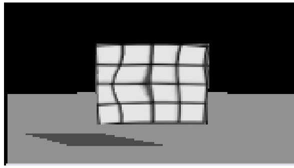
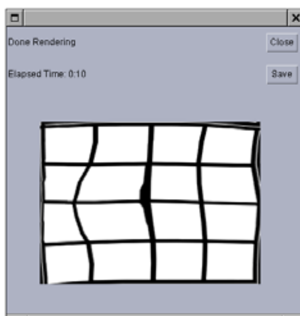
Motsvarande högra har (1.125, -1.695) och övre vänstra (-2.125, 0.695), dvs avståndet i U-led är 3.25 och i V-led 2.39. Verkar som om programmakaren begett ett tankefel.

Vi kan nu enkelt ändra texturkoordinater genom att flytta markerade punkter i texturen. Detta görs i allmänhet för att få små korrigeringar i vissa områden. Men vi försöker i stället se till att den ursprungliga texturen brer ut sig över hela ytan. Efter det tålamodsprövande flyttandet av 25 punkter ser "preview"-delen ut så här:

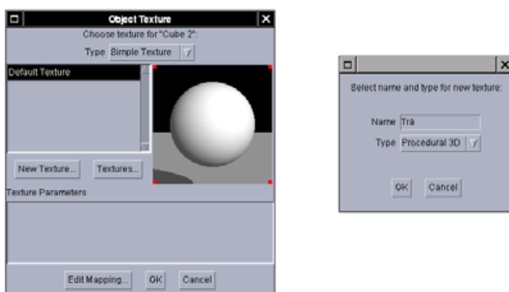


Tryck på minst två OK-knappar och vi är tillbaka i AoI-fönstret. Rendering ger

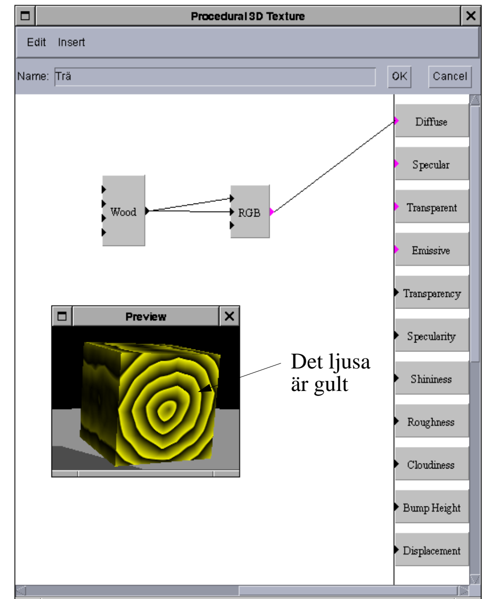


Texturer i Art Of Illusion: Procedurtextur. 1(4)

Man arbetar med bl a svarta lådor på ett sätt som är vanligt och som du kanske mött i något annat program. Vi skall nu knyta en trästruktur till en kub. Vi skapar först en kub och ser till att den är vald. Därefter **Object/Set Texture** (vänstra figuren). Med knappen **New Texture** får vi en ny dialogruta (högra figuren), där vi namngivit texturen med *Trä* och bytt från **Uniform** till **Procedural 3D**.



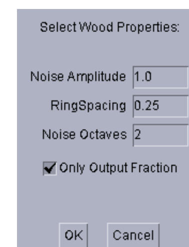
Efter **OK**, ser det ut som på nästa sida, bortsett från att arbetsytan från början är tom. Jag har med **Insert/Color Functions/RGB** och **Insert/ Patterns/Wood** lagt in två boxen RGB och Wood. Utgången på Wood har kopplats ihop med R-ingången och G-ingången på RGB och RGB-utgången har sammanbundits med ingången på Diffuse. Man ser hela tiden ett Preview-fönster av samma typ som tidigare (med MK3 har jag bytt från Sphere till Cube).



