```
/
*****************************************************************************
**
 * patterns.h - Some handy functions for various patterns.  Wherever
 *              possible, antialiased versions will also be given.
 *
 * 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 PATTERNS_H
#define PATTERNS_H 1

#include "filterwidth.h"



/* Handy square routine */
float sqr (float x)
{
    return x*x;
}



/* Antialiased abs().
 * Compute the box filter of abs(t) from x-dx/2 to x+dx/2.
 * Hinges on the realization that the indefinite integral of abs(x) is
 * sign(x) * 1/2 x*x;
 */
float filteredabs (float x, dx)
{
    float integral (float t) {
        return sign(t) * 0.5 * t*t;
    }

    float x0 = x - 0.5*dx;
    float x1 = x0 + dx;
    return (integral(x1) - integral(x0)) / dx;
}
```

```
/* Antialiased smoothstep(e0,e1,x).
 * Compute the box filter of smoothstep(e0,e1,t) from x-dx/2 to x+dx/
2.
 * Strategy: divide domain into 3 regions: t < e0, e0 <= t <= e1,
 * and t > e1.  Region 1 has integral 0.  Region 2 is computed by
 * analytically integrating smoothstep, which is -2t^3+3t^2.  Region 3
 * is trivially 1.
 */
float filteredsmoothstep (float e0, e1, x, dx)
{
    float integral (float t) {
        return -0.5*t*t * (t*t + t);
    }

    /* Compute x0, x1 bounding region of integration, and normalize so
that
     * e0==0, e1==1
     */
    float edgediff = e1 - e0;
    float x0 = (x-e0)/edgediff;
    float fw = dx / edgediff;
    x0 -= 0.5*fw;
    float x1 = x0 + fw;

    /* Region 1 always contributes nothing */
    float int = 0;
    /* Region 2 - compute integral in region between 0 and 1 */
    if (x0 < 1 && x1 > 0)
        int += integral(min(x1,1)) - integral(max(x0,0));
    /* Region 3 - is 1.0 */
    if (x1 > 1)
        int += x1-max(1,x0);
    return int / fw;
}



/* A 1-D pulse pattern:  return 1 if edge0 <= x <= edge1, otherwise 0
*/
float pulse (float edge0, edge1, x)
{
    return step(edge0,x) - step(edge1,x);
}
```

```
float filteredpulse (float edge0, edge1, x, dx)
{
    float x0 = x - dx/2;
    float x1 = x0 + dx;
    return max (0, (min(x1,edge1)-max(x0,edge0)) / dx);
}



/* A pulse train: a signal that repeats with a given period, and is
 * 0 when 0 <= mod(x,period) < edge, and 1 when mod(x,period) > edge.
 */
float pulsetrain (float edge, period, x)
{
    return pulse (edge, period, mod(x,period));
}


/* Filtered pulse train: it's not as simple as just returning the mod
 * of filteredpulse -- you have to take into account that the filter
may
 * cover multiple pulses in the train.
 * Strategy: consider the function that is the integral of the pulse
 * train from 0 to x. Just subtract!
 */
float filteredpulsetrain (float edge, period, x, dx)
{
    /* First, normalize so period == 1 and our domain of interest is >
0 */
    float w = dx/period;
    float x0 = x/period - w/2;
    float x1 = x0+w;
    float nedge = edge / period;   /* normalized edge value */

    /* Definite integral of normalized pulsetrain from 0 to t */
    float integral (float t) {
        extern float nedge;
        return ((1-nedge)*floor(t) + max(0,t-floor(t)-nedge));
    }

    /* Now we want to integrate the normalized pulsetrain over [x0,x1]
*/
    return (integral(x1) - integral(x0)) / w;
}



float
smoothpulse (float e0, e1, e2, e3, x)
{
```

```
    return smoothstep(e0,e1,x) - smoothstep(e2,e3,x);
}


float
filteredsmoothpulse (float e0, e1, e2, e3, x, dx)
{
    return filteredsmoothstep(e0,e1,x,dx) -
filteredsmoothstep(e2,e3,x,dx);
}



/* A pulse train of smoothsteps: a signal that repeats with a given
 * period, and is 0 when 0 <= mod(x/period,1) < edge, and 1 when
 * mod(x/period,1) > edge.
