Stanford Real-Time Procedural Shading System

SIGGRAPH 2001 Course Notes

William R. Mark, April 18, 2001

The Stanford real-time procedural shading system compiles shaders written in a high-level shading language to graphics hardware. In particular, the system can compile to graphics hardware with programmable vertex and fragment pipelines.

Some of the key features of the system are:

- The user writes shaders in a high-level, hardware-independent shading language.
- The shading language supports multiple computation frequencies. These computation frequencies – fragment, vertex, and primitive-group – map well to graphics hardware.
- The system uses a well-defined internal interface to support a variety of compiler back ends. A different compiler back end can be used for each computation frequency. Each compiler back end targets a particular hardware interface (e.g. register-combiner fragment hardware).
- The system includes compiler back ends that target programmable vertex and fragment hardware.

We have written two papers that discuss various aspects of our system:

- A Real-Time Procedural Shading System for Programmable Graphics Hardware. Kekoa Proudfoot, William R. Mark, Zvetoslav Tzvetkov, Pat Hanrahan. SIGGRAPH 2001. *This paper describes the complete system.*
- Compiling to a VLIW Fragment Pipeline. William R. Mark and Kekoa Proudfoot. Submitted to 2001 SIGGRAPH/Eurographics Workshop on Graphics Hardware. *This paper describes the system's compiler for the register-combiner architecture.*

The material in these course notes complements these publications. We have included the following:

1. An example shader (our bowling-pin shader), and the compiled code that our system produces for that shader. The compiled code is for a GeForce3 – it includes fragment code (register-combiner configuration), vertex code (NV_vertex_program code), and primitive-group code (X86 CPU code).
2. Documentation for our system's immediate-mode interface. This interface is used to specify and compile shaders; to specify geometry to be rendered; and to set shader parameters. This interface is a layer that runs on top of OpenGL.
3. An example program that uses our system's immediate-mode interface.
4. Documentation for our system's shading language, with a variety of example shaders.

Additional information is available on our project web page,
`http://graphics.stanford.edu/projects/shading`.

Bowling-Pin Shader and Functions Called by It

Bowling-Pin Surface Shader

```
//
// This shader does the complete bowling pin, and fits into a single pass
// on the GeForce3
//
surface shader float4
bowling_pin(texref basemarks, texref decals, texref bumps, float4 uv) {

    // Compute texture coordinates
    float4  uv_wrap  = { uv[0], 10 * Pobj[1], 0, 1 };
    float4  uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_basemarks  = invert(translate(2.0, -7.5, 0) * scale (4,  15, 1));
    float4  uv_basemarks = t_basemarks * uv_wrap;
    float4  uv_bumps     = uv_basemarks;
    matrix4 t_decals  = scale(0.5, 1, 1) *
                        invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    float4  uv_front  = t_decals * uv_label;
    float4  uv_back   = {1.0 - uv_front[0], uv_front[1], uv_front[2], 1};
    float   front     = select(Pobj[2] >= 0, 1, 0) * select(uv[0] > 3, 0, 1);
    float4  uv_decals = select(front==1, uv_front, uv_back);

    // Look up textures
    float4 Decals    = texture(decals,    uv_decals);
    float4 BaseMarks = texture(basemarks, uv_basemarks);
    float  Marks     = alpha(BaseMarks);
    float3 Base      = rgb(BaseMarks);

    // Compute color, primarily by calling separate 'lightmodel_bumps' routine
    float3 Ma = {.4,.4,.4};
    float3 Md = {.5,.5,.5};
    float3 Ms = {.3,.3,.3};
    float3 Kd = rgb((Decals over {Base, 1.0}) * Marks);
    float3 C = lightmodel_bumps(Kd * Ma, Kd * Md, Ms, bumps, uv_bumps);
    return {C, 1.0};
}
```

Light Shader

(the compiled code given later includes one instance of 'simple_light')

```
// helper function for light shader
light float atten (float ac, float al, float aq) {
    return 1.0 / (aq * Sdist * Sdist + al * Sdist + ac);
}

light shader float4 simple_light (float4 color, float ac, float al, float aq) {
    return color * atten(ac, al, aq);
}
```

Bump-map Function Called by Bowling-pin Shader

```
surface float3
lightmodel_bumps(float3 a, float3 d, float3 s, texref bumps, floatv uv_bumps) {

  // Compute normalized tangent-space light vectors
  vertex perlight float3 Ltan = tangentspace(L);
  vertex perlight float3 Htan = tangentspace(H);

  // Lookup from bump map
  float4 Nlookup  = texture(bumps, uv_bumps); // alpha has short len
  float3 Nbump    = 2.0*(rgb(Nlookup)-triple(0.5));
  float  N_avglen = Nlookup[3]; // Length of mipmap filtered N, before renorm

  // Diffuse
  //perlight float3 Lfrag = 2.0*(cubenorm(Ltan)-{.5,.5,.5});
  perlight float3 Lfrag  = signed_interpolate(Ltan);
  perlight float  NdotL  = dot(Nbump, Lfrag);
  perlight float  shadow = 4*(Lfrag[2] + Lfrag[2]);  // Geometric shadow ramp
  perlight float3 diff   = d * clamp01(NdotL) * clamp01(shadow) * N_avglen;

  // Specular
  perlight float3 Hlookup = cubenorm(Htan);
  perlight float3 Hnorm   = 2.0*(Hlookup-{.5,.5,.5});
  perlight float  NdotH   = clamp01(dot(Nbump, Hnorm));
  perlight float  NdotHs  = select(Hlookup[2] >= 0.5, NdotH, 0.0);
  perlight float  NdotH2  = NdotHs * NdotHs;
  perlight float  NdotH4  = NdotH2 * NdotH2;
  perlight float  NdotH8  = NdotH4 * NdotH4;
  perlight float3 spec    = NdotH8 * shadow * s;

  // Combine
  perlight float3 C = diff + spec;
  return integrate(rgb(Cl) * C) + a;
} // lightmodel_bumps
```

Other Functions Called by Bowling-pin Shader

```
surface float3
tangentspace(float3 V) {
  // Convert vector to tangent space, and normalize it
  float VtanX = dot(V,T);
  float VtanY = dot(V,B);
  float VtanZ = dot(V,N);
  return normalize({VtanX, VtanY, VtanZ});
}

// Clamp scalar to range [0,1]
surface clampf clamp01(float x) {return (clampf) x;}

// interpolate [-1,1] value from vert to frag for nv fragment backend
surface fragment float3 signed_interpolate(vertex float3 v) {
  vertex   float3 t = 0.5*(v+triple(1.0));
  return 2.0*(((fragment float3) t)-triple(0.5));
}
```

Compiler-Generated Fragment Code for Bowling-Pin Shader (Register Combiner Configuration)

```
CLAMPING NOTATION: [] = clamp to [0,1].  {} = clamp to [-1,1]

*** TEXTURE SHADER CONFIG ***
STAGE 0: TEXTURE_2D      TEXREF='decals'      COORD= 'uv_decals'
STAGE 1: TEXTURE_2D      TEXREF='basemarks'   COORD= 'uv_basemarks'
STAGE 2: TEXTURE_2D      TEXREF='bumps'       COORD= 'uv_bumps'
STAGE 3: TEXTURE_CM      TEXREF=CUBENORM      COORD= 'Htan'

***** GLOBAL PASS INPUTS *****
V0.rgb  = interpolate(0.5*(Ltan+{1,1,1}));
V1.rgb  = interpolate(Cl);
T0.rgba = TEXSHADE.rgba
T1.rgba = TEXSHADE.rgba
T2.rgba = TEXSHADE.rgba
T3.rgb  = TEXSHADE.rgb

************ RGB STAGE 0 **************                  ********** ALPHA STAGE 0 *************
T3.rgb = {L}      L = (2*[T2.rgb]-1) dot (2*[T3.rgb]-1)   S0.a = {L}     L = T3.b
T2.rgb = {R}      R = (2*[T2.rgb]-1) dot (2*[V0.rgb]-1)

************ RGB STAGE 1 **************                  ********** ALPHA STAGE 1 *************
                  L = T0.rgb                                             L = [Z0.a]
                  R = T1.rgb * (1-[T0.aaa])                              R = [T3.b]
T0.rgb = {M}      M = L + R                              V0.a = {M}     M = (S0.a < 0.5) ? L : R

************ RGB STAGE 2 **************                  ********** ALPHA STAGE 2 *************
T0.rgb = {L}      L = T0.rgb * T1.aaa                    V0.a = {L}     L = V0.a * V0.a
                                                         T0.a = {R}     R = T2.b

************ RGB STAGE 3 **************                  ********** ALPHA STAGE 3 *************
T1.rgb = {0.5*L}  L = T0.rgb                             V0.a = {L}     L = V0.a * V0.a

************ RGB STAGE 4 **************                  ********** ALPHA STAGE 4 *************
V0.rgb = {L}      L = T1.rgb * [T0.aaa]                                  L = (2*[V0.b]-1)
T1.rgb = {R}      R = V0.aaa * V0.aaa                                    R = (2*[V0.b]-1)
                                                         T1.a = {4*M}   M = L + R

************ RGB STAGE 5 **************                  ********** ALPHA STAGE 5 *************
V0.rgb = {L}      L = V0.rgb * [T1.aaa]                  V0.a = {L}     L = T1.b * T1.a

PER-STAGE PASS INPUTS FOR STAGE 6:
L0.rgb  = {0.300000, 0.300000, 0.300000}
************ RGB STAGE 6 **************                  ********** ALPHA STAGE 6 *************
                  L = V0.rgb * T2.aaa
                  R = V0.aaa * L0.rgb
V0.rgb = {M}      M = L + R

PER-STAGE PASS INPUTS FOR STAGE 7:
L0.rgb  = {0.400000, 0.400000, 0.400000}
************ RGB STAGE 7 **************                  ********** ALPHA STAGE 7 *************
                  L = V1.rgb * V0.rgb
                  R = T0.rgb * L0.rgb
V0.rgb = {M}      M = L + R

*********** RGB FINAL STAGE ***********                 ********** ALPHA FINAL STAGE **********
OUT.rgb = [V0.rgb]                                       OUT.a = (1-[Z0.a])
```

Compiler-Generated Vertex Code for Bowling-Pin Shader (NV_vertex_program code)

"constant" registers

```
c[0]-c[3]   = __projection * __modelview
c[4]-c[7]   = __modelview
c[8]        = __lightpos              [light position]
c[9]-c[11]  = affine(__modelview)
c[12]-c[14] = transpose(invert(affine(__modelview)))
c[15]       = color                   [light color]
c[16].x     = (__lightpos[3] == 0.0)  [is the light directional?]
c[16].y     = aq                      [light quadratic attenuation factor]
c[16].z     = al                      [light linear attenuation factor]
c[16].w     = ac                      [light constant attenuation factor]
c[17]       = {0.0961539 0 0.25 -0.5}
c[18]       = {0 0.192308 0.538462 1}
c[19]       = {0 0.0666667 0.5 3}
c[20].x     = 10
```