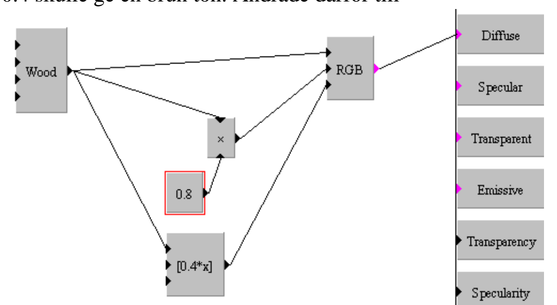
En box har ett antal inportar och normalt en utport. Vad dessa står för kan få reda på genom med godtycklig musknapp markera porten ifråga. Man får då också reda på standardvärdet. T ex ger utporten på RGB-boxen som väntat Color och den översta inporten R(0), som betyder att porten avser rödvärdet och att detta som standard är 0. För

Wood-boxens översta inport får man X(X), som betyder att värdet normalt är x-värdet i modellkoordinatsystemet. Understa blir Noise(0.5), vars tolkning inte är lika uppenbar.

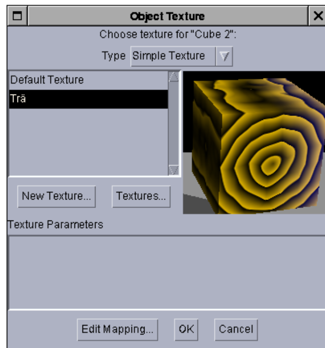
Vissa boxar är redigerbara. Klicka i så fall på boxen. T ex för Wood-boxen.



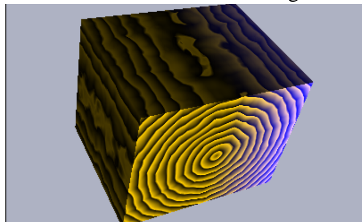
Om man är missnöjd med färgen i trätexturen, får man blanda färgerna annorlunda. Jag hade fått för mig att färgerna RGB i proportion 1:0.8:0.4 skulle ge en brun ton. Ändrade därför till



men nådde inte riktigt önskad brun nyans. Som synes finns det boxar för multiplikation, tal och allmänna uttryck. Dessa hämtas med hjälp av **Insert**-menyns undermenyer.
När vi är nöjda trycker vi på **OK** och återvänder därmed till (se till att Trä är markerad).



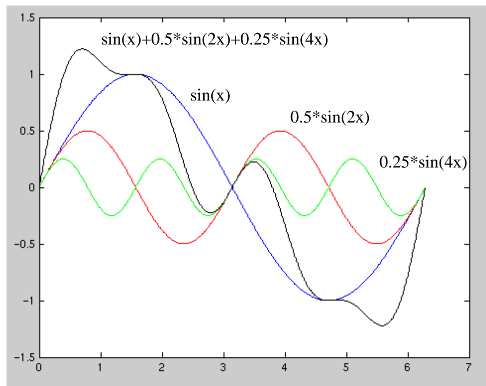
Om vi tar till **Edit Mapping** upptäcker vi att nu finns bara **Linear**-mapping (inte t ex **Spherical**), vilket beror på att en procedurtextur ju beskriver varje voxel direkt. Tryck slutligen på OK och vi är tillbaka hos AoI:s huvudfönster. **Scene/Render Scene** ger



Perlin-brus 1 (10)

Vi skall nu beskriva en teknik som introducerades av Ken Perlin 1985 och som kan användas för tillverkning av texturer och effekter som är en aning påverkade av slumpen. Den har använts mycket i filmindustrin och Perlin har tilldelats pris av denna industri. Han har senare modifierat tekniken på olika sätt (se OH efter dessa 10) och presenterade 2001 en ny metod kallad “simplex brus”, som är lättare att implementera i hårdvara. Jag (liksom Hills bok) gör litet våld på hans algoritm.

Men först en utflykt till MATLAB. Förmodligen vet du att signaler (ljud etc) från verkligheten kan delas upp i olika frekvenser. Ofta finns det en dominerande frekvens (eller frekvensband). I följande figur illustrerar vi detta.

