APPENDIX A: SAMPLE SHADERS

A.1. Surface shaders

A.1.1. Simple brick

```
// ************************************************
//
//     Brick shader.
//
//    Parameters:
//        px_rc_co[3] - This is the output color. The value ranges
//           from 0.0 to 0.999
//        px_shader_texcoord[2] - The standard OpenGL texture
//           coordinates. We expect values to vary from 0 to 1.
//           brick_width, brick_height, mortar - The size of the
//           brick pattern. These are uniform, so we can have
//           multiple sizes of brick.
//
// ************************************************

#include <pftypes.h>
#include <pfman.h>
#include <pfman_excl.h>

surface brick(
    output varying Color px_rc_co[3],

    varying Color px_rc_cl[3],
    varying transform_as_vector unit Short px_rc_l[3],

    varying transform_as_vector unit Short px_rc_eye[3],

    varying transform_as_normal unit Short px_material_normal[3],
    varying transform_as_texture TextureCoord px_shader_texcoord[2],

    // control for basic brick
    //   <--width--> |-- height
    // -> <- mortar  V
```

```
    //   X+--------+
    //   X|        |   |- mortar
    //   X+--------+   V
    //   XXXXXXXXXX
    //               ^ ^
    //               | |
    uniform float width = 0.25,
    uniform float height = 0.1,
    uniform float mortar = 0.01,
    uniform float brick_color[3] = {0.8,0.1,0.1},
    uniform float mortar_color[3] = {0.5,0.5,0.5})
{

    // figure out which row of bricks this is
    uInt1 row = px_shader_texcoord[1] / height;

    // offset even rows by half a row
    // this could potentially cause the texcoord to become larger than 1
    unsigned fixed<17,16> u_texcoord = px_shader_texcoord[0];
    if (row % 2 < 1)
        u_texcoord += width/2;

    // now safe to find column of bricks as well
    uInt1 col = u_texcoord / width;

    // compute "brick coordinates"
    TextureCoord st[2];
    st[0] = u_texcoord % width;
    st[1] = px_shader_texcoord[1] % height;

    // surface color will be either brick or mortar
    float surface_color[3] = brick_color;
    if (st[0] < mortar  ||  st[1] < mortar)
        surface_color = mortar_color;
    // compute vector from surface to eye and flip normal to face it
    float v[3] = normalize(px_rc_eye - getPs());
    if (px_material_normal * v < 0)
        px_material_normal = -px_material_normal;
    px_material_normal = normalize(px_material_normal);

    // accumulate diffuse colors for each light
    float diffuse[3] = {0,0,0};
    illuminance() {
        float l[3] = normalize(px_rc_l);
        varying float n_dot_l = px_material_normal * l;
```

```
        // no negative light on the back side of the object
        if (n_dot_l < 0)
            n_dot_l = 0;

        diffuse += n_dot_l * px_rc_cl;
    }

    // build final color from accumulated intensities and clamp to 1
    Short color_out[3];
    color_out[0] = diffuse[0]*surface_color[0];
    color_out[1] = diffuse[1]*surface_color[1];
    color_out[2] = diffuse[2]*surface_color[2];

    color_out = clamp(color_out, 0, 0.999);

    px_rc_co = color_out;
}
```

A.1.2. Full brick shader

```
// ************************************************
//
//     More complex and realistic looking brick shader.
//
// ************************************************

#include <pftypes.h>
#include <pfman.h>
#include <pfman_excl.h>

surface brick3(
    // output color
    output varying Color px_rc_co[3],

    // light color and direction inside illuminance
    varying Color px_rc_cl[3],
    varying transform_as_vector unit Short px_rc_l[3],

    // eye position
    varying transform_as_vector unit Short px_rc_eye[3],

    // surface normal, texture coordinates, tangent vectors &
    // pixel coverage area in texture space
    varying transform_as_normal unit Short px_material_normal[3],
    varying transform_as_texture TextureCoord px_shader_texcoord[2],
```

```
    varying transform_as_normal Short px_bump_grad_vecs[2][3],
    varying TextureFSqr px_shader_f_sqr,
    float mip_scale = 1,

    // control for basic brick
    uniform float brick_width = 0.25,
    uniform float brick_height = 0.1,
    uniform float mortar_width = 0.01,
    uniform float mortar_height = 0.01,
    uniform float brick_color[3] = {0.8,0.1,0.1},
    uniform float mortar_color[3] = {0.5,0.5,0.5},

    // add symmetric high frequency noise
    uniform float brick_hfnoise_mix = 0,
    uniform float brick_hfnoise_freq = 20,
    uniform float mortar_hfnoise_mix = 0,
    uniform float mortar_hfnoise_freq = 20,

    // add low frequency to bricks
    uniform float brick_lfnoise_scale = 0,
    uniform float brick_lfnoise_u_freq = 10,
    uniform float brick_lfnoise_v_freq = 2,

    // add inward bump to mortar
    uniform float mortar_bump_scale = 0,

    // color individual bricks differently
    uniform float brick_color_scale = 0)
{
    // figure out which row of bricks this is
    uInt1 row = px_shader_texcoord[1] / brick_height;

    // offset even rows by half a row
    // this could potentially cause the texcoord to become larger than 1
    unsigned fixed<17,16> u_texcoord = px_shader_texcoord[0];
    if (row % 2 < 1)
        u_texcoord += brick_width/2;

    // now safe to find column of bricks as well
    uInt1 col = u_texcoord / brick_width;

    // compute "brick coordinates"
    float st[2] = {
        u_texcoord % brick_width,
        px_shader_texcoord[1] % brick_height
```

```
};

// perturbation vector to put each brick in its own place in
// noise space.