vertex-source registers

```
v[0]: __position
v[1]: __tangent
v[2]: __binormal
v[3]: __normal
v[4]: uv
```

```
!!VP1.0
DP4  o[HPOS].x, c[0], v[0] ;
DP4  o[HPOS].y, c[1], v[0] ;
DP4  o[HPOS].z, c[2], v[0] ;
DP4  o[HPOS].w, c[3], v[0] ;
DP4  R6.x, c[4], v[0] ;
DP4  R6.y, c[5], v[0] ;
DP4  R6.z, c[6], v[0] ;
DP4  R6.w, c[7], v[0] ;
MOV  R2, R6 ;
RCP  R6.x, R6.w ;
MUL  R7, R2, R6.x ;
ADD  R2, c[8], -R7 ;
MAD  R2, c[16].x, -R2, R2 ;
MOV  R6, c[8] ;
MAD  R2, c[16].x, R6, R2 ;
DP3  R6.x, R2, R2 ;
RSQ  R6.x, R6.x ;
MUL  R6, R2, R6.x ;
DP3  R5.x, c[9], v[1] ;
DP3  R5.y, c[10], v[1] ;
DP3  R5.z, c[11], v[1] ;
DP3  R8.x, R5, R5 ;
RSQ  R8.x, R8.x ;
MUL  R5, R5, R8.x ;
DP3  R1.x, R6, R5 ;
DP3  R4.x, c[9], v[2] ;
DP3  R4.y, c[10], v[2] ;
DP3  R4.z, c[11], v[2] ;
DP3  R8.x, R4, R4 ;
RSQ  R8.x, R8.x ;
MUL  R4, R4, R8.x ;
DP3  R1.y, R6, R4 ;
DP3  R3.x, c[12], v[3] ;
DP3  R3.y, c[13], v[3] ;
DP3  R3.z, c[14], v[3] ;
DP3  R8.x, R3, R3 ;
RSQ  R8.x, R8.x ;
MUL  R3, R3, R8.x ;
DP3  R1.z, R6, R3 ;
DP3  R8.x, R1, R1 ;
RSQ  R8.x, R8.x ;
MAD  R1, R1, R8.x, c[18].wwwx ;
MUL  o[COL0].xyz, -c[17].w, R1 ;
DP3  R8.x, R2, R2 ;
RSQ  R1, R8.x ;
DST  R2, R8.x, R1 ;
DP3  R1.x, R2, c[16].wzyy ;
RCP  R1.x, R1.x ;
MUL  R1, c[15], R1.x ;
MOV  o[COL1].xyz, R1 ;
SGE  R1.x, v[0].z, c[17].y ;
MAD  R1.z, R1.x, -c[17].y, c[17].y ;
MAD  R1.z, R1.x, c[18].w, R1.z ;
SLT  R1.y, c[19].w, v[4].x ;
MAD  R1.x, R1.y, -c[18].w, c[18].w ;
MAD  R1.x, R1.y, c[17].y, R1.x ;
MUL  R8.y, R1.z, R1.x ;
SGE  R8.x, R8.y, c[18].w ;
SGE  R8.z, c[18].w, R8.y ;
MIN  R8.z, R8.x, R8.z ;
MUL  R1.xy, c[20].x, v[0].xyxx ;
MOV  R1.z, c[17].y ;
MOV  R1.w, c[18].w ;
DP4  R2.x, c[17].xyyz, R1 ;
DP4  R2.y, c[18].xyxz, R1 ;
DP4  R2.z, c[18].xxwx, R1 ;
DP4  R2.w, c[18].xxxw, R1 ;
ADD  R1.x, c[18].w, -R2.x ;
MOV  R1.yz, R2.yyzy ;
MOV  R1.w, c[18].w ;
MAD  R1, R8.z, -R1, R1 ;
MAD  o[TEX0], R8.z, R2, R1 ;
MOV  R2.x, v[4].x ;
MUL  R2.y, c[20].x, v[0].y ;
MOV  R2.z, c[17].y ;
MOV  R2.w, c[18].w ;
DP4  R1.x, c[17].zyyw, R2 ;
DP4  R1.y, c[19].xyxz, R2 ;
DP4  R1.z, c[18].xxwx, R2 ;
DP4  R1.w, c[18].xxxw, R2 ;
MOV  o[TEX1], R1 ;
MOV  o[TEX2], R1 ;
DP3  R2.x, -R7, -R7 ;
RSQ  R2.x, R2.x ;
MAD  R1, -R7, R2.x, R6 ;
DP3  R2.x, R1, R1 ;
RSQ  R2.x, R2.x ;
MUL  R1, R1, R2.x ;
DP3  R0.x, R1, R5 ;
DP3  R0.y, R1, R4 ;
DP3  R0.z, R1, R3 ;
DP3  R1.x, R0, R0 ;
RSQ  R1.x, R1.x ;
MUL  o[TEX3].xyz, R0, R1.x ;
END
```

Compiler-Generated Primitive-Group Code for Bowling-Pin Shader (x86 CPU code)

```
push    ebp
mov     ebp, esp
sub     esp, 0x0000000c
push    esi
push    edi
push    ebx
fnstcw  [ebp-4h]
fnclex
mov     edi, [ebp+Ch]
mov     ebx, [ebp+8h]
mov     ebx, [ebx+50h]
mov     eax, [ebx]
mov     [edi], eax
mov     eax, [ebx+4h]
mov     [edi+4h], eax
mov     eax, [ebx+8h]
mov     [edi+8h], eax
mov     eax, [ebx+Ch]
mov     [edi+Ch], eax
mov     eax, [ebp+8h]
mov     eax, [eax+38h]
mov     eax, [eax]
mov     [edi+10h], eax
mov     ebx, [ebp+8h]
mov     ebx, [ebx+30h]
mov     eax, [ebx]
mov     [edi+14h], eax
mov     eax, [ebx+4h]
mov     [edi+18h], eax
mov     eax, [ebx+8h]
mov     [edi+1Ch], eax
mov     eax, [ebx+Ch]
mov     [edi+20h], eax
mov     ebx, [ebp+8h]
mov     ebx, [ebx+28h]
mov     eax, [ebx]
mov     [edi+24h], eax
mov     eax, [ebx+4h]
mov     [edi+28h], eax
mov     eax, [ebx+8h]
mov     [edi+2Ch], eax
mov     eax, [ebx+Ch]
mov     [edi+30h], eax
mov     eax, [ebx+10h]
mov     [edi+34h], eax
mov     eax, [ebx+14h]
mov     [edi+38h], eax
mov     eax, [ebx+18h]
mov     [edi+3Ch], eax
mov     eax, [ebx+1Ch]
mov     [edi+40h], eax
mov     eax, [ebx+20h]
mov     [edi+44h], eax
mov     eax, [ebx+24h]
mov     [edi+48h], eax
mov     eax, [ebx+28h]
mov     [edi+4Ch], eax
mov     eax, [ebx+2Ch]
mov     [edi+50h], eax
mov     eax, [ebx+30h]
mov     [edi+54h], eax
mov     eax, [ebx+34h]
mov     [edi+58h], eax
mov     eax, [ebx+38h]
mov     [edi+5Ch], eax
mov     eax, [ebx+3Ch]
mov     [edi+60h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+40h]
mov     eax, [eax]
mov     [edi+64h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+48h]
mov     eax, [eax]
mov     [edi+68h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+68h]
mov     eax, [eax]
mov     [edi+6Ch], eax
mov     eax, [ebp+8h]
mov     eax, [eax+70h]
mov     eax, [eax]
mov     [edi+70h], eax
mov     eax, [ebp+8h]
mov     eax, [eax+60h]
mov     eax, [eax]
mov     [edi+74h], eax
lea     eax, [edi+24h]
push    eax
lea     eax, [edi+78h]
push    eax
mov     [ebp-Ch], 0x0040105a
call    [ebp-Ch]
add     esp, 0x00000008
lea     eax, [edi+78h]
push    eax
lea     eax, [edi+9Ch]
push    eax
mov     [ebp-Ch], 0x00401672
call    [ebp-Ch]
add     esp, 0x00000008
lea     eax, [edi+9Ch]
push    eax
lea     eax, [edi+C0h]
push    eax
mov     [ebp-Ch], 0x0040120d
call    [ebp-Ch]
add     esp, 0x00000008
mov     ebx, [ebp+8h]
mov     ebx, [ebx]
mov     eax, [ebx]
mov     [edi+E4h], eax
mov     eax, [ebx+4h]
mov     [edi+E8h], eax
mov     eax, [ebx+8h]
mov     [edi+ECh], eax
mov     eax, [ebx+Ch]
mov     [edi+F0h], eax
mov     eax, [ebx+10h]
mov     [edi+F4h], eax
mov     eax, [ebx+14h]
mov     [edi+F8h], eax
mov     eax, [ebx+18h]
mov     [edi+FCh], eax
mov     eax, [ebx+1Ch]
mov     [edi+100h], eax
mov     eax, [ebx+20h]
mov     [edi+104h], eax
mov     eax, [ebx+24h]
mov     [edi+108h], eax
mov     eax, [ebx+28h]
mov     [edi+10Ch], eax
mov     eax, [ebx+2Ch]
mov     [edi+110h], eax
mov     eax, [ebx+30h]
mov     [edi+114h], eax
mov     eax, [ebx+34h]
mov     [edi+118h], eax
mov     eax, [ebx+38h]
mov     [edi+11Ch], eax
mov     eax, [ebx+3Ch]
mov     [edi+120h], eax
mov     eax, [edi+20h]
mov     [edi+124h], eax
mov     [edi+128h], 0x00000000
fld     [edi+124h]
fcomp   [edi+128h]
fnstsw  eax
test    eax, 0x00004000
mov     eax, 0x3f800000
jnz     l_0
xor     eax, eax
l_0:
mov     [edi+12Ch], eax
mov     eax, [edi+14h]
mov     [edi+130h], eax
mov     eax, [edi+18h]
mov     [edi+134h], eax
mov     eax, [edi+1Ch]
mov     [edi+138h], eax
lea     eax, [edi+24h]
push    eax
lea     eax, [edi+E4h]
push    eax
lea     eax, [edi+13Ch]
push    eax
mov     [ebp-Ch], 0x004014ba
call    [ebp-Ch]
add     esp, 0x0000000c
mov     ebx, [ebp+10h]
mov     eax, [edi]
mov     [ebx], eax
mov     eax, [edi+4h]
mov     [ebx+4h], eax
mov     eax, [edi+8h]
mov     [ebx+8h], eax
mov     eax, [edi+Ch]
mov     [ebx+Ch], eax
mov     eax, [edi+10h]
mov     [ebx+10h], eax
mov     eax, [edi+24h]
mov     [ebx+14h], eax
mov     eax, [edi+28h]
mov     [ebx+18h], eax
mov     eax, [edi+2Ch]
mov     [ebx+1Ch], eax
mov     eax, [edi+30h]
mov     [ebx+20h], eax
mov     eax, [edi+34h]
mov     [ebx+24h], eax
mov     eax, [edi+38h]
mov     [ebx+28h], eax
mov     eax, [edi+3Ch]
mov     [ebx+2Ch], eax
mov     eax, [edi+40h]
mov     [ebx+30h], eax
mov     eax, [edi+44h]
mov     [ebx+34h], eax
```

```
mov     eax, [edi+48h]
mov     [ebx+38h], eax
mov     eax, [edi+4Ch]
mov     [ebx+3Ch], eax
mov     eax, [edi+50h]
mov     [ebx+40h], eax
mov     eax, [edi+54h]
mov     [ebx+44h], eax
mov     eax, [edi+58h]
mov     [ebx+48h], eax
mov     eax, [edi+5Ch]
mov     [ebx+4Ch], eax
mov     eax, [edi+60h]
mov     [ebx+50h], eax
mov     eax, [edi+64h]
mov     [ebx+54h], eax
mov     eax, [edi+68h]
mov     [ebx+58h], eax
mov     eax, [edi+6Ch]
mov     [ebx+5Ch], eax
mov     eax, [edi+70h]
mov     [ebx+60h], eax
mov     eax, [edi+74h]
mov     [ebx+64h], eax
mov     eax, [edi+78h]
mov     [ebx+68h], eax
mov     eax, [edi+7Ch]
mov     [ebx+6Ch], eax
mov     eax, [edi+80h]
mov     [ebx+70h], eax
mov     eax, [edi+84h]
mov     [ebx+74h], eax
mov     eax, [edi+88h]
mov     [ebx+78h], eax
mov     eax, [edi+8Ch]
mov     [ebx+7Ch], eax
mov     eax, [edi+90h]
mov     [ebx+80h], eax
mov     eax, [edi+94h]
mov     [ebx+84h], eax
mov     eax, [edi+98h]
mov     [ebx+88h], eax
mov     eax, [edi+C0h]
mov     [ebx+8Ch], eax
mov     eax, [edi+C4h]
mov     [ebx+90h], eax
mov     eax, [edi+C8h]
mov     [ebx+94h], eax
mov     eax, [edi+CCh]
mov     [ebx+98h], eax
mov     eax, [edi+D0h]
mov     [ebx+9Ch], eax
mov     eax, [edi+D4h]
mov     [ebx+A0h], eax
mov     eax, [edi+D8h]
mov     [ebx+A4h], eax
mov     eax, [edi+DCh]
mov     [ebx+A8h], eax
mov     eax, [edi+E0h]
mov     [ebx+ACh], eax
mov     eax, [edi+12Ch]
mov     [ebx+B0h], eax
mov     eax, [edi+130h]
mov     [ebx+B4h], eax
mov     eax, [edi+134h]
mov     [ebx+B8h], eax
mov     eax, [edi+138h]
mov     [ebx+BCh], eax
mov     eax, [edi+13Ch]
mov     [ebx+C0h], eax
mov     eax, [edi+140h]
mov     [ebx+C4h], eax
mov     eax, [edi+144h]
mov     [ebx+C8h], eax
mov     eax, [edi+148h]
mov     [ebx+CCh], eax
mov     eax, [edi+14Ch]
mov     [ebx+D0h], eax
mov     eax, [edi+150h]
mov     [ebx+D4h], eax
mov     eax, [edi+154h]
mov     [ebx+D8h], eax
mov     eax, [edi+158h]
mov     [ebx+DCh], eax
mov     eax, [edi+15Ch]
mov     [ebx+E0h], eax
mov     eax, [edi+160h]
mov     [ebx+E4h], eax
mov     eax, [edi+164h]
mov     [ebx+E8h], eax
mov     eax, [edi+168h]
mov     [ebx+ECh], eax
mov     eax, [edi+16Ch]
mov     [ebx+F0h], eax
mov     eax, [edi+170h]
mov     [ebx+F4h], eax
mov     eax, [edi+174h]
mov     [ebx+F8h], eax
mov     eax, [edi+178h]
mov     [ebx+FCh], eax
fldcw   [ebp-4h]
pop     ebx
pop     edi
pop     esi
mov     esp, ebp
pop     ebp
ret
```

Shading System Immediate-Mode API v2.1

William R. Mark

April 18, 2001

1 Introduction

This document describes modifications to the OpenGL API to support the immediate-mode use of the Stanford real-time shading language. We collectively refer to these extensions as the shading-language immediate-mode API. These extensions are implemented as a layer on top of regular OpenGL.

