texturen som är en enhetscirkel i en texturkarta utgörs helt enkelt av den bild vi ser då vi tittar på en högreflekterande enhetssfär ortografiskt. Sfären tänkes placerad med mittpunkten mitt i det reflekterande objektet.

Betrakta figuren ovan. Sfären finns till vänster och betraktaren till höger. Allt utspelar sig i vykoordinat/ögon-koordinatsystemet, dvs y-axeln är uppåtriktad och z-axeln är riktad mot betraktaren. En tänkt stråle S1 träffar sfären i punkten P1 och reflekteras tillbaka. Det som syns i P1 placeras alltså mitt i texturkartan (projektionsplanet). Strålen S2 motsvaras av en reflektionsvektor som R2 som går rakt upp. I texturkartan vid S2:s utgångspunkt placeras alltså det som finns rakt ovanför sfären. Strålen S3 går precis förbi sfären och bör alltså motsvaras av det som finns bakom sfären. Mellan utgångspunkterna

Omgivningsavbildning med sfär 2(3)

för strålarna S1 och S2 lagrar vi den delen av världen som finns till höger om sfären. Mellan S2 och S3 det som finns "bakom" sfären (till vänster om den). De båda delarna får väsentligt olika utrymme. Vi ser att olika reflektionsriktningar motsvarar olika punkter i texturen.





I figuren till vänster visas den sfäriska texturkartan symboliskt. Till höger visas en riktig (kafé Verona i Palo Alto, Calif).

Hur bildas den sfäriska kartan? Det finns minst tre sätt.
1. Genom fotografering av en sfär placerad i en verklig miljö.
2. Genom strålföljning.
3. Genom att göra lämpliga transformationer av kubavbildning.

Vid uppritningen genererar OpenGL texturkoordinater per hörnpunkt (motsv) anpassade för den sfäriska texturen förutsatt att vi gjort följande anrop

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Detta görs dock någorlunda korrekt bara för samma betraktningsrikning som användes då texturen bildades. Perspektiv får gärna vara påslaget. OpenGL har tillräcklig information för att beräkna synstrålen mellan betraktaren och hörnet (ögat är ju givet). Eftersom

Omgivningsavbildning med sfär 3(3)

hörnet måste vara försett med normal (som automatiskt omräknas till vykoordinater), kan systemet även beräkna en normaliserad reflektionsvektor R = (R_x,R_y,R_z) i vykoordinatsystemet. Texturkoordinaterna (s,t) beräknas slutligen med

$$L = \sqrt{R_x^{\,2} + R_y^{\,2} + (R_z + 1)^2}$$

$$s = 0.5\left(\frac{R_x}{L} + 1\right)$$

$$t = 0.5\left(\frac{R_y}{L} + 1\right)$$

Utan additionerna +1 och muliplikationerna med 0.5 hamnar (s,t) på en cirkel med radien

$$\frac{R_x^{\,2} + R_y^{\,2}}{R_x^{\,2} + R_y^{\,2} + (R_z + 1)^2} \le 1$$

och mittpunkt i origo. Termen $(R_z+1)^2$ i uttrycket för L gör således att reflektionsvektorer med olika z-komponent får olika texturkoordinater. De nämnda operationerna gör att den i stället hamnar inom det normala texturområdet $0 \le s,t \le 1$.

Vektorstyrd texturkoordinatgenerering

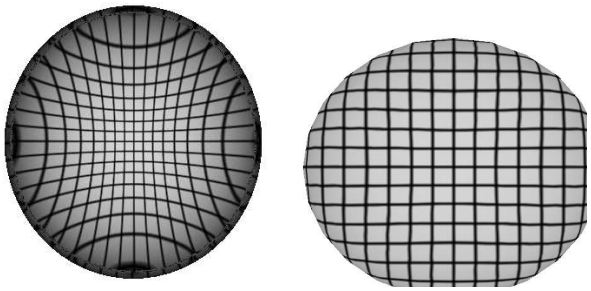
Det är vad som åstadkommes med parametervärdena (suffixet _ARB behövs inte)

`GL_SPHERE_MAP`

`GL_NORMAL_MAP_ARB`

`GL_REFLECTION_MAP_ARB`

Vi har redan avhandlat `GL_SPHERE_MAP`. Här kommer dock två bilder till. Vi har lagt vårt kvadratiska rutnät på en sfär. Högra bilden är gjord med perspektiv och återger mest korrekt den givna texturen. Den vänstra är med ortografisk projektion och förvränger givetvis på grund av texturkoordinatgenereringen mönstret.