uInt1 brick_shift[2] = {
    (uInt1)irandom((uInt2)row),
    (uInt1)irandom((uInt2)col)
};

// surface color will be either brick or mortar
Short mortar[3] = mortar_color;
Short brick[3] = brick_color;

// distance in texture space between adjacent pixels for
// antialiasing
float duv = mip_scale * sqrt((float)px_shader_f_sqr);

// add in high frequency noise to mortar
if (mortar_hfnoise_mix > 0) {
    mortar += {1,1,1} * mortar_hfnoise_mix
        * (1-smoothstep(1/mortar_hfnoise_freq,
                        2/mortar_hfnoise_freq, duv))
         * snoise((float[2])(mortar_hfnoise_freq
         * px_shader_texcoord));
}


// do individual brick color variations
if (brick_color_scale > 0)
    brick *= 1 + brick_color_scale
        * srandom((uInt2)(col + brick_shift[0]))
        * (1-smoothstep(brick_width/2, 2*brick_width, duv))
        * (1-smoothstep(brick_height/2, 2*brick_height, duv));


// add in high frequency noise to the brick
if (brick_hfnoise_mix > 0) {
    brick += {1,1,1} * brick_hfnoise_mix
        * (1-smoothstep(1/brick_hfnoise_freq,
                        2/brick_hfnoise_freq, duv))
        * snoise(brick_hfnoise_freq * st + brick_shift);
}

// add low frequency noise
if (brick_lfnoise_scale > 0) {
```

```
    float lf_texcoord[2] = {
        brick_lfnoise_u_freq * st[0], brick_lfnoise_v_freq * st[1]
    } + brick_shift;

    brick *= 1 + brick_lfnoise_scale
        * (1-smoothstep(1/brick_lfnoise_u_freq,
                        2/brick_lfnoise_u_freq, duv))
        * (1-smoothstep(1/brick_lfnoise_v_freq,
                        2/brick_lfnoise_v_freq, duv))
        * snoise(lf_texcoord);
}


// choose mortar or brick colors based on position in
// brick coordinates
float colormix =
    (smoothstep(mortar_width/2 - duv,
                (varying)mortar_width/2, st[0])
     - smoothstep((varying)(brick_width - mortar_width/2),
                  brick_width - mortar_width/2 + duv, st[0]))
    * (smoothstep(mortar_height/2 - duv,
                  (varying)mortar_height/2, st[1])
       - smoothstep((varying)(brick_height - mortar_height/2),
                    brick_height - mortar_height/2 + duv, st[1]));
float surface_color[3] = mix(mortar, brick, colormix);


////////////////////////////
// bump edges of mortar
float ndisp[2];

// create blend variables -- 0-1 from center of mortar to brick
float left_blend = st[0] / (mortar_width/2);
float right_blend = (brick_width - st[0]) / (mortar_width/2);
float bottom_blend = st[1] / (mortar_height/2);
float top_blend = (brick_height - st[1]) / (mortar_height/2);

// add displacements for each edge
float blend = left_blend;
if (blend > bottom_blend || blend > top_blend || blend >= 1)
    blend = 0;
ndisp[0] = mix(0, -mortar_bump_scale, blend);

blend = right_blend;
if (blend > bottom_blend || blend > top_blend || blend >= 1)
    blend = 0;
```

```
ndisp[0] += mix(0, mortar_bump_scale, blend);

blend = bottom_blend;
if (blend > left_blend || blend > right_blend || blend >= 1)
    blend = 0;
ndisp[1] = mix(0, -mortar_bump_scale, blend);

blend = top_blend;
if (blend > left_blend || blend > right_blend || blend >= 1)
    blend = 0;
ndisp[1] += mix(0, mortar_bump_scale, blend);

// fade bump away to antialias
ndisp[0] *= (1-smoothstep(mortar_width/2, 2*mortar_width, duv));
ndisp[1] *= (1-smoothstep(mortar_height/2, 2*mortar_width, duv));


// compute vector from surface to eye and flip normal to face it
float v[3] = normalize(px_rc_eye - getPs());
if (px_material_normal * v < 0)
    px_material_normal = -px_material_normal;

// bump normal direction
px_material_normal = normalize(px_material_normal);
px_bump_grad_vecs[0] = normalize(px_bump_grad_vecs[0]);
px_bump_grad_vecs[1] = normalize(px_bump_grad_vecs[1]);
px_material_normal += ndisp * px_bump_grad_vecs;
px_material_normal = normalize(px_material_normal);


// accumulate diffuse colors for each light
float diffuse[3] = {0,0,0};
illuminance() {
    float l[3] = normalize(px_rc_l);
    varying float n_dot_l = px_material_normal * l;

    // no negative light on the back side of the object
    if (n_dot_l < 0)
        n_dot_l = 0;

    diffuse += n_dot_l * px_rc_cl;
}

// build final color from accumulated intensities and clamp to 1
Short color_out[3];
```

```
    color_out[0] = diffuse[0]*surface_color[0];
    color_out[1] = diffuse[1]*surface_color[1];
    color_out[2] = diffuse[2]*surface_color[2];

    color_out = clamp(color_out, 0, 0.999);

    px_rc_co = color_out;
}
```

A.1.3. Rippled reflection

```
// Water ripple shader
//
// Produces water ripples from drips on a calm surface, rendered as
// a bump map.