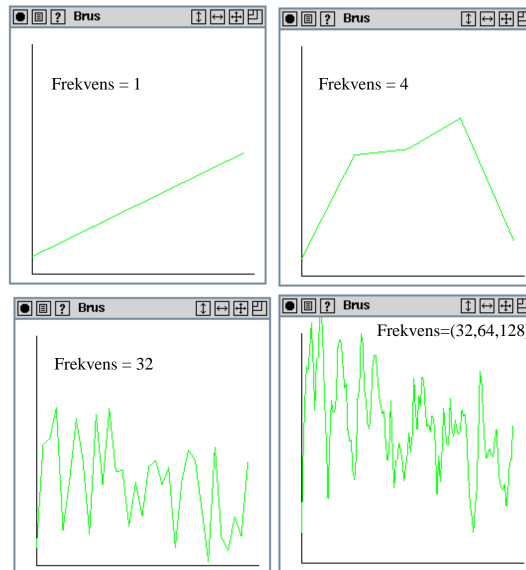



```
>> x = 0:0.01:2*pi;        % Bilda samplingsvektor
>> plot(x,sin(x),'b')      % Rita sin(x) med blått
>> hold on           % Se till att gammalt finns kvar
>> plot(x,0.5*sin(2*x),'r')   % Rita 0.5*sin(2x)
>> plot(x,0.25*sin(4*x),'g')  % Rita 0.25*sin(4x)
>> plot(x,sin(x)+0.5*sin(2*x)+0.25*sin(4*x),'k')
                              % Rita summan
```


Perlin-brus 2 (10)

Vi använder nu analogitänkande för att konstruera olika slag av brus. Vi börjar med endimensionellt kontinuerligt brus för intervallet [0,1]. Först måste vi hitta en motsvarighet till det vanliga frekvensbegreppet.

Låt oss dela in [0,1] i N lika stora delintervall med hjälp av punkterna
$x_i = i/N$, i=0,1,...,N. I var och en av dessa punkter skapar vi ett slumptal y_i (likformig fördelning på [0,1)). Vi förbinder sedan punkterna (x_i,y_i) i tur och ordning med räta linjer. Detta innebär att vi linjärinterpolerar de genererade slumpvärdena för att få en funktion definierad för alla x i intervallet. Resultat för N=1, N=4 och N=32 ser du i följande figur. Låt oss helt enkelt kalla antalet delintervall, dvs N, för **frekvensen**. I den fjärde figuren har

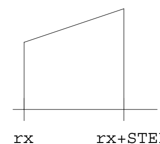


Perlin-brus 3 (10)

vi blandat brus med frekvenserna 32, 64 och 128 på samma sätt som när vi blandade sinuskurvorna, dvs amplituderna är 1, 0.5 respektive 0.25. Vitsen med detta är att vi får detaljvariation utan de kraftiga variationer vi skulle fått med frekvensen 128.

Utdrag ur kod (ligger i omritningsproceduren *display*; vi beräknar brusvärdet i brytpunkterna och ritar):

```
glOrtho (-0.1, 1.1, -0.05, 1.5, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
axlar();   // Koordinataxlarna
STEP = 1.0/frekvens;
glBegin(GL_LINE_STRIP);
   for (i=0; i<=frekvens; i++) {
       rx = i*STEP; b = brus(frekvens,rx);
       glVertex2f(rx,b);
   }
glEnd();
```



I koden anropas en funktion *brus* som givet en frekvens och ett x-värde räknar ut brusvärdet. Den returnerar alltid samma värde för givna parametrar. Hur det går till återkommer vi till.

När man blandar flera frekvenser brukar man tala om **turbulens**. Speciellt vanlig är en blandning av typen (vi skriver den för 3D-fallet)

$$turbulens(frekvens,x,y,z) = \sum_{i}^{M} \frac{1}{2^i} brus(2^i frekvens,x,y,z)$$

Tanken är att s k 1/f-brus skall approximeras, dvs brus där amplituden avtar med 1/frekvensen.