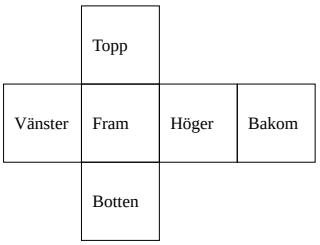


De två andra fallen styr genereringen av tre texturkoordinater (s,t,r) med hörnets normal respektive reflektionsvektor (samma som med `GL_SPHERE_MAP`), dvs även glEnable(GL_TEXTURE_GEN_R) och `glTexGeni(GL_R, GL_TEXTURE_GEN_MODE,...)`måste vara med. En tillämpning kommer på nästa OH.

Omgivningsavbildning med kub 1(2)

Sfärtexturen måste förnyas om observatören byter plats (rörelse i synriktningen må passera). Det vore önskvärt att ha ett sätt som är oberoende av var vi finns. Ett sådant sätt är kubavbildning.

Man placerar en enhetskub med mittpunkt i det reflekterande objektet. Man projicerar den omgivande världen på vart och en av de sex begränsningsplanen. Det kan ske genom att vi ritar upp scenen med sex olika 90^o-iga synpyramider. Detta görs en gång för alla.

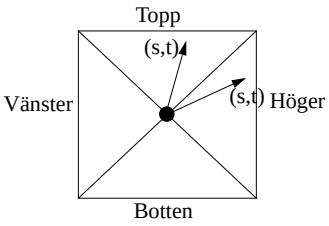


Vid uppritningen gäller det återigen att givet reflektionsvektorn i ett hörn beräkna vilken av de sex texturerna som berörs och koordinater inom den. NVIDIA introducerade hårdvarustöd för detta hösten 1999, vilket gör att sådan avbildning kan göras i realtid. På våra SUN-ar görs det i mjukvara och blir därmed långsamt (går t o m något snabbare med Mesa). I rum 6220 går det från hösten 2004 undan!

Man använder alltså moden `GL_REFLECTION_MAP` för generering av s,t,r-koordinater. Vektorn (s,t,r) är reflektionsvektorn i vykoordinatsystemet. Denna vektor används för val av en av de 6 texturerna och

Omgivningsavbildning med kub 2(2)

sedan indexering i rätt. Låt oss bara antydningsvis belysa detta för 2D-fallet då vektorn är (s,t). Om s och t positiva, väljs höger sida om s>t, etc.



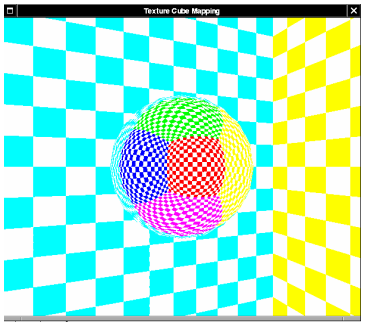
Rent praktiskt i OpenGL

```
glEnable(
  GL_TEXTURE_CUBE_MAP_ARB);
```