 */
float smoothpulsetrain (float e0, e1, e2, e3, period, x)
{
    return smoothpulse (e0, e1, e2, e3, mod(x,period));
}



/* varyEach takes a computed color, then tweaks each indexed item
 * separately to add some variation.  Hue, saturation, and lightness
 * are all independently controlled.  Hue adds, but saturation and
 * lightness multiply.
 */
color varyEach (color Cin; float index, varyhue, varysat, varylum;)
{
    /* Convert to "hsl" space, it's more convenient */
    color Chsl = ctransform ("hsl", Cin);
    float h = comp(Chsl,0), s = comp(Chsl,1), l = comp(Chsl,2);
    /* Modify Chsl by adding Cvary scaled by our separate h,s,l
controls */
    h += varyhue * (cellnoise(index+3)-0.5);
    s *= 1 - varysat * (cellnoise(index-14)-0.5);
    l *= 1 - varylum * (cellnoise(index+37)-0.5);
    Chsl = color (mod(h,1), clamp(s,0,1), clamp(l,0,1));
    /* Clamp hsl and transform back to rgb space */
    return ctransform ("hsl", "rgb", clamp(Chsl,color 0, color 1));
}



/* Given 2-D texture coordinates ss,tt and their filter widths ds, dt,
 * and the width and height of the grooves between tiles (assuming
that
 * tile spacing is 1.0), figure out which (integer indexed) tile we
```

```
are
 * on and what coordinates (on [0,1]) within our individual tile we
are
 * shading.
 */
float
tilepattern (float ss, tt, ds, dt;
             float groovewidth, grooveheight;
             output float swhichtile, twhichtile;
             output float stile, ttile;)
{
    swhichtile = floor (ss);
    twhichtile = floor (tt);
    stile = ss - swhichtile;
    ttile = tt - twhichtile;

    return filteredpulsetrain (groovewidth, 1, ss+groovewidth/2, ds)
             * filteredpulsetrain (grooveheight, 1, tt+grooveheight/2,
dt);
}



/* basic brick tiling pattern --
 *   inputs:
 *      x, y                    positions on a 2-D surface
 *      tilewidth, tileheight   dimensions of each tile
 *      rowstagger              how much does each row stagger
relative to
 *                                   the previous row
 *      rowstaggervary          how much should rowstagger randomly
vary
 *      jaggedfreq, jaggedamp   adds noise to the edge between the
tiles
 *   outputs:
 *      row, column             index which tile the sample is in
 *      xtile, ytile            position within this tile (0-1)
 */
void basicbrick (float x, y;
                 uniform float tilewidth, tileheight;
                 uniform float rowstagger, rowstaggervary;
                 uniform float jaggedfreq, jaggedamp;
                 output float column, row;
                 output float xtile, ytile;
    )
{
    point PP;
    float scoord = x, tcoord = y;

    if (jaggedamp != 0.0) {
```

```
        /* Make the shapes of the bricks vary just a bit */
        PP = point noise (x*jaggedfreq/tilewidth, y*jaggedfreq/
tileheight);
        scoord += jaggedamp * xcomp (PP);
        tcoord += jaggedamp * ycomp (PP);
    }

    xtile = scoord / tilewidth;
    ytile = tcoord / tileheight;
    row = floor (ytile);   /* which brick row? */

    /* Shift the columns randomly by row */
    xtile += mod (rowstagger * row, 1);
    xtile += rowstaggervary * (noise (row+0.5) - 0.5);

    column = floor (xtile);
    xtile -= column;
    ytile -= row;
}



#endif /* defined(PATTERNS_H) */
```