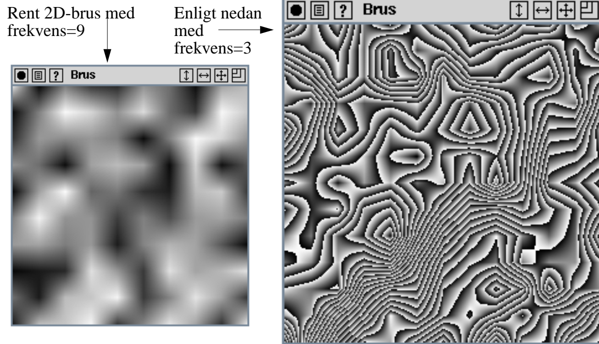
Perlin-brus 4 (10)

Vi kan även arbeta med 2D-brus, dvs brus definierat över en kvadrat

$0 \le x,y \le 1$ (för enkelhets skull). **Ett exempel:**

Rent 2D-brus med frekvens=9

Enligt nedan med frekvens=3

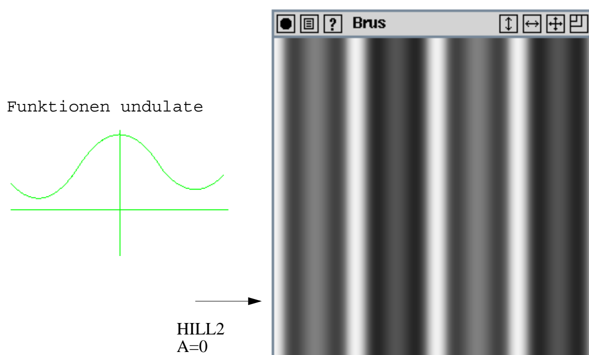


Vi ritar nu punkt för punkt. **Utdrag ur kod** (uppritningsproceduren)

```
glOrtho (0.0, width, 0.0, width, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); STEP = 1.0/width;
for (i=0; i<width; i++) {
    for (j=0; j<width; j++) {
        rx = i*STEP; ry = j*STEP;
        b = brus2(frekvens,rx,ry)+
            0.5*brus2(2*frekvens,rx,ry)+
            0.25*brus2(4*frekvens,rx,ry);
        b = 20*b; b = b - (int)b;
        glColor3f(b, b, b);
        glBegin(GL_POINTS);
            glVertex2f(i,j);
        glEnd();
    }
}
```


Perlin-brus 5 (10)

Ytterligare ett 2D-exempel, där vi försöker syntetisera marmor. Man startar med att identifiera något grunddrag i den önskade texturen. I marmorfallet kanske som här ett antal vertikala ådringar.



HILL2
A=0

Vi skulle kunna få ett mönster av denna typ genom att i förra programmet ha

```
b = sin(12.28*rx);
glColor3f(b, b, b);
```

Dvs än så länge ingen slump. Bilden och de två följande är dock gjorda med ytterligare en rad mellan de två angivna

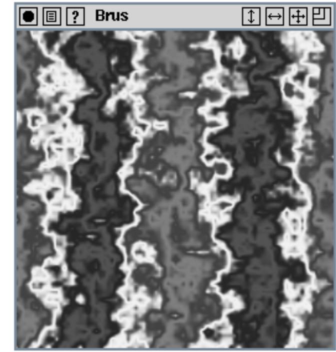
```
b = undulate(b);
```

Se längre fram. Resultatet blir något sämre utan den.

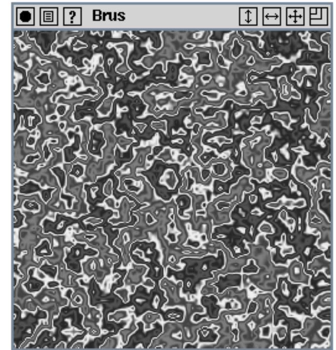
Sedan stör vi det först bildade b-värdet slumpmässigt med variation både i x och y, vilket resulterar i bilderna på nästa sida..

```
b = brus2(frekvens,rx,ry)+
    0.5*brus2(2*frekvens,rx,ry)+
    0.25*brus2(4*frekvens,rx,ry);
b = undulate(sin(12.28*rx+A*b));
glColor3f(b, b, b);
```


Perlin-brus 6 (10)