//
// Handles up to MAX_DROPS at a time, drop_* parameters control the
// individual features of each drop (drop location and starting time)
//
// Time is given by frame_number parameter.
//
// Other control parameters control the general characteristics of the
// ripples (ripple_*: height, speed, wavelength, and two kinds of
// fading)
//
// See comments in the declaration section of surface ripple for more
// details

#include <pftypes.h>
#include <pfman.h>
#include <pfman_excl.h>

#define MAX_DROPS 5


// do texture lookup for environment map.  Parameters are first of 6
// consecutive texture ids (for the maps) and a reflection vector.  The
// reflection vector is destroyed.
Color[3] env_lookup(
    // texture id of first environment map texture
    // texture ids should be texture_id to texture_id+5, corresponding
    // to images in the +x, -x, +y, -y, +z, and -z directions
    // orientation for these images:
    //    +x: (u = +z, v = +y);  -x: (u = -z, v = -y)
    //    +y: (u = +x, v = +z);  -y: (u = -x, v = -z)
    //    +z: (u = +x, v = +y);  -z: (u = -x, v = -y)
```

```
    uniform float texture_id,

    // reflection vector off of surface into environment
    varying float reflection_vector[3]);


// approximate exponential exp(-x) decay with piece of quartic
// decay = 1 at x=0
// decay and 3 derivatives = 0 at x=6
// decay = 0 at all x>6
varying float decay(varying float x)
{
    if (x > 6)
        return 0;
    else {
        // quartic approximation: ((x-6)^4)/1296
        float t = x-6, t2=t*t, t4=t2*t2;
        return t4/1296;
    }
}


// compute normal displacement for ripples from one drop
float[2] ripple_displacement(
    // location that drop started
    uniform float drop_center[2],

    // distance traveled by wave front so far
    uniform float wavefront_progress,

    // position on surface
    varying TextureCoord texture_coordinates[2],

    // ripple parameters
    uniform float ripple_frequency,     // in radians/texel
    uniform float ripple_height,        // height of ripples
    uniform float ripple_source_decay,  // decay at source
                                        //   (in texels)
    uniform float ripple_front_decay    // decay at wave front
                                        //   (in texels)
    )
{
    // distance of this sample from the drop center
    float drop_vector[2] = texture_coordinates - drop_center;
    float drop_distance = sqrt(drop_vector[0]*drop_vector[0] +
```

```
                            drop_vector[1]*drop_vector[1]);

    // length of ripples that have passed this sample
    float ripple_progress = wavefront_progress - drop_distance;
    if (ripple_progress < 0) ripple_progress = 0;

    // normal displacement in direction ripple is moving
    // basically the derivative of the ripple height function
    //   (should really have some terms due to the derivative of each
    //   decay function, but I ignore these)
    float normal_displacement = ripple_height
        * decay(drop_distance * ripple_front_decay)
        * decay(ripple_progress * ripple_source_decay)
        * fastsin(ripple_progress * ripple_frequency);

    return(normal_displacement/drop_distance * drop_vector);
}


surface ripple2(
    // output color
    output varying Color px_rc_co[3],

    // light color and vector to light (valid inside illuminance)
    varying Color px_rc_cl[3],
    varying transform_as_vector unit Short px_rc_l[3],

    // vector from surface to eye
    varying transform_as_vector unit Short px_rc_eye[3],

    // surface description:
    // normal, texture coordinates, and texture gradient vectors
    varying transform_as_normal unit Short px_material_normal[3],
    varying transform_as_texture TextureCoord px_shader_texcoord[2],
    varying transform_as_normal Short px_bump_grad_vecs[2][3],

    // material parameters
    varying Color px_material_diffuse[3] = {.2,.9,.8},
    uniform float px_material_specular[3] = {1,1,1},
    uniform float px_material_shininess = 50,

    // environment map texture ids
    uniform float environment_texture = 0,

    // current frame
```

```
    uniform float time = 0,

    // ripple parameters
    uniform float ripple_height = .5,       // amplitude of ripples