The immediate-mode API supports the following major operations:

1. Loading the source code for a light shader or surface shader from a file.
2. Associating one or more light shader(s) with a surface shader to create a combined surface/light shader.
3. Compiling a combined surface/light shader for the current graphics hardware.
4. Selecting a compiled surface/light shader to use as the current shader for rendering.
5. Setting values of shader parameters.

When a shader is active, many OpenGL commands are no longer allowed, because their functionality is provided through the shading language. The disallowed commands fall into four major categories:

1. Fragment-processing commands (e.g. fog, texturing modes)
2. Texture-coordinate generation and transformation commands
3. Lighting commands.
4. Material-property commands.

When using our “lburg” multi-pass fragment backend, commands that configure framebuffer blending modes are also forbidden (instead, use the `Cprev` builtin variable within a shader). However, these commands are allowed with our “nv” fragment backend.

Using a forbidden OpenGL command while a programmable-shader is active will result in undefined behavior.

2 Initialization

`sglInit()`

The application must initialize the programmable shading system by calling `sglInit()` once, before calling any other `sgl*` routines.

3 Loading shader source code

`int sglShaderFile(GLuint shaderSourceID, char *shaderName, char *filename)`

Loads the source code for the shader named `shaderName` from file `filename`, and assigns it the identifier `shaderCodeID`. Any other shaders that are specified in the file are ignored. The loaded shader source code becomes the active shader source code. The specified shader may be either a light shader or a surface shader. `shaderCodeID` must be unused when this routine is called. The return code is 0 if there were no errors, 1 if there was an error.

4 Compiling and activating shaders

After the source code for a shader is loaded, but before it is used, the shader must be compiled. Our system treats shader source code and compiled shaders as largely separate entities.

`sglCompileShader(GLuint shaderID)`

Compiles the current shader source code. The compiled shader is assigned the user-specified identifier `shaderID`. If the shader is a surface shader, it incorporates any currently associated light shaders (discussed in the next section).

The newly created shader becomes the 'active' shader, as if sglBindShader() had been called. If the shader is a light shader, it is only active in the sense that subsequent `sglParameterHandle()` calls will apply to it. A light shader can only be activated for rendering purposes by associating it with a surface shader using `sglUseLight()`.

The current shader source code remains unchanged by this call.

Note that `shaderID` may not be -1, because this value is reserved for `SGL_STD_OPENGL`, the standard OpenGL lighting/shading model.

`sglBindShader(GLuint shaderID)`

Changes the currently active shader to that specified by `shaderID`. Note that it is illegal to render geometry when a light shader is bound.

Specifying `SGL_STD_OPENGL` reverts to the standard OpenGL lighting/shading model.

5 Associating lights with a surface

For efficiency reasons, the shading system must know which light shaders will be used with a surface shader before the surface shader is compiled.

```
sglUseLight(GLuint lightShaderID)
```

This command binds a "compiled" light shader to the current surface-shader source code. `lightShaderID` indicates the light that is to be associated with the surface.

More than one light can be associated with a surface, by calling `sglUseLight()` multiple times.

However, the same (compiled) light shader may not be used more than once with a single surface. If two identical lights are required, compile the light shader twice. Our system imposes this requirement because the *lightShaderID* is used to specify how light parameters are modified. "Identical" lights will usually have different parameter values (e.g. position).

5.1 Setting parameter values

For performance reasons, shader parameters are identified at rendering-time with numeric identifiers rather than names. For each compiled shader, the programmer can choose the bindings from names to numeric identifiers, within some constraints. We refer to the numeric parameter identifiers as *parameter handles*. Each compiled surface or light shader has its own parameter-handle space.

There is an important advantage to allowing the programmer to choose values of parameter handles. It facilitates the use of a single geometry rendering routine (e.g. renderSphere) with different surface shaders, as long as the programmer chooses a consistent mapping of parameter handles to actual parameters for all of the relevant shaders.

```
sglParameterHandle(char *paramName, GLuint paramHandle)
```

Assigns the parameter handle `paramHandle` to the current shader's parameter `paramName`. The value of paramHandle must be between 0 and `SGL_MAX_PARAMHANDLE`. The value of `SGL_MAX_PARAMHANDLE` is guaranteed to be no less than 15.

```
sglParameter*(GLuint paramHandle, TYPE v, ... )
sglParameter*v(GLuint paramHandle, TYPE *v)
```

Assigns a value to the shader parameter(s) specified by `paramHandle`. For a per-vertex parameter, this routine may be called at any time. For a per-primitive parameter, this routine may only be called outside of a begin/end pair.

Because our shading language does not explicitly identify shader parameters as "colors" or "texture coordinates", the shading system can not automatically assign default values in the manner that OpenGL does. For example, in OpenGL a `glColor3f` command automatically sets the fourth value (alpha) to the default

value of 1.0. When using our system, the user must always specify all four components of the color value. Likewise, the user must always specify all four components of a texture value. For a 2D texture, the third and fourth values should usually be set to 0.0 and 1.0 respectively.

The `sglParameter*` routine is available in `sglParameter1*`, `sglParameter4*`, and `sglParameter16*` variants. The `sglParameter16*` variants are used to specify matrix parameters, using OpenGL's array format.

If the shading language specifies a parameter's type as either `clampf` or `clampfv`, type conversions are performed in the same manner as they are for the OpenGL `glColor*` routines (see OpenGL Red Book, 3rd edition, Table 4-1).[1]. In summary, integer-to-float conversions are performed such that the maximum integer value (e.g. 255 for an unsigned byte) maps to 1.0. This behavior allows colors and normals to be stored in unsigned bytes in a natural manner.

Our shading language uses textures, but the contents of the textures are not defined using the language. Textures are defined by the application program, then passed to the shading-language routine as a 'texref' parameter. Our system relies on OpenGL's texture object facility (`glBindTexture()`). The `sglParameter1i` or `sglParameter1iv` routines are used to specify 'texref' parameters. The value of the integer parameter is the *textureName* created using `glBindTexture()`.

```
sglLightParameter*(GLuint lightShaderID, GLuint paramHandle, TYPE v, ... )
sglLightParameter*v(GLuint lightShaderID, GLuint paramHandle, TYPE *v)
```

Assigns a value to the light parameter specified by `paramHandle`. The "compiled" light shader is specified by `lightShaderID`. For a per-vertex parameter, this routine may be called at any time. For a per-primitive parameter, this routine may only be called outside of a begin/end pair.

5.2 Light Pose

The pose of a light (position, direction, and orientation) is set using a routine defined for that purpose.

```
sglLightPosefv(GLuint lightShaderID, GLuint pname, GLfloat *v)
```

`pname` can be `SGL_POSITION`, `SGL_DIRECTION`, or `SGL_UPAXIS`. The light direction defines the $-Z$ axis in light space, and the up axis defines the Y axis in light space.

The vector `v` should always be a four-element vector, and is considered to be in modelview space (i.e. transformed by the modelview matrix or its inverse transpose, as appropriate). For `SGL_POSITION`, the fourth element of the vector should usually be set to 1.0. For `SGL_DIRECTION` and `SGL_UPAXIS`, the fourth element should usually be set to 0.0.

[1] For implementation simplicitly, our system deviates from the behavior in Table 4-1 in a minor way. Our system treats negative and positive values symmetrically. For example, a signed-byte value of -127 maps to -1.0, whereas in OpenGL the value of -128 maps to -1.0

5.3 Ambient Light

`sglAmbient*( ... )`

Specify the global ambient color. This color is accessible in surface shaders using the pre-defined `Ca` variable. If a surface shader does not use the `Ca` variable, the ambient color will be ignored. This routine can not be called inside a Begin/End pair.

6 New versions of standard OpenGL routines

6.1 Begin/End and Flush/Finish

Use `sglBegin()`, `sglEnd()`, `sglFlush()`, and `sglFinish()` instead of the corresponding standard OpenGL routines. Using the standard OpenGL routines while a programmable shader is active will result in undefined behavior.

6.2 Vertices, Normals, Tangents, Binormals

```
sglVertex*(TYPE v, ... )
sglVertex*v(TYPE v, ... )
sglNormal3*(TYPE v, ... )
sglNormal3*v(TYPE v, ... )
sglTangent3*(TYPE v, ... )
sglTangent3*v(TYPE v, ... )
sglBinormal3*(TYPE v, ... )
sglBinormal3*v(TYPE v, ... )
```

Vertices and local coordinate-frame vectors are passed using our versions of the classical OpenGL routines. The results of calling one of the standard OpenGL routines while a programmable shader is active are undefined.

7 Depth testing

Ideally, depth testing works exactly as it does in standard OpenGL. However, in some implementations, incorrect shading may occur if two (potentially visible) fragments at a pixel have exactly the same depth. This problem only occurs if an implementation uses the framebuffer for inter-pass temporary storage in a multi-pass shader.

8 Error Handling

The shading system has a flexible method for handling errors. Errors are divided into two classes, minor and major. For each class of error, the application can choose one of four behaviors:

- `SG_MSG_NONE` – No message is printed, and program execution continues. Errors can only be detected by polling for them using `sglGetError`.
- `SGL_MSG_WARN_ONCE` – A message is printed for the first error that occurs, and program execution continues. No message is printed for subsequent errors.
- `SGL_MSG_WARN` – A message is printed for every error that occurs, and program execution continues.
- `SGL_MSG_ABORT` – When an error occurs, a messsage is printed and program execution is halted.

`sglDebugLevel(int minor, int major)`

Specify the behavior for minor and major errors. The default is `sglDebugLevel(SGL_MSG_WARN_ONCE, SGL_MSG_ABORT)`.

`GLenum sglGetError(void)`

Poll for an error. If no error has occurred, `GL_NO_ERROR` is returned. If an error has occurred, the error code is returned.

`const GLubyte* sglErrorString(GLenum errorCode)`

Returns a descriptive string corresponding to an error code.

9 System Tips

In our current implementation, every `sglBegin`/`sglEnd` pair is very expensive. If possible, group all primitives into one such pair.

Because of restrictions in current graphics hardware, if a translucent shader is implemented using more than one hardware pass, overlapping transparent primitives will not render correctly. You must call `sglFlush` between each group of potentially overlapping primitives to avoid this problem.