HILL2
frekvens=20
A=2



HILL2
frekvens=20
A=8

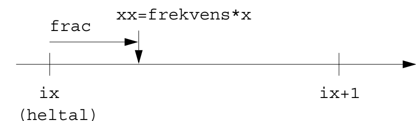
Funktionen *undulate* (andragrads-spline på [-1,1]) är knyckt från Hills bok:
float undulate(float x) { if (x<-0.4) return 0.15+2.857*(x+0.75)*(x+0.75);
else if (x<0.4) return 0.95-2.8125*x*x;
else return 0.26+2.666*(x-0.7)*(x-0.7);}

Perlin-brus 7 (10)

Hur konstrueras brus-funktionen `brus(frekvens, x)` respektive den tvådimensionella `brus2(frekvens, x, y)`? Vi vill kunna beräkna värdet för godtyckliga reella tal $0 \le x,y \le 1$ och (heltals)frekvens<256. Det är viktigt att ett visst par (x,y) - för given frekvens - alltid ger samma resultat (annars skulle texturen förändras), men samtidigt skall värdena sinsemellan te sig slumpmässiga. I 2D-fallet skulle man kunna tänka sig att en gång för alla räkna ut värdena i ett lagom tätt gitter och interpolera för övrigt. Med en frekvens om 255 skulle det betyda cirka 256x256 värden, vilket vore acceptabelt. Men eftersom vi vill överföra idén till 3D, där motsvarande behov i så fall skulle vara 256x256x256 värden, vilket vi bedömer som orimligt stort, skall vi göra på ett annat - approximativt - sätt. En stund framöver antar vi ändå att vi har en brusvektor med komponenter b_i, i=0,1,...,255 resp en brusmatris med komponenter b_{ij}, i,j=0,1,2,...,255.

Då kan vi i **1D-fallet** skriva

```
float brus(int frekvens, float x) {
    float xx = frekvens*x;  // dvs max frekvens
    int ix = (int)xx;   float frac = xx - ix;
    return b[ix] + frac*(b[ix+1]-b[ix]);
}
```



Om vi inför en funktion för linjärinterpolation

```
float LIP(float t, float a, float b) {
     return a + t*(b-a);
}
```

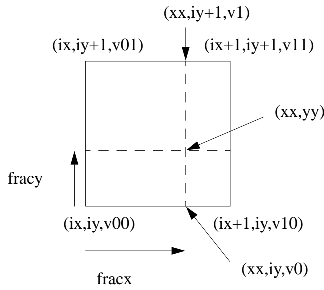
kan return-raden ersättas av

```
return LIP(frac, b[ix], b[ix+1]);
```

(i engelsk litteratur LIP -> lerp = Linear intERPolation)

Perlin-brus 8 (10)

I **2D-fallet** bildar vi på motsvarande sätt xx=frekvens*x och yy=frekvens*y och interpolerar sedan enligt figuren



vilket ger koden

```
float brus2(int frekvens, float x, float y) {
   float xx = frekvens*x, yy = frekvens*y;
   int ix = (int)xx, iy = (int)yy;
   float fracx = xx - ix, fracy = yy - iy;
   float v00, v10, v11, v01;
   float v1, v0;
   v00 = b[ix][iy]; v10 = b[ix+1][iy];
   v11 = b[ix+1][iy+1]; v01 = b[ix][iy+1];
   v1 = LIP(fracx,v01,v11);
   v0 = LIP(fracx,v00,v10);
   return LIP(fracy,v0,v1);
}
```


Perlin-brus 9 (10)

3D-fallet, som var vårt egentliga mål, hanteras på liknande sätt. Nu måste vi förutsätta att vi har slumpvärden b_{ijk}, i,j,k=0,1,...,255. Och antalet interpolationer växer.