    uniform float ripple_speed = 0.01,      // in texels/second
    uniform float ripple_wavelength = 0.05, // in texels
    uniform float ripple_source_fade = 18,  // source will be 0 when
                                            // wave has traveled x
                                            // wavelengths
    uniform float ripple_front_fade = 50,   // wave front will be 0
                                            // after it has traveled
                                            // x wavelengths

    // drop locations and start times
    uniform float num_drops = 3,
    uniform float drop_center[MAX_DROPS][2] =
            {{.5,.5}, {.25,.25}, {.75,.97}, {.33,.66}, {0,0}},
    uniform float drop_start[MAX_DROPS] = {10,30,35,500,510},

    uniform float view_xform[3][3] = {{1,0,0},{0,1,0},{0,0,1}}
    )
{
    // conversion from easy to specify quantities to easy to use
    //   versions frequency (in radians/texel)
    uniform float ripple_frequency = 2*PI/ripple_wavelength;
    //   decay at source (in 1/texels)
    //     factor of 6 is because exp(-x) is effectively 0 by x = 6
    //     (and my approximation is exactly 0 by x = 6)
    uniform float ripple_source_decay
        = 6/(ripple_wavelength*ripple_source_fade);
    //   decay at wave front with distance from source (in 1/texels)
    uniform float ripple_front_decay
        = 6/(ripple_wavelength*ripple_front_fade);
    //   maximum progress of ripple
    uniform float ripple_end
        = (ripple_source_fade + ripple_front_fade) * ripple_wavelength;




    float normal_displacement[2] = {0,0};   // displacement of normal

    // displace from each drop
    uniform uInt4 i;
    for(i=0; i<num_drops; ++i) {
```

```
    // distance traveled by wave front so far
    uniform float wavefront_progress =
        (time - drop_start[i]) * ripple_speed;

    if (wavefront_progress > 0  &&  wavefront_progress < ripple_end)
        normal_displacement += ripple_displacement(
            drop_center[i], wavefront_progress, px_shader_texcoord,
            ripple_frequency, ripple_height,
            ripple_source_decay, ripple_front_decay);
}

// modify normal by bump displacements
px_material_normal = normalize(px_material_normal);
px_bump_grad_vecs[0] = normalize(px_bump_grad_vecs[0]);
px_bump_grad_vecs[1] = normalize(px_bump_grad_vecs[1]);
px_material_normal += normal_displacement * px_bump_grad_vecs;

px_material_normal = normalize(px_material_normal);


// compute lighting
float v[3] = normalize(px_rc_eye - getPs());
float two_n_dot_v = 2*(px_material_normal * v);
if (two_n_dot_v < 0) {
    two_n_dot_v = -two_n_dot_v;
    px_material_normal = -px_material_normal;
}

// accumulate diffuse and specular colors for each light
float diffuse[3] = {0,0,0}, specular[3] = {0,0,0};
illuminance() {
    float l[3] = normalize(px_rc_l);
    float n_dot_l = px_material_normal * l;
    float v_dot_l = v * l;

    // approximation (1 - (D.D/2)(n/K))^K;  D = R-L;  K = 4
    //   from [Lyon93]
    float spec = 1 + v_dot_l - two_n_dot_v * n_dot_l;
    spec = 1 - spec * (px_material_shininess / 4);
    if (spec < 0) spec = 0;
    spec *= spec;
    spec *= spec;

    // no negative light on the back side of the object
    if (n_dot_l < 0)
        n_dot_l = spec = 0;
```

```
        // accumulate the two intensities
        diffuse += n_dot_l * px_rc_cl;
        specular += spec * px_rc_cl;
    }

    // do environment map lookup
    float reflection[3];
    if (environment_texture) {
        reflection = view_xform * (two_n_dot_v * px_material_normal
                                   - v);

        // add in environment map color
        specular += env_lookup(environment_texture, reflection);
    }


    // build final color from accumulated intensities and clamp to 1
    Short color_out[3];
    for(i = 0; i < 3; i++)
        color_out[i] = diffuse[i]*px_material_diffuse[i]
            + specular[i]*px_material_specular[i];

    px_rc_co = clamp(color_out, 0, 0.999);
}
```

A.1.4. Environment map

```
// simple environment map shader

#include <pftypes.h>
#include <pfman.h>
#include <pfman_excl.h>

// do texture lookup for environment map.  Parameters are first of 6
// consecutive texture ids (for the maps) and a reflection vector.  The
// reflection vector is destroyed.
