Procedural Bump Mapping and Noise

Code and images from Ebert, David S., editor, *Texturing and Modeling: a Procedural Approach*. 1994

Bump Mapping Details

F(u,v) = bump height function

P(u,v) = surface position

U = ∂f/∂u (N x ∂P/∂v)

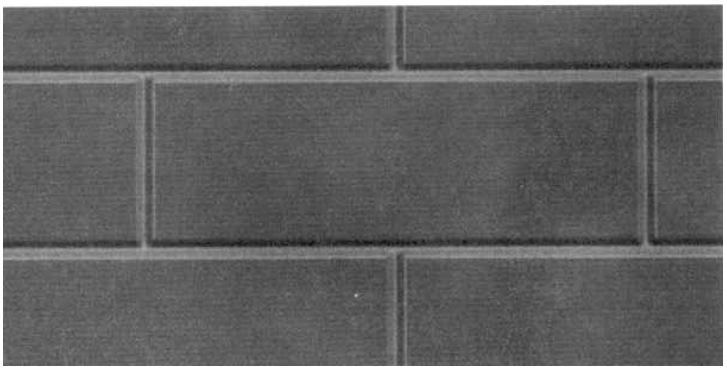
V = -∂f/∂v (N x ∂P/∂u)

D = U + V

N′ = (N + D) / | N + D |

(see Figure 17 on p. 39 of Ebert)

Bump-Mapped Brick



Example - Bumped Brick

Describe height function in terms of texture coordinates

Using built-in RenderMan functions:

- **displace point along normal according to height**
- **find partial derivatives of new surface with respect to texture coordinates**
- **cross the partials to get vector normal to new surface**

Without Special Assistance

Compute ∂P/∂u and ∂P/∂v analytically according to surface geometry (e.g. sphere)

OR

- **Evaluate P at 4 nearby points by varying u and v slightly, then approximate partial using differences**

Compute ∂f/∂u and ∂f/∂v analytically according to height function

Apply preceding formulas

Bevelling Effects

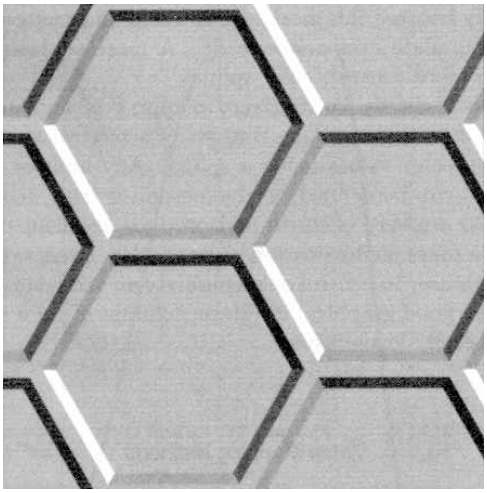
Nice ridges along edges of geometric figures

Parameters:

- **Total ridge and plateau widths**
- **slope at top and bottom of ridge**

Use perpendicular direction to closest edge as D (to add to normal), and scale according to ridge function

Bevelling



Noise Functions

Break up regularity

Enable modelling of irregular phenomena

White Noise

Sequence of random numbers

Uniformly distributed

Totally uncorrelated

- **no correlation between successive values**

Not desirable for texture generation

- **Too sensitive to sampling problems**
- **Arbitrarily high frequency content**

Ideal Noise for Texture Generation

Repeatable pseudorandom function of inputs

Known range [-1, 1]

Band-limited (maximum freq. about 1)

No obvious periodicities

Stationary and isotropic

- **statistical properties invarient under translation and rotation**

Lattice Noise

Low pass filtered version of white noise

- **Random values associated with integer positions in noise space**
- **Intermediate values generated by some form of interpolation**
- **Frequency content limited by spacing of lattice**

Generating a Lattice

Generate a fixed-size table of random numbers

Hashing function indexes into the table to get value at any lattice point

Example Lattice Indexing

```
#define TABSIZE     256
#define TABMASK     ( TABSIZE - 1 )
#define PERM(x)     perm[ (x) & TABMASK ]
#define INDEX(ix,iy,iz)                    \
        PERM((ix)+PERM((iy)+PERM(iz)))
```

perm contains random permutation of integers in

[0, TABSIZE - 1]

Value Noise

Create additional table of random values (in range [-1,1])

Index table according to permuation-based INDEX function just presented

(see sample code handout)