Simple Program that uses Immediate-Mode Interface

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <GL/glut.h>

#include "imode.h"

/*
 * Macro to check OpenGL error status, and print message if so
 */
#define check_gl_error do {GLenum glerr; \
    while ((glerr = glGetError()) != GL_NO_ERROR) \
      fprintf(stderr, "OpenGL error '%s' at %s:%i\n", gluErrorString(glerr), \
                      __FILE__, __LINE__);} while(0)

/* The following parameter handles can be chosen mostly arbitrarily
   (must be smallish, unique numbers) */
#define PH_COLOR      1
#define PH_AC         2
#define PH_AL         3
#define PH_AQ         4
#define PH_TEX        7
#define PH_UV         8
#define PH_SURFCOLOR  9

/* glBindTexture ID */
#define TEXID 1

GLfloat light_diffuse[] = {1.0, 0.0, 0.0, 1.0};

/*
 * Checkboard texture
 */
#define checkWidth  64
#define checkHeight 64
static GLubyte checkImage[checkWidth][checkHeight][4];

void makeCheckImage(void) {
  int i,j,c;
  for (i=0; i<checkHeight; i++) {
    for (j=0; j<checkWidth; j++) {
      c = (((i&0x8)==0)^((j&0x8)==0))*255;
      checkImage[i][j][0] = (GLubyte) c;
      checkImage[i][j][1] = (GLubyte) c;
      checkImage[i][j][2] = (GLubyte) c;
      checkImage[i][j][3] = (GLubyte) 255;
    }
  }
}

/*
 * Sets up checkboard as texture #TEXID
 */
void setupTexture() {
  makeCheckImage();
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glBindTexture(GL_TEXTURE_2D, TEXID);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  GL_NEAREST);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkWidth, checkHeight,
               0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
}

/*
 * draw cube with clockwise verts when looking at cube from outside
 */
void drawcube(void) {
  GLfloat red[]   = {1.0, 0.0, 0.0, 1.0};
  GLfloat green[] = {0.0, 1.0, 0.0, 1.0};
  GLfloat blue[]  = {0.0, 0.0, 1.0, 1.0};
  float UVa[] = {0.0, 0.0, 0.0, 1.0};
  float UVb[] = {0.0, 1.0, 0.0, 1.0};
  float UVc[] = {1.0, 1.0, 0.0, 1.0};
  float UVd[] = {1.0, 0.0, 0.0, 1.0};

  sglBegin(GL_QUADS);
  sglParameter4fv(PH_SURFCOLOR, red);
  /* x=1 face */
  sglNormal3f( 1.0,  0.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f( 1.0, -1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f( 1.0, -1.0,  1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0,  1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f( 1.0,  1.0, -1.0);
  /* x=-1 face */
  sglNormal3f(-1.0,  0.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0, -1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f(-1.0,  1.0, -1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f(-1.0,  1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f(-1.0, -1.0,  1.0);
  /* y=1 face */
  sglParameter4fv(PH_SURFCOLOR, green);
  sglNormal3f( 0.0, 1.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0, 1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f( 1.0, 1.0, -1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0, 1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f(-1.0, 1.0,  1.0);
  /* y=-1 face */
  sglNormal3f( 0.0,-1.0,  0.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0,-1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f(-1.0,-1.0,  1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0,-1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f( 1.0,-1.0, -1.0);
  /* z=1 face */
  sglParameter4fv(PH_SURFCOLOR, blue);
  sglNormal3f( 0.0, 0.0,  1.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0,-1.0,  1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f(-1.0, 1.0,  1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0, 1.0,  1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f( 1.0,-1.0,  1.0);
  /* z=-1 face */
  sglNormal3f( 0.0, 0.0, -1.0);
  sglParameter4fv(PH_UV, UVa);   sglVertex3f(-1.0,-1.0, -1.0);
  sglParameter4fv(PH_UV, UVb);   sglVertex3f( 1.0,-1.0, -1.0);
  sglParameter4fv(PH_UV, UVc);   sglVertex3f( 1.0, 1.0, -1.0);
  sglParameter4fv(PH_UV, UVd);   sglVertex3f(-1.0, 1.0, -1.0);
  sglEnd();
}
```

continued on next page

```
static void init_shader_params(void) {
  static float light_ambient[4] = { 0.2, 0.2, 0.2, 1.0 };
  static int frame = 0;

  /*
   * Changes to modelview matrix
   */
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 5,    /* Eye */
                    0.0, 0.0, 0.0,  /* Center */
                    0.0, 1.0, 0.0); /* Up */
  glTranslatef(0.0, 0.0, -1.0);
  glRotatef((float)frame++, 0.0, 1.0, 0.0);
  glRotatef(35.264, 1.0, 0.0, 0.0);
  glRotatef(45, 0.0, 0.0, 1.0);

  sglAmbient4fv(light_ambient);
}

/* GLUT keyboard callback -- Quit when 'Q' key is pressed */
void keyboard(unsigned char key, int x, int y) {
  switch (key) {
  case 'q': case 'Q': /* quit */
    exit(0);
    break;
  }
}

/* GLUT reshape callback -- reset viewport when window size changes */
void reshape(GLint w, GLint h) {
  sglViewport(0, 0, w, h);
}

/* GLUT idle callback -- continuously redraw so that we get animation */
void dynamicIdle(void) {
  glutPostRedisplay();
}

/* GLUT display callback -- draw the scene */
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  sglBindShader(200);
  init_shader_params();
  drawcube();
  glutSwapBuffers();
  check_gl_error;
}

void gfxinit(void) {
  float light_color[4] = { 1.0, 1.0, 1.0, 1.0 };
  float atten_constant = 1.0;
  float atten_linear = 0.01;
  float atten_quadratic = 0.0;
  GLfloat light_position[] = {3.0, 3.0, 3.0, 1.0};

  glEnable(GL_DEPTH_TEST);
  glMatrixMode(GL_PROJECTION);
  gluPerspective( /* FOV in deg */ 40.0, /* Aspect ratio */ 1.0,
                  /* Znear */ 1.0, /* Zfar */ 10.0);
  glMatrixMode(GL_MODELVIEW);
  gluLookAt(0.0, 0.0, 5,    /* Eye */
                    0.0, 0.0, 0.0,  /* Center */
                    0.0, 1.0, 0.0); /* Up */
  glTranslatef(0.0, 0.0, -1.0);

  /*
   * Shading system setup
   */
#if 0
  /* Specify specific codegens; without this defaults are used */
  set_bcodegen("x86");
  set_vcodegen("nv20");
  set_fcodegen("nv");
#endif
  sglInit();


  /*
   * Load and compile light shader
   */
  sglShaderFile(99, "simple_light", "../simpshade.in");
  sglCompileShader(299);
  sglParameterHandle("color",       PH_COLOR);
  sglParameterHandle("ac",          PH_AC);
  sglParameterHandle("al",          PH_AL);
  sglParameterHandle("aq",          PH_AQ);
  /*
   * Specify light position & configuration
   */
  sglLightPosefv(299,               SGL_POSITION,       light_position);
  sglLightParameter4fv(299,         PH_COLOR,           light_color);
  sglLightParameter1f(299,          PH_AC,              atten_constant);
  sglLightParameter1f(299,          PH_AL,              atten_linear);
  sglLightParameter1f(299,          PH_AQ,              atten_quadratic);

  /*
   * Load and compile surface shader
   */
  sglShaderFile(1, "simple_surface", "../simpshade.in");
  sglUseLight(299);
  sglCompileShader(200);
  sglParameterHandle("surfcolor",   PH_SURFCOLOR);
  sglParameterHandle("tex",         PH_TEX);
  sglParameterHandle("uv",          PH_UV);
}

int main(int argc, char **argv) {
  /*
   * GLUT setup
   */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutCreateWindow("simple");
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutIdleFunc(dynamicIdle);
  glutKeyboardFunc(keyboard);

  /*
   * Initialize graphics
   */
  gfxinit();
  setupTexture();
  sglParameter1i(PH_TEX, TEXID);

  /*
   * Start event loop
   */
  glutMainLoop();
  return 0;
}
```

Stanford Real-Time Shading Language

Version 5
Kekoa Proudfoot
October 26, 2000

Basics

The general format of our language, as well as our language's declaration and expression syntax, is similar to C. Our language does, however, have a number of notable differences. These include a different set of data types, a number of specialized type modifiers, a slightly different set of operators, and different semantics with regards to function calls and global variables. These differences will become clearer as you proceed through this document.

As with C, our language relies on white space and indenting only to the extent that they separate tokens in the language. White space and indenting are otherwise ignored.

Comments are allowed in our language. These may be denoted using either the C /* */ syntax or the C++ // comment syntax. Identifiers, integers, and floats are all specified as they are in C. Identifiers are case-sensitive.

Base data types

We begin the discussion of our language with a description of its data types.

In our language, data types are composed of a base data type preceeded by an optional list of type modifiers. In this section, we describe the base data types. We leave the discussion of type modifiers for later sections.

Our language supports ten base data types. They are:

`bool`	boolean value
`clampf1`	scalar [0,1]-clamped floating-point value
`clampf3`	3-component [0,1]-clamped floating-point vector
`clampf4`	4-component [0,1]-clamped floating-point vector
`float1`	scalar unclamped floating-point value
`float3`	3-component unclamped floating-point vector
`float4`	4-component unclamped floating-point vector
`matrix3`	3x3 floating point matrix
`matrix4`	4x4 floating point matrix
`texref`	texture reference

Two of these types need further explanation.

The `bool` type is either true or false. It has no numerical value.

The `texref` type stores a reference to a texture. Its value corresponds to an OpenGL texture name as specified to `glBindTexture`.

Additionally, note that although the clamped float types are described as floating point, because their ranges are limited to [0,1], they may be implemented using either fixed- or floating-point.

In addition to the ten base types, we support some additional type names for compatibility with the previous version of the language:

`clampf`	same as clampf1
`clampfv`	same as clampf4
`float`	same as float1
`floatv`	same as float4
`matrix`	same as matrix4

Expressions, operators, and builtin functions

The expression syntax of our language is much like that of C, except that we provide a different set of operators and also a core set of builtin functions. In this section, we introduce and describe these operators and functions.

Most operators that we provide have both `float` and clampf versions, where the `clampf` versions are defined to clamp their results (but not their intermediate values) to [0,1]. We make special note of operators which either do not have `clampf` versions or do not operate on float or `clampf` values at all.

We begin with operators for manipulating scalars and vectors.

The join operator `{}` assembles scalars into vectors and vectors into matrices. It comes in five versions:

`{ x, y, z }`	make a 3-vector from scalars x, y, and z
`{ x, y, z, w }`	make a 4-vector from scalars x, y, z, and w
`{ xyz, w }`	make a 4-vector from 3-vector xyz and scalar w
`{ r0, r1, r2 }`	make a 3x3 matrix from 3-vector rows r0, r1, r2
`{ r0, r1, r2, r3 }`	make a 4x4 matrix from 4-vector rows r0, r1, r2, r3

The index operator `[]` extracts a scalar from a 3- or 4-component vector. Indexing is zero-based:

```
{ x, y, z }[0]      // extract x
{ x, y, z }[2]      // extract z
{ x, y, z, w }[3]   // extract w
```

The index operator `[]` can also extract a row from a 3x3 matrix or a 4x4 matrix:

```
{ r0, r1, r2 }[0]      // extract r0 from 3x3 matrix { r0, r1, r2 }
```

```
{ r0, r1, r2 }[2]      // extract r2 from 3x3 matrix { r0, r1, r2 }
{ r0, r1, r2, r3 }[3]  // extract r3 from 4x4 matrix { r0, r1, r2, r3 }
```

The `rgb()`, `alpha()`, and `blue()` operators help make compilation to fragment pipelines efficient. Their various forms are shown here:

```
rgb({ r, g, b, a })     // extract 3-vector { r, g, b } from 4-vector
alpha({ r, g, b, a })   // extract scalar a from 3-vector
blue({ r, g, b, a })    // extract scalar b from 3-vector
blue({ r, g, b })       // extract scalar b from 3-vector
rgb(c)                  // construct 3-vector { c, c, c } from scalar c
```

We provide scalar and vector versions of add, multiply, subtract, and divide. For multiply and divide, we also provide versions that operate on one scalar and one vector in either order. Some examples:

```
a + b
a - b
a * b
a / b
{ ax, ay, az } + { bx, by, bz }
{ ax, ay, az } - { bx, by, bz }
{ ax, ay, az } * { bx, by, bz }
{ ax, ay, az } / { bx, by, bz }
a * { bx, by, bz }
a / { bx, by, bz }
{ ax, ay, az } * b
{ ax, ay, az } / b
```
etc.

Multiplication of two matrices and multiplication of one matrix (on the left) and one vector (on the right) are also supported. Since we do not support clamped matrices, there are no clampf matrix-matrix or matrix-vector multiply operations.

We provide an unclamped floating-point negate operator:

```
- a
```

We do not provide a `clampf` version of the negate operator, since its result would always be zero.

We provide a generic blend operator that operates on clamped and unclamped 4-vectors only. The blend operator is based on the OpenGL blend function and takes the following form:

```
blend ( src_factor, dst_factor )
```

Note this the blend operator is a binary infix operator. The value to the left of the blend is called the source (src) and the value to the right of the blend is called the destination (dst):

```
src blend(src_factor,dst_factor) dst
```

Such an expression computes:

src_factor * src + dst_factor * dst

Both src_factor and dst_factor are placeholders for names chosen from the following list. Each has the value indicated:

Factor Name	**Factor Value**
ZERO	{ 0, 0, 0, 0 }
ONE	{ 1, 1, 1, 1 }
SRC_COLOR	src
SRC_ALPHA	{ src[3], src[3], src[3], src[3] }
DST_COLOR	dst
DST_ALPHA	{ dst[3], dst[3], dst[3], dst[3] }
ONE_MINUS_SRC_COLOR	{ 1, 1, 1, 1 } - src
ONE_MINUS_SRC_ALPHA	{ 1, 1, 1, 1 } - { src[3], src[3], src[3], src[3] }
ONE_MINUS_DST_COLOR	{ 1, 1, 1, 1 } - dst
ONE_MINUS_DST_ALPHA	{ 1, 1, 1, 1 } - { dst[3], dst[3], dst[3], dst[3] }

We provide two additional blend operators to simplify the specification of common blend operations. The `over` operator composites two values with premultiplied alpha, and is equivalent to `blend(ONE,ONE_MINUS_SRC_ALPHA)`. The `blend_over` operator composites two values where only second value has premultiplied alpha. The first value has non-premultiplied alpha. It is equivalent to `blend(SRC_ALPHA,ONE_MINUS_SRC_ALPHA)`.

We provide a standard set of comparison operators (==, !=, >, <, >=, <=) for computing boolean values.

Boolean expressions are used with the conditional `select` operator. The `select` operator takes three parameters: a boolean, a value to return if the boolean is true, and a value to return if the boolean is false. Some examples:

```
select(0 == 0, t, f)        // value is t
select(0 > 1, t, f)         // value is f
```

We provide a number of additional operations, including: scalar and vector `clamp`, `min`, and `max` operations; vector `dot`, `length`, and `normalize` operations; a 3-vector `reflect` and `cross` operations; `sin`, `cos`, `pow`, and `sqrt`. Some examples:

```
clamp(0.5, 0, 1)                                  // value is 0.5
clamp({ -1, 0, 1, 2 }, 0, 1)                      // value is { 0, 0, 1, 1 }
clamp({ -1, 1, 3 }, { 0, 0, 1 }, { 1, 2, 2})      // value is { 0, 1, 2 }
min({ -1, 1, 2, 3 }, { 1, 0, 1, 4 })              // value is { -1, 0, 1, 4 }
dot({ 0, 1, 2, 3 }, { 4, 5, 6, 7 })               // value is 38
length({ 3, 4, 0 })                               // value is 5
length({ 1, 1, 1 })                               // value is 1.7320...
length({ 1, 1, 1, 1 })                            // value is 2
normalize({ 0, 0, 2 })                            // value is { 0, 0, 1 }
reflect({ 1, 1, 1 }, { 0, 0, 1 })                 // value is { -1, -1, 1 }
reflect({ 1, 0, 0 }, { 0, 1, 0 })                 // value is { 0, 0, 1 }
sin(3.14159)                                      // value is 0
cos(3.14159)                                      // value is -1
```

```
pow(10,2)                                   // value is 100
sqrt(2)                                     // value is 1.4142...
```

We also provide a number of matrix operations:

`affine`	extracts the upper-left 3x3 matrix from a 4x4 matrix
`frustum`	generates a 4x4 frustum projection matrix
`identity`	generates a 4x4 identity matrix
`invert`	inverts a 3x3 or a 4x4 matrix
`lookat`	generates a 4x4 lookat matrix
`ortho`	generates a 4x4 orthographic projection matrix
`rotate`	generates a 4x4 rotation matrix of an angle about an axis
`scale`	generates a 4x4 scale matrix
`translate`	generates a 4x4 translation matrix
`transpose`	transposes a 3x3 or 4x4 matrix
`identity3`	generates a 3x3 identity matrix
`rotate3`	generates a 3x3 rotation matrix
`scale3`	generates a 3x3 scale matrix

The exact parameters needed for each matrix operation are discussed in the operator appendix.

A number of texturing and lookup operations are also available:

`cubemap`	perform a cubemap lookup given a texref and a 3-vector
`cubenorm`	perform a 3-vector normalization given a 3-vector
`lut`	perform a component-wise fragment `clamp4` table lookup
`texture`	perform a 2d texture lookup given a texref and a 3- or 4-vector
`texture3d`	perform a 3d texture lookup given a texref and a 3- or 4-vector
`bumpdiff`	perform a diffuse bumpmap operation
`bumpspec`	perform a specular bumpmap operation (requires bumpdiff)

The exact parameters needed for each texture/lookup operation are discussed in the operator appendix.

The `lut` operator performs a component-wise table lookup of fragment value. It uses the OpenGL color lookup table defined using `glPixelMap`. Our intent is to eventually abstract lookup table specification to allow multiple lookup tables, but currently we only support one color lookup table at a time.

The `bumpdiff` and `bumpspec` operators implement bumpmapping as described for NVIDIA hardware by Mark Kilgard. The `bumpdiff` operator computes the diffuse reflection coefficient given a tangent-space normal map, texture coordinates, and a tangent-space light vector. The `bumpspec` operator computes the specular reflection coefficient given the same normal map and texture coordinates plus the tangent-space half-angle vector. The `bumpdiff` operator leaves a self-shadowing term in alpha which must be used to modulate the `bumpspec` result. The blend operator, configured as `blend(ONE,SRC_ALPHA)`, is used to accomplish this.

As with C, we support parentheses () for grouping expressions to override the default operator precedences.

Two special operators are the assignment and cast operators. Both are used as they typically are in C. Assignment implies a cast to the type of the value being set. Type conversion is discussed in greater detail in a later section describing type conversion.

An important note about the assignment operator. We currently do not support assignment to an indexed vector element:

```
v[3] = 0;                       // forbidden
```

Use something like this instead:

```
v = { v[0], v[1], v[2], 0 };   // use something like this instead
```

Finally, we mention the `integrate()` operator, which we discuss in more detail in a later section on surface and light shaders.

Operator precedence

We define the following binary operator precedences, by group from lowest precedence to highest precedence:

```
=
== !=
> < >= <=
+ -
blend over blend_over
* /
```

All of the binary operators are left associative, except for =, which is right associative.

Statements

Our language supports three kinds of statements: variable declarations, expression statements, return statements. Empty statements are permitted; these are ignored.

A variable declaration is similar to C, and consists of a type followed by an identifier followed by an optional initializer followed by a semicolon.

```
float1 f1;                      // declare f1
float1 f2 = 1;                  // declare and initialize f2
float4 v1 = { 1, 2, 3, 4 };     // declare and initialize v1
float4 v2 = f1 * v1;            // declare and initialize v2
```

As with C++, variables may be declared anywhere in a basic block.

Expression statements are simply an expression followed by a semicolon:

```
    1;                  // valid but useless, eventually optimized away
    N = normalize(N);   // normalize N
    NdotL = dot(N,L);   // compute dot product of N and L
```

A return statement is used to indicate the final value of a shader or function:

```
    return color;
```

Functions

Our language allows functions to be defined and called mostly like they are in C, with a few exceptions. First, there is no such thing as a `void' function, and therefore all functions must return a value. Second, there is (currently) no such thing as a function declaration for user-defined functions. All user-defined functions must be defined before they may be used. Finally, recursion is forbidden.

All of these differences are due to the way function calls are implemented. All function calls are inlined.

Here are some examples:

```
float4 lerp (float4 a, float4 b, float afrac)
{
    return afrac * a + (1 - afrac) * b
}

float4 bilerp (float4 v00, float4 v01, float4 v10 float4 v11,
               float frac0, float frac1)
{
    float4 v0 = lerp(v00, v01, frac0);
    float4 v1 = lerp(v10, v11, frac0);
    return lerp(v0, v1, frac1);
}
```

Surface shaders, light shaders, and the integrate() operator

Our language borrows the RenderMan concept of separate surface and light shaders to provide orthogonality between these shading operations. Light shaders compute how much light is incident on a surface, while surface shaders compute the amount of light reflected toward the viewer, possibly querying lights to determing and account for the amount of light arriving from each light source.

Surface and light shaders are written as functions are, except that their return types are preceeded by the `shader' modifier plus also either the `surface' or the `light' modifier. In addition, shaders must return a float4 or a clamp4 type:

```
float func () { return ...; }                    // an ordinary function
surface shader float4 surf () { return ...; }    // a surface shader
light shader float4 light () { return ...; }     // a light shader
```

The surface and light modifiers may also be applied to functions. When this is done, such a function may access special features (variables and such) available only to surface and light shaders. In addition, the function becomes accessible only to other surface or light functions and shaders, as appropriate. More examples:

```
surface float surffunc () { return ...; }    // a surface function
surface float lightfunc () { return ...; }   // a light function
```

To query light sources, surface shaders (and functions) use the `integrate()` operator. This operator takes an expression and loops over all active light sources, evaluating the expression once per light source. The operator returns the sum of the expression evaluations.

The `integrate()` operator evaluates special `per-light' expressions, which are expressions that depend directly on special built-in per-light values (in particular the light vector, the half-angle vector, and the light intensity) and/or other per-light expressions. In evaluating a per-light expression once per light, the `integrate()` operator removes the per-light attribute of the integrated expression.

We use a type modifier scheme to track per-light expressions. Just as every value in our system has a type, every value also has a type modifier that specifies whether or not the value changes with every light. In our system, the keyword `perlight' is used to indicate such a value. We require all variables and return values that hold per-light values to be declared with the `perlight` modifier. We impose this requirement to make user code more readable. Our compiler separately infers which values are perlight, and it uses this information to report an error when a perlight value is stored to a non-perlight variable.

Here are some examples of perlight values and the `integrate()` operator. Assume `L`, `H`, and `Cl` are per-light values:

```
float4 Kd = ...;                            // compute diffuse surface color
perlight float NdotL = max(dot(N,L),0);     // max(dot(N,L),0) is perlight
perlight float intensity = Cl * NdotL;      // Cl * NdotL is perlight
float color = Kd * integrate(intensity);    // integrate light and modulate

perlight float NdotH = dot(N,H);    // dot(N,H) is perlight
float NdotH = dot(N,H);             // error: missing perlight modifier
```

As we will see in a later section on builtin global values, `Cl` in particular references the amount of light incident on the surface from each light. By referencing `Cl`, surface shaders indirectly reference the active light shaders.

Values that have been integrated once cannot be integrated again. This is something of an artificial restriction that was imposed because it really doesn't make a lot of sense to integrate a value that has already been integrated.

Computation frequencies and computation frequency type modifiers

A key aspect of our system is its support for computations at a variety of different rates, or computation frequencies. We support four different computation frequencies: once at compile time, once per group of primitives, once per vertex, and once per fragment. In our system every shading computation occurs at one of the rates.

Note that we do not provide a frequency that corresponds to once per primitive. Ideally we would support such a frequency, in particular for flat shading, but do not because OpenGL only provides limited support for that computation frequency. Specifically, OpenGL does not provide support for per-primitive texture coordinates.

As with our treatment of per-light expressions, we use a type modifier system to control the frequencies at which computations occur. This modifier specifies how often that value is computed (or specified, if the value is a parameter).

There is one type modifier for each computation frequency. The modifiers are: `` `constant' ``, `` `vertex' ``, `` `primitive group' ``, and `` `fragment' ``. We provide an additional modifier, `` `perbegin' ``, for compatibility with the previous language version. This additional modifier is equivalent to the `primitive group` modifier.

Three base types, namely the two matrix types and the texref type, have a maximum computation frequency of primitive group. This restriction effectively limits how often matrices and texrefs may be computed or specified. This is somewhat of an arbitrary restriction for the matrix types, since there is no reason matrices cannot be computed per-vertex or per-fragment; however, we impose this restriction to simplify our compiler somewhat. The restriction on texrefs reflects the fact that in OpenGL, textures are specified for entire primitive groups and never more often (such as per-vertex).

Our language defines a set of rules to allow compilers to infer how often a particular value is computed. Such a set of rules is important both because it removes the need for the user to explicitly manage computation frequencies and because it allows for efficient generation of code when the user does not know the computation frequencies of certain values, in particular the intensity of light arriving at a surface, which can reasonably have any computation frequency. In the latter case, a compiler that can infer computation frequencies can properly choose, for example, vertex operations or fragment operations to integrate vertex and fragment lights, respectively.

Two rules are used to infer computation frequencies. The first deals with the default computation frequencies of shader parameters, while the second deals with the propagation of computation

frequencies across operators. By applying these rules, a compiler can always infer the computation frequency of a given operation.

All shader parameters have a well-defined default computation frequency that indicates how often the parameter may be specified. This frequency depends on the parameter's base type and the corresponding shader's type (surface or light):

Type	Default for surfaces	Default for lights
`bool`	vertex	primitive group
`clampf1`	vertex	primitive group
`clampf3`	vertex	primitive group
`clampf4`	vertex	primitive group
`float1`	vertex	primitive group
`float3`	vertex	primitive group
`float4`	vertex	primitive group
`matrix3`	primitive group	primitive group
`matrix4`	primitive group	primitive group
`texref`	primitive group	primitive group

Note that the defaults are different for surfaces and lights. This reflects the fact that typically light properties do not change more often than per-primitive-group.

The default shader parameter computation frequencies take effect when no computation frequency is specified with the parameter. An explicitly-specified computation frequency overrides the default.

Some examples:

```
surface shader float4 surf1 (float1 f) { ... }      // f is vertex
surface shader float4 surf2 (matrix3 m) { ... }     // m is primitive group
light shader float light1 (float1 f) { ... }        // f is primitive group
light shader float light2 (vertex float1 f) { ... } // f is vertex
light shader float light3 (matrix3 m) { ... }       // m is primitive group
```

Note that the rules for default computation frequencies do not apply to functions. They only apply to shaders:

```
surface surffunc1 (float1 f) { ... }   // no default computation frequency
```

In this case, the computation frequency of `f` is determined by the value passed to `f` when `surffunc1` is called.

The computation frequencies of computed values are determined by applying a second rule that propagates computation frequencies across operators. For the most part, we try to compute things as infrequently as possible. Specifically, the computation frequency of a computed value is the least frequent computation frequency possible given the constraint that a value must be computed at least as often as the most frequent value it depends on. For example, the result of adding a vertex

value to another vertex value is a vertex value, but adding a vertex value to a fragment value results in a fragment value, both because of the rule previously mentioned and because really it doesn't make any sense to try to obtain vertex values from fragment ones.

A number of operations can only be evaluated at certain computation frequencies. For example, texturing can only be computed per-fragment, while matrix-matrix multiplication can be computed at most per-primitive-group. We place additional constraints on computation frequencies to satisfy the limitations of each operation. We describe the details of these per-operator constraints in the operator appendix.

While the computation frequencies of computed values are inferred using the rules just described, they may be controlled by explicitly specifying computation frequencies. For example, if two vertex values N and L are to be used to compute `dot(N,L)`, the result of the dot product will normally be per-vertex. However, a per-fragment dot product can be achieved by first casting N or L (or both) to a fragment value:

```
float3 Nf = (fragment float3) N;   // cast N, fragment Nf inferred
float3 Lf = (fragment float3) L;   // cast L, fragment Lf inferred
// compute and use dot(Nf,Lf)...

fragment float3 Nf = N;            // use implicit cast from assign
fragment float3 Lf = L;            // use implicit cast from assign
// compute and use dot(Nf,Lf)...

dot(N, (fragment float3)L)...      // cast L only
```

In all three cases, once a fragment version of N or L is computed, the resulting dot product is inferred to be evaluated per-fragment.

Type conversion

A number of type conversions are permitted, including conversion of clamped values to float values, conversion of float values to clamped values, conversion from one computation frequency to a more-frequency computation frequency, and conversion of non-per-light values to per-light values.

Converting clamped values to float values has no effect except perhaps one of number representation (specifically, floating point or fixed point). Also, since floating-point values are more general than clamped floating-point values, this conversion is considered a promotion. Before performing an operation that involves both clamped and unclamped values, clamped values are automatically promoted to unclamped values.

Converting a float value to a clamped value clamps the float value to [0,1]. The number representation possibly changes also. This conversion may be performed explicitly using a type cast, or implicitly when assigning a `float` value to a `clampf` variable.

Conversion from one computation frequency to another is only possible if the new computation frequency is more frequent than the old one. In most cases, such a conversion simply replicates the old value at the new computation frequency; however, the conversion from vertex to fragment is special. In this case, vertex values are interpolated between vertices to obtain a fragment value. The exact nature of the interpolation is currently being left unspecified. Our compiler follows what OpenGL specifies, i.e. texture coordinates are perspective-correct while color values are not necessarily that way.

The conversion of the computation frequencies of operands to an operator is performed automatically as necessary for each operator. This process follows the rules for operator overloading and the function prototypes for operators discussed in later sections.

A non-per-light value may be converted into a per-light value. Performing this conversion has the effect of replicating the non-per-light value for every light.

Unlike in C, there is no way to interpret the value of a comparison numerically.

Global variables

Our system supports user-defined global variables as long as they are constant and their values are specified. Globals must be explicitly declared as constant:

```
constant float4 Red = { 1, 0, 0, 1 };  // valid
constant float4 Red;                   // error: missing definition
float4 Red = { 1, 0, 0, 1 };           // error: missing constant keyword

constant float4 DarkRed = 0.5 * Red;   // functions of constants are valid
```

Predefined globals

A number of global values are predefined and initialized on demand before a shader executes, or, in the case of predefined perlight globals, before each evaluation of the expression integrated by the corresponding integrate() operator. The predefined light shader global variables are:

```
vertex float3 S;          // light-space surface vector, normalized
vertex float Sdist;       // distance to surface point
```

The predefined surface shader globals are:

```
vertex float3 N;          // eye-space normal vector, normalized
vertex float3 T;          // eye-space tangent vector, normalized
vertex float3 B;          // eye-space binormal vector, normalized
vertex float3 E;          // eye-space eye vector, normalized

vertex float4 P;          // eye-space surface position, w=1
vertex float4 Pobj;       // object-space surface position, w=1
```

```
perbegin float4 Ca;        // color of global ambient light

vertex float4 Cprev;       // previous framebuffer color
                           // (note: not supported by 'nv' fragment backend)

vertex perlight float3 L;  // eye-space light vector, normalized
vertex perlight float3 H;  // eye-space halfangle vector, normalized

vertex perlight float4 Cl; // color of light (from a light shader)
```

Note that the definitions of the various globals currently cause light shaders to be evaluated in light space and surface shaders to be evaluated in eye space. Light space is defined by the light's position and orientation, while eye space is defined by the viewer's position and orientation.

The use of builtin parameters implicitly makes a shader dependent on one or more implicit shader parameters which are used to evaluate the builtin parameters. It is important to recognize these implicit shader parameters even though they are not a formal part of the language, since ultimately the user must set these parameters in addition to all those explicitly required by the active surface and light shaders. The implicit parameters are:

```
perbegin float4 __ambient;            // color of global ambient light
perbegin matrix4 __modelview;         // modelview matrix
perbegin matrix4 __projection;        // projection matrix

vertex float3 __normal;               // object-space normal vector
vertex float3 __tangent;              // object-space tangent vector
vertex float3 __binormal;             // object-space binormal vector
vertex float4 __position;             // object-space surface position

perbegin perlight float4 __lightpos;  // homogeneous position of light
perbegin perlight float3 __lightdir;  // unnormalized eye-space light direction
perbegin perlight float3 __lightup;   // unnormalized eye-space light up vector
```

Perlight builtin parameters must be specified once per active light shader.

Note that all shaders depend on `__modelview`, `__projection`, and `__position`.

Function overloading

Our language allows functions to be overloaded in a manner similar to C++. Overloading allows for many functions to be available when a function is called. Availability is defines as a function with the same name and number of parameters. We define a set of rules to select which function to select when more than one choice is available. The rules examine the base types of the parameters used in the call to form groups of matching functions.

The first group consists of functions whose parameter base types match the base types of the parameters in the call exactly.

The second group consists of functions whose parameter base types match the base types of the parameters in the call through the possible use of promotion. In particular, we consider the promotion of clamped floats to floats to form matches.

The third group consists of functions whose parameter base types match the base types of the parameters in the call through the use of both promotion and demotion.

The first group is checked first. If empty, the second group is checked, and likewise for the third group. If all three groups are empty, there is no match, and an error is generated. If any group being checked has more than one choice available, the call is ambiguous, and an error is generated. A match is found only if exactly one match is available in the first non-empty group.

This overloading mechanism is used for user-defined functions as well as builtin functions and builtin operators. Builtin functions and operators are defined using function prototypes in the operator appendix, below.

Conditional compilation

Today's hardware platforms offer differing sets of functionality. Some operators are not available on all hardware. To solve this problem, our language supports conditional compilation using a very-limited subset of C-preprocessor directives. We support:

```
#if <integer>
#ifdef <identifier>
#ifndef <identifier>
#else
#endif
#define <identifier>
#undef <identifier>
```

To promote the creation of function libraries, we also provide a limited include directive:

```
#include "<filename>"
```

We only support relative filenames, which must be double-quoted. We do not support angle-bracked filenames for searching include directories.

Our compiler predefines a number of identifiers based on whether or not certain hardware features are available. These identifiers are:

```
HAVE_FRAGMENT_SUBTRACT
HAVE_TEXTURE_3D
HAVE_CUBEMAP
HAVE_BUMPOPS
HAVE_REGISTER_COMBINERS
HAVE_FRAGMENT_INDEX
HAVE_FRAGMENT_COMPARES
```

The HAVE_FRAGMENT_SUBTRACT, HAVE_TEXTURE_3D, and HAVE_CUBEMAP identifiers indicate whether or not the subtract operator is available per-fragment, whether or not the texture3d operator is available, and whether or not the cubemap operator is available, respectively. The HAVE_BUMPOPS identifier indicates whether or not the bumpdiff and bumpspec operators are available. The HAVE_REGISTER_COMBINERS identifier covers the availability of the following operators per-fragment: dot, select, rgb, blue, alpha, lthalf, cubenorm. The HAVE_FRAGMENT_INDEX identifier indicates whether or not the [] operator is available per-fragment. The HAVE_FRAGMENT_COMPARES identifier indicates whether or not the ==, !=, >, <, >=, and <= operators are available per-fragment.

Builtin Operators and Functions

In this appendix, we describe the enumerate the builtin operators and functions made available by our language. Except for the syntax by which they are referred to, builtin operators and functions behave identically.

Every builtin operator and function has a range of computation frequencies at which it may be evaluated; the range specifies both a minimum and a maximum frequency.

As described earlier, values are evaluated as infrequently as possible. We define this computation frequency precisely as the maximum frequency among all of an operator's operands and the operator's miminum computation frequency.

Minimum and maximum computation frequencies limit the kinds of operations available at each computation frequency. For example, they restrict many matrix manipulation operations to a maximum computation frequency of per-primitive-group, and they force texture mapping to be per-fragment.

An error is generated if an operator's evaluation computation frequency exceeds the operator's maximum computation frequency.

In addition to each operator having a range of computation frequencies, every operand of every operator also has an associated range of computation frequencies. In most cases, this range has a minimum frequency of constant and a maximum frequency equal to the maximum frequency of the operator itself, but in a few cases, the range is more restrictive. For example, current hardware does not support the use of per-fragment texture coordinates. We therefore limit the maximum computation frequency of texture coordinates to vertex values.

In cases where the minimum frequency of an operand is not met, the value passed to the operand is automatically cast to an appropriate computation frequency. In cases where the maximum frequency of an operand is exceeded, an error is generated.

We now list all of the available operators. In the listings below, ranges are specified using a `[min:]max` syntax. For operators, if the `min` is unspecified, it defaults to constant. For operands, if the `min` and `max` are unspecified, the range defaults to the range of the corresponding operator, otherwise if only the `min` is unspecified, the min defaults to the `max`.