Color[3] env_lookup(
    // texture id of first environment map texture
    // texture ids should be texture_id to texture_id+5, corresponding
    // to images in the +x, -x, +y, -y, +z, and -z directions
    // orientation for these images:
    //    +x: (u = +z, v = +y);  -x: (u = -z, v = -y)
    //    +y: (u = +x, v = +z);  -y: (u = -x, v = -z)
    //    +z: (u = +x, v = +y);  -z: (u = -x, v = -y)
    uniform float texture_id,
```

```
    // reflection vector off of surface into environment
    varying float reflection_vector[3])
{
    //
    // pick a cube face

    // first figure out which direction is largest
    // and swap to make it reflection_vector[2];
    float rabs[3] = reflection_vector;
    rabs[0] = abs(rabs[0]);
    rabs[1] = abs(rabs[1]);
    rabs[2] = abs(rabs[2]);

    // reflection_vector is already good if z is largest
    // set tid to first of pair of z textures
    uInt2 tid = texture_id + 4;

    // x largest, swap to z-y-z order and fix tid
    if (rabs[0] > rabs[1]  &&  rabs[0] > rabs[2]) {
        reflection_vector = {reflection_vector[2],
                             reflection_vector[1],
                             reflection_vector[0]};

        tid -= 4;
    }

    // y largest, swap to x-z-y order and fix tid
    if (rabs[1] > rabs[0]  &&  rabs[1] > rabs[2]) {
        reflection_vector = {reflection_vector[2],
                             reflection_vector[0],
                             reflection_vector[1]};

        // also switch tid to first of y textures
        tid -= 2;
    }

    // figure out whether + face or -
    if (reflection_vector[2] < 0) tid += 1;

    // project onto face;
    TextureCoord texcoord[2] = {
        (1 + reflection_vector[0]/reflection_vector[2]),
        (1 + reflection_vector[1]/reflection_vector[2])
    }/2;
```

```
    return texture(tid, texcoord);
}


surface env(
    // output color
    varying output Color px_rc_co[3],

    // surface normal
    varying transform_as_vector unit Short px_rc_eye[3],

    // surface normal
    varying transform_as_normal unit Short px_material_normal[3],

    // texture IDs for environment map
    uniform float texture_id,

    // transform
    uniform float view_xform[3][3] = {{1,0,0},{0,1,0},{0,0,1}}
    )
{
    // compute reflection vector
    float v[3] = px_rc_eye - getPs();
    v /= sqrt(v*v);
    px_material_normal = normalize(px_material_normal);
    float reflection[3] = view_xform * (
        2*(v*px_material_normal)*px_material_normal - v);

    px_rc_co = env_lookup(texture_id, reflection);
}
```

A.1.5. Mandelbrot and Julia sets

```
// A combined Mandelbrot/Julia set shader using high-resolution.
// coordinates. Both Mandelbrot and Julia set work by iterating
//   z = z * z  +  c
// with complex z and c.  Points in the set never go to infinity, and
// it's been proven that once |z| >= 2, z will go to infinity.  So, we
// can't really find points in the set, but we can iterate for a while
// to find points we know are not in the set.  We color each point by
// how many iterations it took to leave the |z| < 2 circle.
//
// For the Mandelbrot set, c is the point on the plane and z starts
// at 0. For the Julia set, z is the point on the plane and c is a
// constant (which controls the shape of the set).  To allow this
```

```
// function to handle both sets, we have two potentially varying
// variables: mand_base, which is the initial value for z, and
// mand_offset, which is the value for c.

#include <pftypes.h>

surface mandf(
    output varying Color px_rc_co[3],
    // allow coordinates from -2 to 2: 2 integer bits
    varying fixed<61,54> mand_offset[2],
    varying fixed<31,27> mand_base[2],
    uniform float mand_colormap[16][3],
    uniform float mand_loops,
    uniform float mand_use_float)
{
    if (mand_use_float) {

        uniform uInt4 iter;
        varying uInt4 done = 0;

        // real and imaginary components of current iteration
        varying float r = mand_base[0], i = mand_base[1];
        varying float x = mand_offset[0], y = mand_offset[1];

        // r^2 and i^2
        varying float r2 = r*r, i2 = i*i;

        for (iter = 1; iter <= mand_loops; iter++) {
            i = 2*r*i + mand_offset[1];  // (r,i) = (r,i)*(r,i) + (x,y)
            r = r2 - i2 + mand_offset[0];
            r2 = r*r;
            i2 = i*i;

            // if outside set, then save iteration number
            varying float t = r2+i2;
            done = done ? done : (t>4 ? iter : 0);
        }

        Bit in_set = (done == 0);

        // set color from color ramp
        done = done & 15;
        for (iter = 0; iter < 16; iter++)
            if (done == iter)
                px_rc_co = mand_colormap[iter];
```

```
        // white inside set
        if (in_set)
            px_rc_co = {.999,.999,.999};
    }

    // fixed point version
    else {

        uniform uInt4 iter;
        varying uInt4 done = 0;

        // real and imaginary components of current iteration
        // if old |r,i| < 2, new |r,i| < 10: 4 integer bits
        varying fixed<31,27> r = mand_base[0], i = mand_base[1];
        varying fixed<61,54> x = mand_offset[0], y = mand_offset[1];

        // r^2 and i^2
        // if r,i < 10, r2,i2 < 100: 7 integer bits
        varying fixed<61,54> r2 = r*r, i2 = i*i;

        for (iter = 1; iter <= mand_loops; iter++) {
            i = 2*r*i + mand_offset[1];  // (r,i) = (r,i)*(r,i) + (x,y)
            r = r2 - i2 + mand_offset[0];
            r2 = r*r;
            i2 = i*i;

            // if outside set, then save iteration number
            varying fixed<62,54> t = r2+i2;
            done = done ? done : (t>4 ? iter : 0);
        }

        Bit in_set = (done == 0);

        // set color from color ramp
        done = done & 15;
        for (iter = 0; iter < 16; iter++)
            if (done == iter)
                px_rc_co = mand_colormap[iter];

        // white inside set
        if (in_set)
            px_rc_co = {.999,.999,.999};
    }
}
```

A.1.6.Wood

```
// this shader produces a 3D wood texture.