Interpolation Schemes

Linear interpolation -

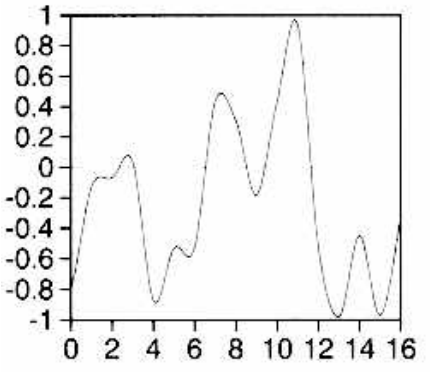
- **not really smooth enough**

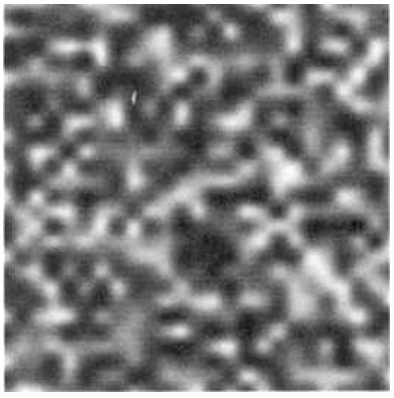
Quadratic or cubic spline interpolation

- **may still have some artifacts resulting from grid layout**

Convolution with radially symmetric filter kernel

1D and 2D Value Noise





Gradient Noise

Store direction vector at each lattice point

Noise values at lattice point is zero

Computing intermediate values:

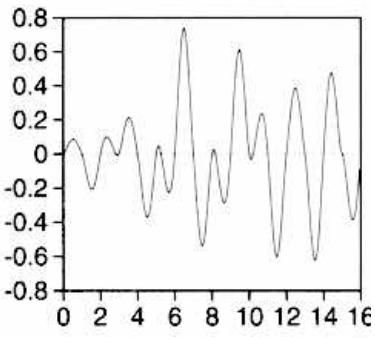
For each neighboring lattice point

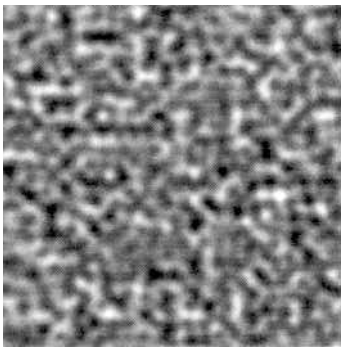
compute displacement along direction

Linearly interpolate between resulting 8 values to get final value

(see sample code handout)

1D and 2D Gradient Noise





Value vs. Gradient Noise

Both noises have limited frequencies

Value noise slightly simpler to compute

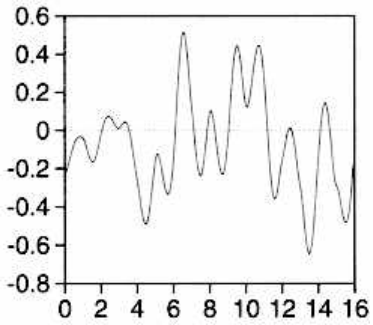
Gradient noise has most of the energy in the higher frequencies

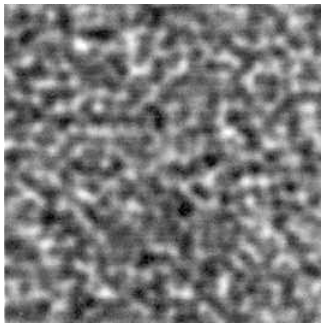
- **forced zero crossings**

Gradient noise has regularity because of zero crossings

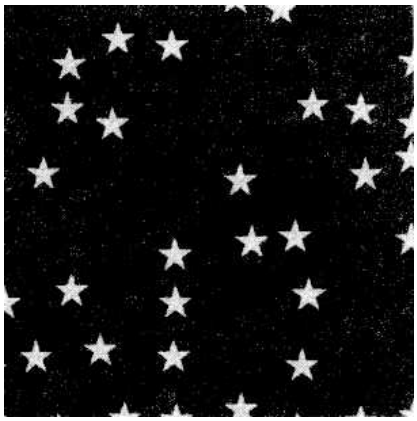
Value Gradient Noise

Weighted sum of value and gradient noises





Example - Star Wallpaper



Example - Star Wallpaper

Divide 2D texture space into uniform grid

Decide whether or not to place a star in each cell

Perturb position of star within each cell

To render a point on surface, check nearby cells for stars which may cover point

(see code handout)

Example - Perturbed Texture

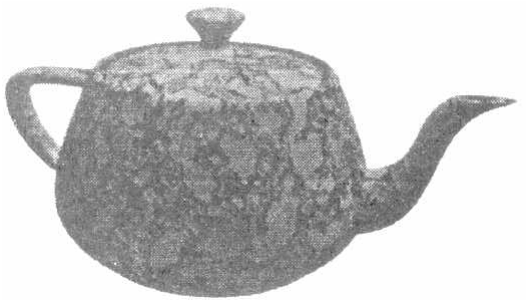
Example - Perturbed Texture

Use noise function to apply perturbation to texture coordinates

Look up image texture (or generate procedural texture) using modified coordinates

(see code handout)

Example - Blue Marble



Marble vase (right) from Foley, van Dam, Feiner, and Hughes. *Computer Graphics: Principles and Practice*.