```
fragment float1 operator+ (float1, float1)
fragment float3 operator+ (float3, float3)
fragment float4 operator+ (float4, float4)
fragment clampf1 operator+ (clampf1, clampf1)
fragment clampf3 operator+ (clampf3, clampf3)
fragment clampf4 operator+ (clampf4, clampf4)

fragment float1 operator- (float1, float1)
fragment float3 operator- (float3, float3)
fragment float4 operator- (float4, float4)
fragment clampf1 operator- (clampf1, clampf1)
fragment clampf3 operator- (clampf3, clampf3)
fragment clampf4 operator- (clampf4, clampf4)
```

```
fragment float1 operator* (float1, float1)
fragment float3 operator* (float3, float3)
fragment float3 operator* (float1, float3)
fragment float3 operator* (float3, float1)
fragment float4 operator* (float4, float4)
fragment float4 operator* (float1, float4)
fragment float4 operator* (float4, float1)
fragment clampf1 operator* (clampf1, clampf1)
fragment clampf3 operator* (clampf3, clampf3)
fragment clampf3 operator* (clampf1, clampf3)
fragment clampf3 operator* (clampf3, clampf1)
fragment clampf4 operator* (clampf4, clampf4)
fragment clampf4 operator* (clampf1, clampf4)
fragment clampf4 operator* (clampf4, clampf1)
```

```
perbegin matrix3 operator* (matrix3, matrix3)
perbegin matrix4 operator* (matrix4, matrix4)
vertex float3 operator* (matrix3, float3)
vertex float4 operator* (matrix4, float4)

vertex float1 operator/ (float1, float1)
vertex float3 operator/ (float3, float3)
vertex float3 operator/ (float1, float3)
vertex float3 operator/ (float3, float1)
vertex float4 operator/ (float4, float4)
vertex float4 operator/ (float1, float4)
vertex float4 operator/ (float4, float1)
vertex clampf1 operator/ (clampf1, clampf1)
vertex clampf3 operator/ (clampf3, clampf3)
vertex clampf3 operator/ (clampf1, clampf3)
vertex clampf3 operator/ (clampf3, clampf1)
vertex clampf4 operator/ (clampf4, clampf4)
vertex clampf4 operator/ (clampf1, clampf4)
vertex clampf4 operator/ (clampf4, clampf1)

fragment float1 operator- (float1)
fragment float3 operator- (float3)
fragment float4 operator- (float4)

fragment float1 operator[] (float3)
fragment float1 operator[] (float4)
fragment clampf1 operator[] (clampf3)
fragment clampf1 operator[] (clampf4)
perbegin float3 operator[] (matrix3)
perbegin float4 operator[] (matrix4)

vertex float3 operator{} (float, float, float)
vertex float4 operator{} (float, float, float, float)
vertex clampf3 operator{} (clampf, clampf, clampf)
vertex clampf4 operator{} (clampf, clampf, clampf, clampf)
fragment float4 operator{} (float3 rgb, float1 alpha)
fragment clampf4 operator{} (clampf3 rgb, clampf1 alpha)
perbegin matrix3 operator{} (float3, float3, float3)
perbegin matrix4 operator{} (float4, float4, float4, float4)

fragment bool operator== (float, float)
fragment bool operator!= (float, float)
fragment bool operator> (float, float)
fragment bool operator< (float, float)
fragment bool operator>= (float, float)
fragment bool operator<= (float, float)

fragment bool operator== (clampf, clampf)
fragment bool operator!= (clampf, clampf)
fragment bool operator> (clampf, clampf)
fragment bool operator< (clampf, clampf)
fragment bool operator>= (clampf, clampf)
fragment bool operator<= (clampf, clampf)

fragment float4 operator blend (float4, float4)
fragment clampf4 operator blend (clampf4, clampf4)
fragment float4 operator over (float4, float4)
fragment clampf4 operator over (clampf4, clampf4)
fragment float4 operator blend_over (float4, float4)
fragment clampf4 operator blend_over (clampf4, clampf4)

surface fragment float1 operator integrate (float1)
surface fragment float3 operator integrate (float3)
surface fragment float4 operator integrate (float4)
surface fragment clampf1 operator integrate (clampf1)
surface fragment clampf3 operator integrate (clampf3)
surface fragment clampf4 operator integrate (clampf4)

vertex bool operator () (bool)
fragment float operator () (float)
fragment float3 operator () (float3)
fragment float4 operator () (float4)
fragment clampf operator () (clampf)
fragment clampf3 operator () (clampf3)
fragment clampf4 operator () (clampf4)
perbegin matrix3 operator () (matrix4)
perbegin matrix4 operator () (matrix4)
perbegin texref operator () (texref)

constant matrix3 identity3 ()
constant matrix4 identity ()

perbegin matrix3 affine (matrix4)
perbegin matrix3 invert (matrix3)
perbegin matrix3 rotate3 (float angle, float x, float y, float z)
perbegin matrix3 scale3 (float x, float y, float z)
perbegin matrix3 transpose (matrix3)
perbegin matrix4 frustum (float l, float r, float b, float t, float n, float f)
perbegin matrix4 invert (matrix4)
perbegin matrix4 lookat (float ex, float ey, float ez, float cx, float cy,
                         float cz, float ux, float uy, float uz)
perbegin matrix4 ortho (float l, float r, float b, float t, float n, float f)
perbegin matrix4 rotate (float angle, float x, float y, float z)
perbegin matrix4 scale (float x, float y, float z)
perbegin matrix4 translate (float x, float y, float z)
perbegin matrix4 transpose (matrix4)

vertex float clamp (float val, float lo, float hi)
vertex float3 clamp (float3 val, float lo, float hi)
vertex float3 clamp (float3 val, float3 lo, float3 hi)
vertex float4 clamp (float4 val, float lo, float hi)
vertex float4 clamp (float4 val, float4 lo, float4 hi)
vertex float3 cross (float3, float3)
vertex float dot (float4, float4)
vertex float length (float3)
vertex float length (float4)
vertex float max (float, float)
vertex float3 max (float3, float3)
vertex float4 max (float4, float4)
vertex float min (float, float)
vertex float3 min (float3, float3)
vertex float4 min (float4, float4)
vertex float3 normalize (float3)
vertex float4 normalize (float4)
vertex float pow (float val, float exp)
vertex float3 reflect (float3 vec, float3 norm)
vertex float sqrt (float)
vertex float cos (float)
vertex float sin (float)
vertex float ceil (float)
vertex float floor (float)
vertex float mod (float, float)
vertex float trunc (float)

fragment float dot (float3, float3)
fragment float1 select (bool, float1, float1)
fragment float3 select (bool, float3, float3)
fragment float4 select (bool, float4, float4)
fragment clampf1 select (bool, clampf1, clampf1)
fragment clampf3 select (bool, clampf3, clampf3)
fragment clampf4 select (bool, clampf4, clampf4)
fragment float3 rgb (float1)
fragment float3 rgb (float4)
fragment clampf3 rgb (clampf1)
fragment clampf3 rgb (clampf4)
fragment float1 blue (float3)
fragment float1 blue (float4)
fragment clampf1 blue (clampf3)
fragment clampf1 blue (clampf4)
fragment float1 alpha (float4)
```

```
fragment clampf1 alpha (clampf4)
fragment bool lthalf (float1)
fragment bool lthalf (clampf1)

fragment:fragment clampf4 lut (fragment clampf4)
fragment:fragment clampf4 texture (texref tex, constant:vertex float3 coord)
fragment:fragment clampf4 texture (texref tex, constant:vertex float4 coord)
fragment:fragment clampf4 texture3d (texref tex, constant:vertex float3 coord)
fragment:fragment clampf4 texture3d (texref tex, constant:vertex float4 coord)
fragment:fragment clampf4 cubemap (texref ref, constant:vertex float3 coord)
fragment:fragment clampf4 cubemap (texref ref, constant:vertex float4 coord)
fragment:fragment clampf3 cubenorm (constant:vertex float3 vec)
fragment:fragment clampf4 bumpdiff (texref ref, constant:vertex float4 coord,
                                    constant:vertex float3 Ltan)
fragment:fragment clampf4 bumpspec (texref ref, constant:vertex float4 coord,
                                    constant:vertex float3 Htan)
```

Not all operations are supported by all hardware at all computation frequencies. The compiler is allowed to generate an error when an unsupported operation is used. The section regarding conditional compilation enumerates the most important sets of operators that fall into this category.

Grammar

The following grammar describes the overall organization of the language.

```
PROGRAM : DECL_LISTopt

DECL_LIST : DECL_LISTopt DECL

DECL : TYPE IDENT ;
     | TYPE IDENT = EXPR ;
     | TYPE IDENT ( PARAM_LISTopt ) { STMT_LIST }

TYPE : MOD_LISTopt BASE_TYPE

MOD_LIST : MOD_LISTopt MOD

MOD : constant | primitive group | vertex | fragment | light | surface |
      shader | perlight | perbegin

BASE_TYPE : bool | clampf | clampf1 | clampf3 | clampf4 | clampfv |
            float | float1 | float3 | float4 | floatv | matrix3 | matrix4 |
            matrix | texref

PARAM_LIST : PARAM
           | PARAM_LIST ',' PARAM

PARAM : TYPE IDENT

STMT_LIST : STMT_LISTopt STMT

STMT : TYPE IDENT ;
     | TYPE IDENT = EXPR ;
     | EXPR ;
     | return EXPR ;
     | ;

EXPR : UNARY = EXPR
     | EXPR BINOP EXPR
     | UNARY

BINOP : == | != | > | < | >= | <= | + | - | blend | over | blend_over | * | /

UNARY : - UNARY
      | ( TYPE ) UNARY
      | PRIMARY

PRIMARY : ( EXPR )
        | { EXPR_LIST }
        | IDENT
        | PRIMARY [ INTEGER ]
        | integrate ( EXPR )
        | IDENT ( EXPR_LISTopt )
        | INTEGER
        | FLOAT

EXPR_LIST : EXPR
          | EXPR_LIST , EXPR
```

The following non-terminals are described by regular expressions:

```
IDENT : [_a-zA-Z][_a-zA-Z0-9]*
INTEGER : [0-9]+
FLOAT : (([0-9]+(\.[0-9]*)?)|(\.[0-9]+))([eE][-+]?[0-9]+)?f?
```

Sample shaders

The following example shaders serve to illustrate how the shading language might be used to implement a number of interesting shading effects.

```
// Useful constants

constant float4 Zero = { 0, 0, 0, 0 };
constant float4 Black = { 0, 0, 0, 1 };
constant float4 White = { 1, 1, 1, 1 };

constant float pi = 3.14159;

// Light shaders

light float
atten (float ac, float al, float aq)
{
   return 1.0 / ((aq * Sdist + al) * Sdist + ac);
}

light shader float4
simple_light (float4 color, float ac, float al, float aq)
{
   return color * atten(ac, al, aq);
}

float
smoothstep (float value, float min, float max)
{
   float t = clamp((value - min) / (max - min), 0, 1);
   return t * t * (3 - 2 * t);
}

float
smoothspot (float spot_cos, float inner_edge_angle, float outer_edge_angle)
{
   float inner_cos = cos(inner_edge_angle * pi / 180);
   float outer_cos = cos(outer_edge_angle * pi / 180);
   return smoothstep(spot_cos, outer_cos, inner_cos);
}

light shader float4
spotlight (float4 color, float ac, float al, float aq)
{
   float4 Cl = smoothspot(-S[2], 15, 30) * color * atten(ac, al, aq);
   return Cl;
}

light float4
star_projector_f (float4 color, float ac, float al, float aq, texref stars,
                  float time)
{
   float4 Cl = smoothspot(-S[2], 15, 30) * color * atten(ac, al, aq);
   float4 uv = { S[0], S[1], 0, -S[2] }; // project
   matrix4 t_rot = rotate(time * 15, 0, 0, 1);
   return Cl * texture(stars, t_rot * scale(1.5, 1.5, 1) * uv);
}

light shader float4
star_projector (float4 color, float ac, float al, float aq, texref stars)
{
   return star_projector_f(color, ac, al, aq, stars, 0);
}

light shader float4
star_projector_anim (float4 color, float ac, float al, float aq, texref stars,
                     float time)
{
   return star_projector_f(color, ac, al, aq, stars, time);
}