// based on Pixar's wood shader [Upstill90]

#include <pftypes.h>
#include <pfman.h>
#include <pfman_excl.h>

sFrac2 snoise(varying float t);
sFrac2 snoise(varying float t[2]);
sFrac2 snoise(varying float t[3]);
sFrac2 snoise(varying float x);
sFrac2 snoise(varying float x, varying float y);
sFrac2 snoise(varying float x, varying float y, varying float z);
uFrac2 noise(varying float x);
uFrac2 noise(varying float x, varying float y);
uFrac2 noise(varying float x, varying float y, varying float z);

surface wood(
    // output color
    output varying Color px_rc_co[3],

    // light color and direction
    varying Color px_rc_cl[3],
    varying transform_as_vector unit Short px_rc_l[3],

    // location of eye
    varying transform_as_vector unit Short px_rc_eye[3],

    // surface normal
    varying transform_as_normal unit Short px_material_normal[3],

    // parameters controlling wood itself

    // distance between growth rings (larger = closer rings)
    float grain = 5,

    // amount of turbulence
    float swirl = .25,
    float swirlfreq = 1,

    // point and vector defining axis line of tree
    float axis_point[3] = {0,0,0},
    float axis_vec[3] = {0,0,1},
```

```
    // object space transformation
    float view_xform[3][3] = {{1,0,0},{0,1,0},{0,0,1}},
    float view_xlate[3] = {0,0,0},

    // colors (darkcolor = -1,-1,-1 means use 30% of
    //   px_material_diffuse)
    float darkcolor[3] = {-1,-1,-1},
    float px_material_diffuse[3] = {.8,.7,.2},
    float px_material_specular[3] = {1,1,1},
    float px_material_shininess = 10)
{
    float P[3] = view_xform * getPs();
    P += view_xlate;

    // compute extra offset for turbulence
    float swirlpt[3] = swirlfreq * P;
    float swirl_distance = 0;
    swirl_distance += snoise(   swirlpt);
    swirl_distance += snoise( 2*swirlpt)/2;
    swirl_distance += snoise( 4*swirlpt)/4;
    swirl_distance += snoise( 8*swirlpt)/8;
    swirl_distance += snoise(16*swirlpt)/16;

    // compute distance from surface position to tree axis
    float axis[3] = axis_vec / length(axis_vec);
    float distance = length(axis ^ (P - axis_point))
        + swirl*swirl_distance;

    // compute the scale factor for the grain
    distance = (grain*distance) % 1;   // scale distance for grain size
    float alpha = distance * distance;
    alpha = 2*((1 - alpha)/5 + .3 - .3*snoise(8*distance));

    // finally, compute the wood surface color
    if (darkcolor[0] < 0) darkcolor = .3*px_material_diffuse;
    float surface_color[3] = mix(darkcolor, px_material_diffuse, alpha);

    // compute view vector and forward-facing surface normal
    Short V[3] = normalize(px_rc_eye - getPs());
    Short Nf[3] = normalize(px_material_normal);
    Short two_n_dot_v = 2 * Nf * V;
    if (two_n_dot_v < 0) {
        two_n_dot_v = -two_n_dot_v;
        Nf = -Nf;
    }
```

```
    // accumulate the Phong shading
    Short diffuse[3] = {0,0,0}, specular[3] = {0,0,0};

    illuminance() {
        // normalize light vector
        Short L[3] = normalize(px_rc_l);

        // dot product of unit normal and unit light vector
        Short n_dot_l = Nf * L;

        // dot product of unit view vector and unit light vector
        Short v_dot_l = V * L;

        // specular contribution
        //   approximation (1 - (D.D/2)(n/K))^K;  D = R-L;  K = 4
        //   from [Lyon93]
        float spec = 1 + v_dot_l - two_n_dot_v * n_dot_l;// D.D / 2
        spec = 1 - spec * (px_material_shininess / 4);
        if (spec < 0) spec = 0;
        spec *= spec;
        spec *= spec;

        // don't want negative light on the back sides of things!