Example - Blue Marble

Use 3D position to compute 3D texture coordinates

Accumulate noise functions at several frequencies

- **one type of spectral synthesis**

Use sum of noise to determine marble color

- **using spline interpolation between colors**

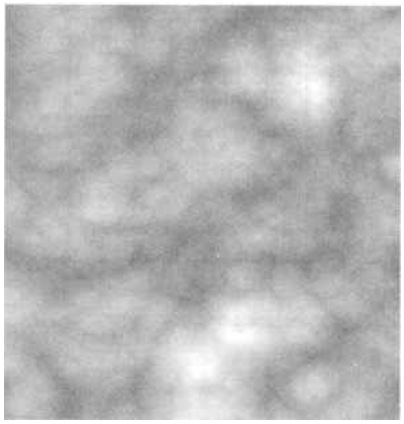
Modelling Gases

Represent 3D gas as density volume

Use turbulence function as basic gas description

Adjust turbulence by raising it to a power, taking the sine, etc.

Turbulence



Turbulence

```
float turbulence(point Q)
{
   float value = 0;
   for (f= MINFREQ; f < MAXFREQ; f *= 2)
        value += abs(noise(Q*f))/f;
   return value;
}
```

(in practice, don't use a round number like 2)

Basic gas

```
float gas(point P, float max_density,
          float exponent)
{
   float turb, density;
   turb = turbulence(pt);
   /* or turb = (1 + sin(turbulence(pt)*PI))/2 */
   density =
        pow(turb*max_density, exponent);
   return density;
}
```

Placing and Shaping Gas

Place some primitive shape to contain density volume

Attenuate density to account for dissipation

Steaming teacup example

- **attenuate according to distance from center of tea surface**
- **attenuate according to height above tea surface**

Steaming Tea Cup

More Turbulence Uses

Add variation to color of surface textures

Use as bump mapping function to add variety to normals

```
#include "proctext.h"

#define BRICKWIDTH      0.25
#define BRICKHEIGHT     0.08
#define MORTARTHICKNESS 0.01

#define BMWIDTH         (BRICKWIDTH+MORTARTHICKNESS)
#define BMHEIGHT        (BRICKHEIGHT+MORTARTHICKNESS)
#define MWF             (MORTARTHICKNESS*0.5/BMWIDTH)
#define MHF             (MORTARTHICKNESS*0.5/BMHEIGHT)

surface
brickbump(
   uniform float Ka = 1;
   uniform float Kd = 1;
   uniform color Cbrick = color (0.5, 0.15, 0.14);
   uniform color Cmortar = color (0.5, 0.5, 0.5);
    )
{  color Ct;
   point Nf;
   float ss, tt, sbrick, tbrick, w, h;
   float scoord = s;
   float tcoord = t;
   float sbump, tbump, stbump;

   Nf = normalize(faceforward(N, I));

   ss = scoord / BMWIDTH;
   tt = tcoord / BMHEIGHT;

   if (mod(tt*0.5,1) > 0.5)
      ss += 0.5;  /* shift alternate rows */
   sbrick = floor(ss); /* which brick? */
   tbrick = floor(tt); /* which brick? */
   ss -= sbrick;
   tt -= tbrick;
   w = step(MWF,ss) - step(1-MWF,ss);
   h = step(MHF,tt) - step(1-MHF,tt);

   Ct = mix(Cmortar, Cbrick, w*h);

   /* compute bump-mapping function for mortar grooves */
   sbump = smoothstep(0,MWF,ss) - smoothstep(1-MWF,1,ss);
   tbump = smoothstep(0,MHF,tt) - smoothstep(1-MHF,1,tt);
   stbump = sbump * tbump;

   /* compute shading normal - move surface and cross the partial derivatives to get new normal */
   Nf = calculatenormal(P + normalize(N) * stbump);
   Nf = normalize(faceforward(Nf, I));

   /* diffuse reflection model */
   Oi = Os;
   Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

Rendering Techniques Handout — Noise Functions