och för varje kubsida enligt modellen

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB,...
```

Exempel: `cubemap.c` (i Mesa-distributionen). En reflekterande sfär



i ett rum med rutiga väggar. **Demo**: Bubbles från NVIDIA om jag bara hade kunnat visa det.

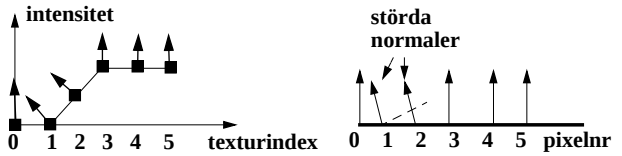
Bump-mapping 1-2(2)

Syfte: Med texturering ökar vi realismen. Men det finns mer att önska sig. Ett äpple är rätt slätt, men det gäller inte en apelsin som har små gropar (eng. bump). Även tegelmurar och gräsmattor är “gropiga”. Vi vill alltså ge intryck av gropighet eller reliefstruktur utan att detaljmodellera.

Metod: Normalerna störs med utgångspunkt från t ex en höjdkarta lagrad som en textur. Man får på det viset en störd normal N’ för varje bildpunkt och räknar med hjälp av den ut ljusvärdet per bildpunkt i stället för som normalt per hörnpunkt. Förfaringssättet är kostsamt. Men med nyare processorer går det som en dans.

På nästa OH finns tre bilder kring bumpmapping som fungerar tryckttekniskt (OBS! Ej avsedda som reklam).

Överst syns resultatet. Nedtill till höger finns den grundläggande texturen. Till vänster finns den textur, en höjdkarta (vitt=hög höjd, svart = låg), som styr bump-mappingen. Låt oss försöka beskriva idén i en endimensionell variant. Givet höjdkartan (vänstra fig) kan vi beräkna normaler i varje texel. När vi sedan lägger den grundläggande texturen på vår yta (här för enkelhets skull ett plan och 1-1 pixel/texel, högra fig) stör vi vid belysningsberäkningen ytans normaler proportionellt mot den horisontella komponenten av normalen i vänstra fig. Detta gör att ytan illusoriskt upplevs brant (streckad linje i högra fig).



3D-texturer

Våra föremål är i allmänhet 3-dimensionella. Ibland är texturerna på angränsande ytor av litet olika typ men ändå måste passa ihop. Ett typiskt exempel är en träkloss, där vi på en sida kan se ringar, men på en annan bara den längsgående ådringen.

Lösningen på detta problem är 3D-texturer. Man talar även om solid texturering. Man kan tänka sig två varianter:

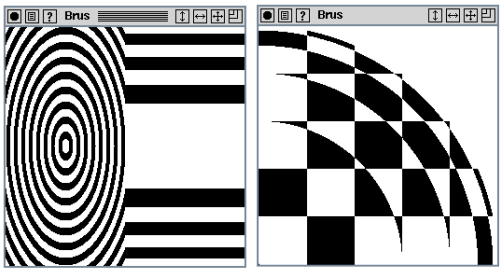
1. Ett antal lager av 2D-texturer. Om varje 2D-textur är på 256x256 skulle man kanske ha 256 lager. Utrymmesbehovet växer således krafigt och knappast något grafikkort skulle kunna hålla allt hos sig. Visst kan man minska på texturupplösningen, men då missar man ju detaljer. T ex passar tekniken inte för träklossar. Inte heller eventuell komprimering förbättrar situationen tillräckligt. Men för enklare texturer går det ju bra.
   OpenGL 1.2 och 1.3 (som finns på våra SUN’ar) har stöd för denna teknik. I princip behöver man bara byta ut konstanten `GL_TEXTURE_2D` mot `GL_TEXTURE_3D` genomgående och anropa `glTexImage3D` i stället för `glTexImage2D`. Härvid anger man förutom texturbredd och texturhöjd även texturdjup . Dessutom måste man naturligtvis nu ange tre texturkoordinater för varje hörn, vilket sker med `glTexCoord3f(...)`.
   OpenGL-stödet kan också användas för s k volymvisualisering, dvs visualisering av 3D-mätdata.
2. En annat sätt är att generera erforderlig texturinformationen vid behov. Kort sagt när vi behöver texturen för en texturkoordinat (s,t,u) anropar vi en funktion med punkten som parameter. Funktionen räknar ut aktuellt värde. Detta kan innebära ett omfattande räknearbete och metoden passar inte realtidskrav. Man brukar tala om **procedurtexturer** (eng procedural textures). OpenGL ger inget som helst stöd. Används av alla avancerade strålföljare. I framtiden kanske den här typen av beräkningar kan utföras av grafikprocessorn (eller någon sidoprocessor till den).
   Detta sätt kan användas även för 2D-texturer. Ibland kallar man texturer framställda så här för **syntetiska texturer**.

Ofta lagras inte höjdkartan utan i stället motsvarande beräknade normalkarta.
En komplikation är att om ytan är krökt eller ligger snett, måste texturnormalerna transformeras till ytans tangentplan.

Procedurtexturer

Ex. 1: Trätextur. Vi tittar mot en kant på en kub. Vänstra figuren. Här liksom i nästa exempel ortografisk projektion.

Ex. 2: Blocktextur. Vi tittar mot en sfär. Högra figuren.



För trätexturen användes proceduren (funktionen)

```
float wood(int frekvens, float x, float y, float z) {
   float r;
   r = sqrt((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5));
   return ((int)(frekvens*r))%2;
}
```

med frekvensen = 32. Den ger 0 eller 1, som får styra vitt/svart. För blocktexturen

```
float block(int frekvens, float x, float y, float z) {
   int f = frekvens;
   return ((int)(f*x)+(int)(f*y)+(int)(f*z)) % 2;
}
```

med frekvensen = 5. Även nu 0 eller 1 som resultat. Vi är inte riktigt nöjda. Vi behöver litet slump för att det skall bli mera verklighetsnära.