        if (n_dot_l < 0)
            n_dot_l = spec = 0;

        // add in diffuse and specular contributions
        diffuse += px_rc_cl * n_dot_l;
        specular += px_rc_cl * spec;
    }

    // build final color from accumulated intensities
    Short color_out[3];
    uniform uInt4 i;
    for(i=0; i<3; i++)
        color_out[i] = diffuse[i]*surface_color[i] +
            specular[i]*px_material_specular[i];

    // copy into REAL output color
    px_rc_co = clamp(color_out, 0, 0.999);
}
```

A.2. Light

A.2.1. Light through a window

```
// Light shader simulating sunlight through a window
// based on windowlight.sl distributed with RenderMan

#include <pftypes.h>
#include <pfman.h>
#include <pfman_excl.h>

light win(output varying Color px_rc_cl[3],
          output varying Short px_rc_l[3],

          // light direction
          uniform transform_as_point float px_light_position[3]
              = {-1,1,2},
          // center of window
          uniform transform_as_point float Center[3] = {0,0,0},
          // direction from center into room
          uniform transform_as_point float In[3] = {0,0,1},
          // direction from center to the top of the window
          uniform transform_as_vector float Up[3] = {0,1,0},

          // color of light inside and outside of each pane
          uniform float px_light_diffuse[3] = {.999,.2,.2},
          uniform float darkcolor[3] = {.3,.2,.1},

          /* number of panes horizontally */
          uniform float xorder    =  3,
          /* number of panes vertically   */
          uniform float yorder    =  4,
          /* horizontal size of a pane    */
          uniform float panewidth =  .7,
          /* vertical size of a pane      */
          uniform float paneheight =  1,
          /* sash width between panes, % of pane size     */
          uniform float framewidth =  0.07,
          /* transition region between pane and sash */
          uniform float fuzz    = .01
    )
{
    // normalize light direction
    px_rc_l = px_light_position / length(px_light_position);

    // compute coordinate frame for window
    uniform float in[3] = In/length(In);
```

```
uniform float right[3];        // = Up ^ in
right[0] = Up[1]*in[2] - Up[2]*in[1];
right[1] = Up[2]*in[0] - Up[0]*in[2];
right[2] = Up[0]*in[1] - Up[1]*in[0];
right /= length(right);

uniform float up[3];        // = in ^ right
up[0] = in[1]*right[2] - in[2]*right[1];
up[1] = in[2]*right[0] - in[0]*right[2];
up[2] = in[0]*right[1] - in[1]*right[0];
up /= length(up);

// lower left corner of wall with window
uniform float corner[3] =
    Center - right*(xorder*panewidth/2) - up*(yorder*paneheight/2);

// vector from surface to corner of window
float p_to_c[3] = corner - getPs();

// vector from surface to corner of window in direction light
// is shining
float p_to_f[3] = px_rc_l * ((p_to_c*in)/(px_rc_l*in));

// vector from the corner of the window to intersection point
// on the window
float c_to_loc[3] = p_to_f - p_to_c;

// 1 = lightcolor, 0 = darkcolor.  Start out at 1
float lightness = 1;

// project c_to_loc onto up vector to get vertical offset
float offset = c_to_loc * up;

// are we inside the window?
if(offset>0 && offset < yorder*paneheight){
    Short vpaneLoc=(offset/paneheight) % 1;

    if(vpaneLoc > .5) // exploit the symmetry in the pane
        vpaneLoc = 1 - vpaneLoc;

    // include the sash fuzz
    lightness *= smoothstep(framewidth/2 - fuzz/2,
                            framewidth/2 + fuzz/2,
                            vpaneLoc);
}
```

```
    else
        lightness = 0;


    // project c_to_loc onto right vector to get horizontal offset
    offset = c_to_loc * right;
    px_rc_l[1] = offset;

    // are we inside the window?
    if(offset > 0 && offset < xorder*panewidth){
        Short hpaneLoc = (offset/panewidth) % 1;

        if(hpaneLoc > .5)
            hpaneLoc = 1 - hpaneLoc;

        //include the sash fuzz
        lightness *= smoothstep(framewidth/2 - fuzz/2,
                                framewidth/2 + fuzz/2,
                                hpaneLoc);
    }
    else
        lightness = 0;

    px_rc_cl = mix(darkcolor,px_light_diffuse,lightness);
}
```

A.3. Primitives

A.3.1. Simple direct rendered primitive

```
// simple test primitive that produces a flat “wiggly square”

#include <pfman.h>
#include <pfman_excl.h>


primitive testprim(
    varying uFrac2 px_shader_z,
    transform_as_point float center[1][4],
    float size[1])
{
    uInt2 x = linear_expri(center[0][3], 0, -center[0][0]);
    uInt2 y = linear_expri(0, center[0][3], -center[0][1]);
    uInt4 x2 = x*x, y2 = y*y;
    uniform float r = size[0]*center[0][3], r2 = r*r, r4 = r2*r2;

    // compare against silhouette
```

```
    if (x2*x2 + y2*y2 < r4(1 + x2 + y2)) {

        // compare against Z
        uFrac2 z = center[0][2]/center[0][3];
        if (px_shader_z > z) {