```
/*
 * Declarations and preprocessor definitions used in the various noise
 * functions.
 * Darwyn Peachey, June, 1994.
 */

#ifndef _NOISE_H_
#define _NOISE_H_ 1

#define TABSIZE         256
#define TABMASK         (TABSIZE-1)
#define PERM(x)         perm[(x)&TABMASK]
#define INDEX(ix,iy,iz)  PERM((ix)+PERM((iy)+PERM(iz)))

#define RANDMASK  0x7fffffff
#define RANDNBR   ((random() & RANDMASK)/(double) RANDMASK)

extern unsigned char perm[TABSIZE];         /* see perm.c */

extern float catrom2(float d);              /* see catrom2.c */

#endif /* _NOISE_H_ */
```

Value Noise

```
#include "proctext.h"
#include "noise.h"

static float valueTab[TABSIZE];

static void valueTabInit(int seed);
static float vlattice(int ix, int iy, int iz);

float
vnoise(float x, float y, float z)
{
   int ix, iy, iz;
   int i, j, k;
   float fx, fy, fz;
   float xknots[4], yknots[4], zknots[4];
   static int initialized = 0;
```

```
    if (!initialized) {
        valueTabInit(665);
        initialized = 1;
    }

    ix = FLOOR(x);
    fx = x - ix;

    iy = FLOOR(y);
    fy = y - iy;

    iz = FLOOR(z);
    fz = z - iz;

    for (k = -1; k <= 2; k++) {
        for (j = -1; j <= 2; j++) {
            for (i = -1; i <= 2; i++)
                xknots[i+1] = vlattice(ix+i,iy+j,iz+k);
            yknots[j+1] = spline(fx, 4, xknots);
        }
        zknots[k+1] = spline(fy, 4, yknots);
    }
    return spline(fz, 4, zknots);
}

static void
valueTabInit(int seed)
{
    float *table = valueTab;
    int i;

    srandom(seed);
    for(i = 0; i < TABSIZE; i++)
        *table++ = 1. - 2.*RANDNBR;
}

static float
vlattice(int ix, int iy, int iz)
{
    return valueTab[INDEX(ix,iy,iz)];
}
```

Gradient Noise

```
#include "proctext.h"
#include "noise.h"

#define SMOOTHSTEP(x)  ((x)*(x)*(3 - 2*(x)))

static float gradientTab[TABSIZE*3];

static void gradientTabInit(int seed);
static float glattice(int ix, int iy, int iz, float fx, float fy, float fz);

float
gnoise(float x, float y, float z)
{
   int ix, iy, iz;
   float fx0, fx1, fy0, fy1, fz0, fz1;
   float wx, wy, wz;
   float vx0, vx1, vy0, vy1, vz0, vz1;
   static int initialized = 0;

   if (!initialized) {
      gradientTabInit(665);
      initialized = 1;
   }

   ix = FLOOR(x);
   fx0 = x - ix;
   fx1 = fx0 - 1;
   wx = SMOOTHSTEP(fx0);

   iy = FLOOR(y);
   fy0 = y - iy;
   fy1 = fy0 - 1;
   wy = SMOOTHSTEP(fy0);

   iz = FLOOR(z);
   fz0 = z - iz;
   fz1 = fz0 - 1;
   wz = SMOOTHSTEP(fz0);

   vx0 = glattice(ix,iy,iz,fx0,fy0,fz0);
   vx1 = glattice(ix+1,iy,iz,fx1,fy0,fz0);
   vy0 = LERP(wx, vx0, vx1);
   vx0 = glattice(ix,iy+1,iz,fx0,fy1,fz0);
```

```
   vx1 = glattice(ix+1,iy+1,iz,fx1,fy1,fz0);
   vy1 = LERP(wx, vx0, vx1);
   vz0 = LERP(wy, vy0, vy1);

   vx0 = glattice(ix,iy,iz+1,fx0,fy0,fz1);
   vx1 = glattice(ix+1,iy,iz+1,fx1,fy0,fz1);
   vy0 = LERP(wx, vx0, vx1);
   vx0 = glattice(ix,iy+1,iz+1,fx0,fy1,fz1);
   vx1 = glattice(ix+1,iy+1,iz+1,fx1,fy1,fz1);
   vy1 = LERP(wx, vx0, vx1);
   vz1 = LERP(wy, vy0, vy1);

   return LERP(wz, vz0, vz1);
}

static void
gradientTabInit(int seed)
{
   float *table = gradientTab;
   float z, r, theta;
   int i;

   srandom(seed);
   for(i = 0; i < TABSIZE; i++) {
      z = 1. - 2.*RANDNBR;
      /* r is radius of x,y circle */
      r = sqrtf(1 - z*z);
      /* theta is angle in (x,y) */
      theta = 2 * M_PI * RANDNBR;
      *table++ = r * cosf(theta);
      *table++ = r * sinf(theta);
      *table++ = z;
   }
}

static float
glattice(int ix, int iy, int iz,
   float fx, float fy, float fz)
{
   float *g = &gradientTab[INDEX(ix,iy,iz)*3];
   return g[0]*fx + g[1]*fy + g[2]*fz;
}
```

Rendering Techniques Handout — Noise-based Shaders

Star Wallpaper