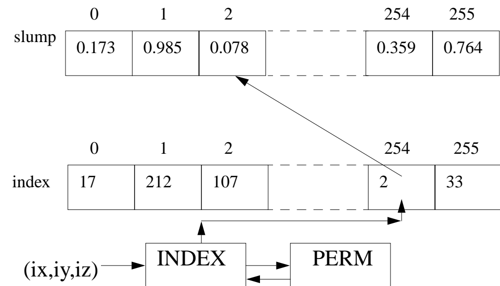
```
float brus3(int frekvens, float x, float y, float z) {
   float xx = frekvens*x, yy = frekvens*y;
         zz = frekvens*z;
   int ix = (int)xx, iy = (int)yy, iz = (int)zz;
   float fracx = xx - ix, fracy = yy - iy,
         fracz = zz -iz;
   float v000, v100, v110, v010, v001, v101,
         v111, v011;
   float v00, v10, v11, v01;
   float y1, y0;
   // Hörnvärdena; snurror bättre men trasslar till
   // figuren
   v000 = b[ix][iy][iz]; v100 = b[ix+1][iy][iz];
   v110 = b[ix+1}[iy+1][iz];
   v010 = b[ix][iy+1][iz)];
   v001 = b[ix][iy][iz+];
   v101 = b[ix+1][iy][iz+1)];
   v111 = b[ix+1][iy+1][iz+1];
   v011 = b[ix][iy+1][iz+1];
   // Interpolera till planet iz+fracz
   v00 = LIP(fracz, v000, v001);
   v10 = LIP(fracz,v100, v101);
   v11 = LIP(fracz,v110, v111);
   v01 = LIP(fracz,v010, v011);
   // Interpolera till linje
   y1 = LIP(fracx,v01,v11);
   y0 = LIP(fracx,v00,v10);
   // Interpolera till punkt
   return LIP(fracy,y0,y1);
}
```


Perlin-brus 10 (10)

Till sist hur vi undviker matrisen [b_{ij}] och aggregatet [b_{ijk}].

Vi bildar en enda slumpvektor (större längd möjlig) `float slump[256]` med 256 tal mellan 0 och 1. För det kan vi använda `rand(...)/(RAND_MAX-1.0)`.
C-funktionen `rand()` producerar i vår miljö heltalsslumptal i intervallet [0,32767] och RAND_MAX=32767, dvs vi får reella tal i intervallet [0,1).

Vi bildar dessutom en heltalsvektor `int index[256]` med heltalen 0-255 väl blandade.



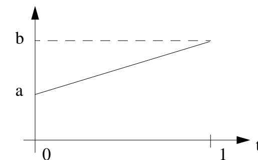
Under processen behöver vi beräkna slumptal för trippler (ix,iy,iz) med heltal. I princip skulle vi kunna använda slumpvärdet slump[(ix+iy+iz) % 256]. Men för att reducera risken för oönskade regelbundenheter görs detta på ett något omständligare sätt via funktioner INDEX och PERM. Dessa kan se ut så här (för den C-bildade: dessa kan med fördel vara makron):

```
int PERM(int x) {
 return index[x & 255];//Snabbare än mod-beräkning med %
}
int INDEX(int i,int j, int k) {
   return PERM(i+PERM(j+PERM(k)));
}
```

Vi skriver sedan slump[INDEX(ix,iy,iz)] i st f b[ix][iy][iz] i koden ovan.

Mera Perlin-brus

Bruset som vi hittills beskrivet det blir hackigt (ej kontinuerlig derivata). Det beror naturligtvis på att vi interpolerar linjärt



vilket kan skrivas
$F(t) = a + t(b - a) = (1 - t)a + tb = (1 - g(t))a + g(t)b$ med $g(t) = t$.
Enda sättet att åstadkomma kontinuerlig lutning är att se till att $F'(0) = F'(1) = 0$. För det måste vi välja g(t) annorlunda. Eftersom $F'(t) = (b-a)g'(t)$, måste $g'(0) = g'(1) = 0$ och självklart $g(0) = 0$ och $g(1) = 1$. Dvs g(t) Hermite-interpolerar. Man finner lätt att
$g(t) = 3t^2 - 2t^3$ uppfyller kravet. Perlin använde motsvarande F(t) i sin ursprungliga metod. Senare ville han även kontinuerlig andraderivata, vilket uppfylles med femtegradspolynomet
$g(t) = 6t^5 - 15t^4 -10t^3$.

I vänstra figuren är 1-g(t) uppritad för de tre fallen. Till höger en slumpkurva också för de tre fallen.