            px_shader_z = z;

            // do interpolation
            interpolation {
                // no built-in interpolator types for this primitive
            }
        }
    }
}
```

A.3.2. Basic triangle without clipping

```
// simple triangle implementing the algorithm in [Olano97]
#include <pfman.h>
#include <pfman_excl.h>

primitive tri_2dh(
    varying unsigned fixed<32,32> px_shader_z,
    interpolation_data interp_data[3],
    transform_as_point float vertex[3][4])
{
    // invert matrix of 2D homogeneous vertex coordinates (e.g. [x y w])
    uniform float edge[3][3];
    edge[0][0] = vertex[1][1] * vertex[2][3]
               - vertex[1][3] * vertex[2][1];
    edge[1][0] = vertex[1][3] * vertex[2][0]
               - vertex[1][0] * vertex[2][3];
    edge[2][0] = vertex[1][0] * vertex[2][1]
               - vertex[1][1] * vertex[2][0];
    edge[0][1] = vertex[2][1] * vertex[0][3]
               - vertex[2][3] * vertex[0][1];
    edge[1][1] = vertex[2][3] * vertex[0][0]
               - vertex[2][0] * vertex[0][3];
    edge[2][1] = vertex[2][0] * vertex[0][1]
               - vertex[2][1] * vertex[0][0];
    edge[0][2] = vertex[0][1] * vertex[1][3]
               - vertex[0][3] * vertex[1][1];
    edge[1][2] = vertex[0][3] * vertex[1][0]
               - vertex[0][0] * vertex[1][3];
    edge[2][2] = vertex[0][0] * vertex[1][1]
               - vertex[0][1] * vertex[1][0];
```

```
uniform float det =
      vertex[0][0] * edge[0][0]
    + vertex[0][1] * edge[1][0]
    + vertex[0][3] * edge[2][0];
edge /= det;

// use matrix columns as coefficients for linear edge tests
// scaling factors avoid edge overflow
uniform float s0 =
    1/sqrtf(edge[0][0]*edge[0][0]+edge[1][0]*edge[1][0]);
uniform float s1 =
    1/sqrtf(edge[0][1]*edge[1][1]+edge[1][1]*edge[1][1]);
uniform float s2 =
    1/sqrtf(edge[0][2]*edge[0][2]+edge[1][2]*edge[1][2]);
if (linear_expri(s0*edge[0][0],s0*edge[1][0],s0*edge[2][0]) >= 0 &&
    linear_expri(s1*edge[0][1],s1*edge[1][1],s1*edge[2][1]) >= 0 &&
    linear_expri(s2*edge[0][2],s2*edge[1][2],s2*edge[2][2]) >= 0) {

    // multiply Z at each vertex by matrix to get coefficients
    // for linear expression for Z test
    uFrac4 z = linear_exprz(
        edge[0][0]*vertex[0][2] +
        edge[0][1]*vertex[1][2] +
        edge[0][2]*vertex[2][2],

        edge[1][0]*vertex[0][2] +
        edge[1][0]*vertex[1][2] +
        edge[1][2]*vertex[2][2],

        edge[2][0]*vertex[0][2] +
        edge[2][1]*vertex[1][2] +
        edge[2][2]*vertex[2][2]);

    if (px_shader_z > z) {
        px_shader_z = z;

        // do interpolation
        interpolation {
            // only option is linear interpolation
            case PXlinear:
                interpolator_return (
                    linear_expr(edge[0]*interp_data,
                        edge[1]*interp_data,
                        edge[2]*interp_data));
        }
    }
```

```
    }
}
```

A.3.3. Subdivision primitive

```
// simple subdivision primitive
// input is three vertices and an offset direction
// subdivides triangle defined by the three vertices into four
// smaller triangles and offsets the new vertices

#include <pfman.h>
#include <pfman_excl.h>

// prototype for base primitive
primitive triangle(
    varying unsigned fixed<32,32> px_shader_z,
    transform_as_point float vertex[3][4]);

primitive subdivision(
    varying unsigned fixed<32,32> px_shader_z,
    transform_as_normal float offset_direction[3],
    transform_as_point float vertex[3][4])
{
    // create three new vertices
    // each is the average of two of the existing vertices
    uniform float new_vertex[3][4];
    new_vertex[0] = (vertex[1] + vertex[2])/2;
    new_vertex[1] = (vertex[2] + vertex[0])/2;
    new_vertex[2] = (vertex[0] + vertex[1])/2;

    // displace the new vertices by offset_direction
    new_vertex[0][0] += offset_direction[0]*new_vertex[0][3];
    new_vertex[0][1] += offset_direction[1]*new_vertex[0][3];
    new_vertex[0][2] += offset_direction[2]*new_vertex[0][3];

    new_vertex[1][0] += offset_direction[0]*new_vertex[1][3];
    new_vertex[1][1] += offset_direction[1]*new_vertex[1][3];
    new_vertex[1][2] += offset_direction[2]*new_vertex[1][3];

    new_vertex[2][0] += offset_direction[0]*new_vertex[2][3];
    new_vertex[2][1] += offset_direction[1]*new_vertex[2][3];
    new_vertex[2][2] += offset_direction[2]*new_vertex[2][3];

    // render four new triangles
    uniform float new_tri[3][4];
    new_tri[0] = vertex[0];
    new_tri[1] = new_vertex[1];
```

```
    new_tri[2] = new_vertex[2];
    triangle(px_shader_z, new_tri);

    new_tri[0] = vertex[1];
    new_tri[1] = new_vertex[2];
    new_tri[2] = new_vertex[0];
    triangle(px_shader_z, new_tri);

    new_tri[0] = vertex[2];
    new_tri[1] = new_vertex[0];
    new_tri[2] = new_vertex[1];
    triangle(px_shader_z, new_tri);

    triangle(px_shader_z, new_vertex);
}
```