```
#define NCELLS 10
#define CELLSIZE (1/NCELLS)
#define snoise(s,t)       (2*noise((s),(t))-1)

surface
wallpaper(
   uniform float Ka = 1;
   uniform float Kd = 1;
   uniform color starcolor = color (1.0000,0.5161,0.0000);
   uniform float npoints = 5;
    )
{
   color Ct;
   point Nf;
   float ss, tt, angle, r, a, in_out;
   float sctr, tctr, scell, tcell;
   float scellctr, tcellctr;
   float i, j;
   uniform float rmin = 0.01, rmax = 0.03;
   uniform float starangle = 2*PI/npoints;
   uniform point p0 = rmax*(cos(0),sin(0),0);
   uniform point p1 = rmin*
      (cos(starangle/2),sin(starangle/2),0);
   uniform point d0 = p1 - p0;
   point d1;

   scellctr = floor(s*NCELLS);
   tcellctr = floor(t*NCELLS);
   in_out = 0;

   for (i = -1; i <= 1; i += 1) {
      for (j = -1; j <= 1; j += 1) {
            scell = scellctr + i;
            tcell = tcellctr + j;
            if (float noise(3*scell-9.5,7*tcell+7.5) < 0.55) {
            sctr = CELLSIZE * (scell + 0.5
                + 0.6 * snoise(scell+0.5, tcell+0.5));
            tctr = CELLSIZE * (tcell + 0.5
                + 0.6 * snoise(scell+3.5, tcell+8.5));
            ss = s - sctr;
            tt = t - tctr;
```

```
            angle = atan(ss, tt) + PI;
            r = sqrt(ss*ss + tt*tt);
            a = mod(angle, starangle)/starangle;

            if (a >= 0.5)
               a = 1 - a;
            d1 = r*(cos(a), sin(a),0) - p0;
            in_out += step(0, zcomp(d0^d1));
         }
      }
   }
   Ct = mix(Cs, starcolor, step(0.5,in_out));

   /* "matte" reflection model */
   Nf = normalize(faceforward(N, I));
   Oi = Os;
   Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

Perturbed Texture

```
#include "proctext.h"

surface
perturb ()
{
   point Psh;
   float ss, tt;

   Psh = transform("shader", P) * 0.2;
   ss = s + 0.1 * snoise(Psh);
   tt = t + 0.05 * snoise(Psh+(1.5,6.7,3.4));
   Ci = texture("example.tx", ss, tt);
}
```

Blue Marble

```
#include "proctext.h"

#define PALE_BLUE        color (0.25, 0.25, 0.35)
#define MEDIUM_BLUE      color (0.10, 0.10, 0.30)
#define DARK_BLUE        color (0.05, 0.05, 0.26)
#define DARKER_BLUE      color (0.03, 0.03, 0.20)
#define NNOISE           4

color
marble_color(float m)
{
   return color spline(
      clamp(2*m + .75, 0, 1),
      PALE_BLUE, PALE_BLUE,
      MEDIUM_BLUE, MEDIUM_BLUE, MEDIUM_BLUE,
      PALE_BLUE, PALE_BLUE,
      DARK_BLUE, DARK_BLUE,
      DARKER_BLUE, DARKER_BLUE,
      PALE_BLUE, DARKER_BLUE);
}

surface
blue_marble(
   uniform float Ka = 1;
   uniform float Kd = 0.8;
   uniform float Ks = 0.2;
   uniform float texturescale = 2.5;
   uniform float roughness = 0.1;
    )
{
   color Ct;
   point NN;
   point PP;
   float i, f, marble;

   NN = normalize(faceforward(N, I));
   PP = transform("shader", P) * texturescale;

   marble = 0; f = 1;
   for (i = 0; i < NNOISE; i += 1) {
      marble += snoise(PP * f)/f;
      f *= 2.17;
   }
```

```
    Ct = marble_color(marble);

    Ci = Os * (Ct * (Ka * ambient() + Kd * diffuse(NN))
        + Ks * specular(NN, normalize(-I), roughness));
}
```