```
/
**************************************************************************
**
 * noises.h - various noise-based patterns
 *
 * Author: Larry Gritz (gritzl@acm.org)
 *
 * Reference:
 *   _Advanced RenderMan: Creating CGI for Motion Picture_,
 *   by Anthony A. Apodaca and Larry Gritz, Morgan Kaufmann, 1999.
 *
 * $Revision: 1.1 $    $Date: 2000/08/28 01:30:35 $
 *

**************************************************************************
**/


#ifndef NOISES_H
#define NOISES_H 1


#ifndef FILTERWIDTH_H
#include "filterwidth.h"    /* Needed for filterwidth and friends */
#endif

#ifndef PATTERNS_H
#include "patterns.h"    /* Needed for filteredabs */
#endif




#ifndef snoise
/*
 * Signed noise -- the original Perlin kind with range (-1,1) We
prefer
 * signed noise to regular noise mostly because its average is zero.
 * We define three simple macros:
 *   snoise(p) - Perlin noise on either a 1-D (float) or 3-D (point)
domain.
 *   snoisexy(x,y) - Perlin noise on a 2-D domain.
 *   vsnoise(p) - vector-valued Perlin noise on either 1-D or 3-D
domain.
 */
#define snoise(p) (2 * (float noise(p)) - 1)
#define snoisexy(x,y) (2 * (float noise(x,y)) - 1)
#define vsnoise(p) (2 * (vector noise(p)) - 1)
#endif
```

```
/* If we know the filter size, we can crudely antialias snoise by
fading
 * to its average value at approximately the Nyquist limit.
 */
#define filteredsnoise(p,width) (snoise(p) * (1-smoothstep
(0.2,0.75,width)))
#define filteredvsnoise(p,width) (vsnoise(p) * (1-smoothstep
(0.2,0.75,width)))



/* fractional Brownian motion
 * Inputs:
 *    p, filtwidth   position and approximate inter-pixel spacing
 *    octaves        max # of octaves to calculate
 *    lacunarity     frequency spacing between successive octaves
 *    gain           scaling factor between successive octaves
 */
float fBm (point p; float filtwidth;
           uniform float octaves, lacunarity, gain)
{
    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0, fw = filtwidth;
    uniform float i;

    for (i = 0;  i < octaves;  i += 1) {
#pragma nolint
        sum += amp * snoise (pp, fw);
        amp *= gain;  pp *= lacunarity;  fw *= lacunarity;
    }
    return sum;
}


/* Typical use of fBm: */
#define fBm_default(p)  fBm (p, filterwidthp(p), 4, 2, 0.5)




/* A vector-valued antialiased fBm. */
vector
vfBm (point p; float filtwidth;
      uniform float octaves, lacunarity, gain)
{
    uniform float amp = 1;
    varying point pp = p;
```

```
    varying vector sum = 0;
    varying float fw = filtwidth;
    uniform float i;

    for (i = 0;  i < octaves;  i += 1) {
#pragma nolint
        sum += amp * filteredvsnoise (pp, fw);
        amp *= gain;  pp *= lacunarity;  fw *= lacunarity;
    }
    return sum;
}


/* Typical use of vfBm: */
#define vfBm_default(p)  vfBm (p, filterwidthp(p), 4, 2, 0.5)



/* The stuff that Ken Musgrave calls "VLNoise" */
#define VLNoise(Pt,scale) (snoise(vsnoise(Pt)*scale+Pt))
#define filteredVLNoise(Pt,fwidth,scale) \

(filteredsnoise(filteredvsnoise(Pt,fwidth)*scale+Pt,fwidth))


float
VLfBm (point p; float filtwidth;
       uniform float octaves, lacunarity, gain, scale)
{
    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0;
    varying float fw = filtwidth;
    uniform float i;

    for (i = 0;  i < octaves;  i += 1) {
#pragma nolint
        sum += amp * filteredVLNoise (pp, fw, scale);
        amp *= gain;  pp *= lacunarity;  fw *= lacunarity;
    }
    return sum;
}


/* Typical use of vfBm: */
#define VLfBm_default(p)      VLfBm (p, filterwidthp(p), 4, 2, 0.5,
1.0)
```

```
/* Antialiased turbulence.  Watch out -- the abs() call introduces
infinite
 * frequency content, which makes our antialiasing efforts much
trickier!
 */
float turbulence (point p; float filtwidth;
                  uniform float octaves, lacunarity, gain)
{
    extern float du, dv;   /* Needed for filterwidth macro */
    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0, fw = filtwidth;
    uniform float i;

    for (i = 0;  i < octaves;  i += 1) {
#pragma nolint
        float n = filteredsnoise (pp, fw);
        sum += amp * filteredabs (n, fw);
        amp *= gain;  pp *= lacunarity;  fw *= lacunarity;
    }
    return sum;
}


/* Typical use of turbulence: */
#define turbulence_default(p)  turbulence (p, filterwidthp(p), 4, 2,
0.5)




/
**************************************************************************
*****
 * Voronoi cell noise (a.k.a. Worley noise) functions
 *
 * These functions assume that space is filled with "features" (points
 * of interest).  There are interestingpatterns we can make by
 * figuring out which feature we are closest to, or to what extent
 * we're on the boundary between two features.  Several varieties of
 * these computations are below, categorized by the dimension of their
 * domains, and the number of close features they are interested in.
 *
 * All these functions have similar inputs:
 *   P      - position to test (for 3-D varieties; 2-D varieties use
ss,tt)
 *   jitter - how much to jitter the cell center positions (1 is
typical,
 *            smaller values make a more regular pattern, larger
```

```
values
 *             make a more jagged pattern; use jitter >1 at your
risk!).
 * And outputs:
 *   f_n    - distance to the nth nearest feature (f1 is closest, f2
is
 *            the distance to the 2nd closest, etc.)
 *   pos_n  - the position of the nth nearest feature.  For 2-D
varieties,
 *            these are instead spos_n and tpos_n.

*************************************************************************
*****/

/* Voronoi cell noise (a.k.a. Worley noise) -- 3-D, 1-feature version.
*/
void
voronoi_f1_3d (point P;
               float jitter;
               output float f1;
               output point pos1;
    )
{
    point thiscell = point (floor(xcomp(P))+0.5, floor(ycomp(P))+0.5,
                            floor(zcomp(P))+0.5);
    f1 = 1000;
    uniform float i, j, k;
    for (i = -1;  i <= 1;  i += 1) {
        for (j = -1;  j <= 1;  j += 1) {
            for (k = -1;  k <= 1;  k += 1) {
                 point testcell = thiscell + vector(i,j,k);
                point pos = testcell +
                            jitter * (vector cellnoise (testcell) -
0.5);
                 vector offset = pos - P;
                float dist = offset . offset; /* actually dist^2 */
                if (dist < f1) {
                    f1 = dist;  pos1 = pos;
                }
            }
         }
    }
    f1 = sqrt(f1);
}


/* Voronoi cell noise (a.k.a. Worley noise) -- 3-D, 2-feature version.
*/
void
voronoi_f1f2_3d (point P;
```

```
                  float jitter;
                  output float f1;  output point pos1;
                  output float f2;  output point pos2;
    )
{
    point thiscell = point (floor(xcomp(P))+0.5, floor(ycomp(P))+0.5,
                            floor(zcomp(P))+0.5);
    f1 = f2 = 1000;
    uniform float i, j, k;
    for (i = -1;  i <= 1;  i += 1) {
        for (j = -1;  j <= 1;  j += 1) {
            for (k = -1;  k <= 1;  k += 1) {
                 point testcell = thiscell + vector(i,j,k);
                point pos = testcell +
                            jitter * (vector cellnoise (testcell) -
0.5);
                 vector offset = pos - P;
                float dist = offset . offset; /* actually dist^2 */
                if (dist < f1) {
                    f2 = f1;  pos2 = pos1;
                    f1 = dist;  pos1 = pos;
                } else if (dist < f2) {
                    f2 = dist;  pos2 = pos;
                 }
            }
         }
    }
    f1 = sqrt(f1);  f2 = sqrt(f2);
}


/* Voronoi cell noise (a.k.a. Worley noise) -- 2-D, 1-feature version.
*/
void
voronoi_f1_2d (float ss, tt;
               float jitter;
               output float f1;
               output float spos1, tpos1;
    )
{
    float sthiscell = floor(ss)+0.5, tthiscell = floor(tt)+0.5;
    f1 = 1000;
    uniform float i, j;
    for (i = -1;  i <= 1;  i += 1) {
         float stestcell = sthiscell + i;
        for (j = -1;  j <= 1;  j += 1) {
            float ttestcell = tthiscell + j;
            float spos = stestcell +
                     jitter * (float cellnoise(stestcell, ttestcell)
- 0.5);
```

```
            float tpos = ttestcell +
                 jitter * (float cellnoise(stestcell+23,
ttestcell-87) - 0.5);
            float soffset = spos - ss;
            float toffset = tpos - tt;
            float dist = soffset*soffset + toffset*toffset;
            if (dist < f1) {
                f1 = dist;
                spos1 = spos;  tpos1 = tpos;
            }
        }
    }
    f1 = sqrt(f1);
}


/* Voronoi cell noise (a.k.a. Worley noise) -- 2-D, 2-feature version.
*/
void
voronoi_f1f2_2d (float ss, tt;
                 float jitter;
                 output float f1;
                 output float spos1, tpos1;
                 output float f2;
                 output float spos2, tpos2;
    )
{
    float sthiscell = floor(ss)+0.5, tthiscell = floor(tt)+0.5;
    f1 = f2 = 1000;
    uniform float i, j;
    for (i = -1;  i <= 1;  i += 1) {
        float stestcell = sthiscell + i;
        for (j = -1;  j <= 1;  j += 1) {
            float ttestcell = tthiscell + j;
            float spos = stestcell +
                   jitter * (cellnoise(stestcell, ttestcell) - 0.5);
            float tpos = ttestcell +
                   jitter * (cellnoise(stestcell+23, ttestcell-87) -
0.5);
            float soffset = spos - ss;
            float toffset = tpos - tt;
            float dist = soffset*soffset + toffset*toffset;
            if (dist < f1) {
                f2 = f1;  spos2 = spos1;  tpos2 = tpos1;
                f1 = dist;  spos1 = spos;  tpos1 = tpos;
            } else if (dist < f2) {
                f2 = dist;
                spos2 = spos;  tpos2 = tpos;
            }
        }
```

```
    }
    f1 = sqrt(f1);  f2 = sqrt(f2);
}


#endif /* NOISES_H */
```