// Reflection models

surface float4
lightmodel (float4 a, float4 d, float4 s, float4 e, float sh)
{
   perlight float diffuse = dot(N,L);
   perlight float specular = pow(max(dot(N,H),0),sh);
   perlight float4 fr = select(diffuse > 0, d * diffuse + s * specular, Zero);
   return a * Ca + integrate(fr * Cl) + e;
}

surface float4
lightmodel_diffuse (float4 a, float4 d)
{
   perlight float diffuse = dot(N,L);
   perlight float4 fr = select(diffuse > 0, d * diffuse, Zero);
   return a * Ca + integrate(fr * Cl);
}

surface float4
lightmodel_specular (float4 s, float4 e, float sh)
{
   perlight float diffuse = dot(N,L);
   perlight float specular = pow(max(dot(N,H),0),sh);
   perlight float4 fr = select(diffuse > 0, s * specular, Zero);
   return integrate(fr * Cl) + e;
}

surface float4
lightmodel_anisotropic_u (float4 a, float4 d, float4 s, float4 e, float sh)
{
   float EdotT = dot(E,T);
   perlight float LdotT = dot(L,T);
   perlight float diff = sqrt(1 - LdotT * LdotT);
   perlight float spec = max(diff * sqrt(1 - EdotT*EdotT) - LdotT*EdotT, 0);
   perlight float4 fr = max(dot(N,L),0) * (d * diff + s * pow(spec,sh));
   return a * Ca + integrate(fr * Cl) + e;
}

surface float4
lightmodel_anisotropic_v (float4 a, float4 d, float4 s, float4 e, float sh)
{
   float EdotB = dot(E,B);
   perlight float LdotB = dot(L,B);
   perlight float diff = sqrt(1 - LdotB*LdotB);
   perlight float spec = max(diff * sqrt(1 - EdotB*EdotB) - LdotB*EdotB, 0);
   perlight float4 fr = max(dot(N,L),0) * (d * diff + s * pow(spec,sh));
   return a * Ca + integrate(fr * Cl) + e;
}

float center (float value) { return 0.5 * value + 0.5; }

surface float4
lightmodel_textured_anisotropic_u (texref anisotex, float4 a, float4 e)
{
   perlight float4 uv = { center(dot(T,E)), center(dot(T,L)), 0, 1 };
   // moving Cl helps group vertex/fragment computations
   //perlight float4 fr = max(dot(N,L),0) * texture(anisotex, uv);
   //return a * Ca + integrate(Cl * fr) + e;
   perlight float4 clfr = Cl * max(dot(N,L),0) * texture(anisotex, uv);
   return a * Ca + integrate(clfr) + e;
}
```

```
surface float4
lightmodel_textured_anisotropic_v (texref anisotex, float4 a, float4 e)
{
   perlight float4 uv = { center(dot(B,E)), center(dot(B,L)), 0, 1 };
   // moving Cl helps group vertex/fragment computations
   //perlight float4 fr = max(dot(N,L),0) * texture(anisotex, uv);
   //return a * Ca + integrate(Cl * fr) + e;
   perlight float4 clfr = Cl * max(dot(N,L),0) * texture(anisotex, uv);
   return a * Ca + integrate(clfr) + e;
}

surface float4
lightmodel_cartoon (texref cartoon, float4 a, float4 d)
{
   perlight float fr = max(dot(N,L),0);
   // clamp upper end to avoid texture border color
   float4 uv = { min(integrate(fr) + 0.2, 0.75), 0, 0, 1 };
   return a * Ca + d * texture(cartoon, uv);
}

// Standard material properties

constant float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
constant float4 Md = { 0.50, 0.50, 0.50, 1.00 };
constant float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
constant float4 Me = { 0.00, 0.00, 0.00, 0.00 };
constant float Msh = 300;

surface shader float4
default ()
{
   return lightmodel(Ma, Md, Ms, Me, Msh);
}

surface shader float4
cartoontest (texref cartoon)
{
   return lightmodel_cartoon(cartoon, {.4, .4, .8, 1}, {.4, .4, .8, 1});
}

surface shader float4
bowling_pin (texref pinbase, texref bruns, texref circle, texref coated,
         texref marks, float4 uv)
{
   float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
   float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
   matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
   matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
   matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
   matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
   matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
   float front = select(Pobj[2] >= 0, 1, 0);
   float back = select(Pobj[2] <= 0, 1, 0);
   float4 Base = texture(pinbase, t_base * uv_wrap);
   float4 Bruns = front * texture(bruns, t_bruns * uv_label);
   float4 Circle = front * texture(circle, t_circle * uv_label);
   float4 Coated = back * texture(coated, t_coated * uv_label);
   float4 Marks = texture(marks, t_marks * uv_wrap);
   float4 Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
   float4 Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, Zero, 20);
   return (Circle over (Bruns over (Coated over Base))) * (Marks * Cd) + Cs;
}

surface shader float4
glossy_moons (texref gloss, float4 uv)
{
   float4 base_a = { 0.1, 0.1, 0.1, 1.00 };
   float4 base_d = { 0.70, 0.40, 0.10, 1.00 };
   float4 base_s = { 0.07, 0.04, 0.01, 1.00 };
   float4 base_e = { 0.00, 0.00, 0.00, 1.00 };
   float base_sh = 15;

   float4 gloss_a = { 0.07, 0.04, 0.01, 1.00 };
   float4 gloss_d = { 0.07, 0.04, 0.01, 1.00 };
   float4 gloss_s = { 1.00, 0.90, 0.60, 1.00 };
   float4 gloss_e = { 0.00, 0.00, 0.00, 1.00 };
   float gloss_sh = 25;

   float4 Cbase = lightmodel(base_a, base_d, base_s, base_e, base_sh);
   float4 Cgloss = lightmodel(gloss_a, gloss_d, gloss_s, gloss_e, gloss_sh);

   float4 uv_gloss = invert(scale(.335,.335,1)) * uv;
   return Cbase + Cgloss * texture(gloss, uv_gloss);
}

surface shader float4
anisotropic_ball_vertex (texref star)
{
   float4 Ma = { 0.1, 0.1, 0.1, 1.0 };
   float4 Md = { 0.3, 0.3, 0.3, 1.0 };
   float4 Ms = { 0.7, 0.7, 0.7, 1.0 };
   float4 Me = { 0.0, 0.0, 0.0, 0.0 };
   float Msh = 15;
   float4 base = texture(star, { center(Pobj[2]), center(Pobj[0]), 0, 1 });
   return base * lightmodel_anisotropic_v(Ma, Md, Ms, Me, Msh);
}

surface shader float4
anisotropic_ball_texture (texref star, texref anisotex)
{
   float4 Ma = { 0.1, 0.1, 0.1, 1.0 };
   float4 Me = { 0.0, 0.0, 0.0, 0.0 };
   float4 base = texture(star, { center(Pobj[2]), center(Pobj[0]), 0, 1 });
   return base * lightmodel_textured_anisotropic_v(anisotex, Ma, Me);
}

surface float4
spheremap (texref env)
{
   float3 R = normalize(reflect(E,N) + { 0, 0, 1 });
   float4 uv = { center(R[0]), center(R[1]), 0, 1 };

   return texture(env, uv);
}

surface shader float4
sphere_map_env (texref env)
{
   return spheremap(env);
}

surface shader float4
poolball (texref one, float4 uv)
{
   float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
   float4 Md = { 0.50, 0.50, 0.50, 1.00 };
   float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
   float4 Me = { 0.00, 0.00, 0.00, 1.00 };
   float Msh = 127;
   float4 Cd = lightmodel_diffuse(Ma, Md);
   float4 Cs = lightmodel_specular(Ms, Me, Msh);
   matrix4 tm = invert(translate(0.35, 0.2, 0.0) * scale(0.3, 0.6, 1.0));
   return Cd * texture(one, tm * uv) + Cs;
}

surface shader float4
poolball_with_env (texref one, texref env, float4 uv)
{
   float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
   float4 Md = { 0.50, 0.50, 0.50, 1.00 };
   float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
   float4 Me = { 0.00, 0.00, 0.00, 1.00 };
   float Msh = 127;
   float4 Cd = lightmodel_diffuse(Ma, Md);
   float4 Cs = lightmodel_specular(Ms, Me, Msh);
   matrix4 tm = invert(translate(0.35, 0.2, 0.0) * scale(0.3, 0.6, 1.0));
   return Cd * texture(one, tm * uv) + (Cs + spheremap(env));
}
```

```
float4
turb (texref noise, float4 uv)
{
    float4 uv_0 = invert(rotate(30.2, 0, 0, 1) * scale(4, 4, 1)) * uv;
    float4 uv_1 = invert(rotate(-35.5, 0, 0, 1) * scale(2, 2, 1)) * uv;
    float4 uv_2 = invert(rotate(274.1, 0, 0, 1) * scale(1, 1, 1)) * uv;
    float4 N_0 = 0.57 * texture(noise, uv_0);
    float4 N_1 = 0.29 * texture(noise, uv_1);
    float4 N_2 = 0.14 * texture(noise, uv_2);
    return N_0 + N_1 + N_2;
}

surface shader float4
noise_2d_multipass (texref noise, float4 uv)
{
    return turb(noise, uv);
}

surface shader float4
noise_2d_multipass_specular_modulate (texref noise, float4 uv)
{
    float4 Cl = lightmodel(Ma, Md, Ms, Me, Msh);
    return Cl * turb(noise, uv);
}

surface shader float4
noise_2d_multipass_specular_separate (texref noise, float4 uv)
{
    float4 Cd = lightmodel_diffuse(Ma, Md);
    float4 Cs = lightmodel_specular(Ms, Me, Msh);
    return Cd * turb(noise, uv) + Cs;
}

float4
skymap (texref clouds, float4 dir, float time)
{
    dir = normalize(dir);
    dir = { dir[0], dir[1], 4 * (dir[2] + 0.707), 0 };
    dir = normalize(dir);
    float4 uv_lo = dir * { 2, 2, 0, 0 } + { time / 15 , time / 15, 0, 1 };
    float4 uv_hi = dir * { 3, 3, 0, 0 } + { time / 15 , time / 15, 0, 1 };
    float4 Lo = texture(clouds, uv_lo);
    float4 Hi = texture(clouds, rotate(125, 0, 0, 1) * uv_hi);
    // for now, do not use Lo over (Hi over { 0.6, 0.5, 1.0, 1.0 })
    // texture_env_combine does not do over correctly
    return Lo over Hi over { 0.6, 0.5, 1.0, 1.0 };
}

surface shader float4
quake_sky (texref clouds, float time)
{
    return skymap(clouds, { Pobj[0], -Pobj[2], Pobj[1], 0 }, time);
}

surface shader float4
bowling_pin_with_sky (texref pinbase, texref bruns, texref circle,
                texref coated, texref marks, float4 uv,
                texref clouds, float time)
{
    float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
    matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
    float front = select(Pobj[2] >= 0, 1, 0);
    float back = select(Pobj[2] <= 0, 1, 0);
    float4 Base = texture(pinbase, t_base * uv_wrap);
    float4 Bruns = front * texture(bruns, t_bruns * uv_label);
    float4 Circle = front * texture(circle, t_circle * uv_label);
    float4 Coated = back * texture(coated, t_coated * uv_label);
    float4 Marks = texture(marks, t_marks * uv_wrap);
    float Lscale = 0.5;
    float4 Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
    Cd = Cd * Lscale;
    float4 Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, Zero, 20);
    Cs = Cs * Lscale;
    float3 R = reflect(E,N);
    return (Circle over (Bruns over (Coated over Base))) * (Marks * Cd) + Cs +
                0.5 * skymap(clouds, { R[0], -R[2], R[1], 0 }, time);
}

#ifdef HAVE_BUMPOPS

surface shader float4
bowling_pin_bump (texref pinbase, texref bruns, texref circle, texref coated,
            texref marks, texref marksbump, float4 uv)
{
    float4 uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
    float4 uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    matrix4 t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix4 t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix4 t_circle = invert(translate(-0.8, -1.15, 0) * scale(1.4, 1.4, 1));
    matrix4 t_coated = invert(translate(2.6, -2.8, 0) * scale(-5.2, 5.2, 1));
    matrix4 t_marks = invert(translate(2.0, 7.5, 0) * scale (4, -15, 1));
    float front = select(Pobj[2] >= 0, 1, 0);
    float back = select(Pobj[2] <= 0, 1, 0);
    float4 Base = texture(pinbase, t_base * uv_wrap);
    float4 Bruns = front * texture(bruns, t_bruns * uv_label);
    float4 Circle = front * texture(circle, t_circle * uv_label);
    float4 Coated = back * texture(coated, t_coated * uv_label);
    float4 uv_marks = t_marks * uv_wrap;
    float4 Marks = texture(marks, uv_marks);
    perlight float3 Lt = { dot(T,L), dot(B,L), dot(N,L) };
    perlight float3 Ht = { dot(T,H), dot(B,H), dot(N,H) };
    float4 Ma = {.4,.4,.4,1};
    float4 Md = {.5,.5,.5,1};
    float4 Ms = {.3,.3,.3,1};
    float4 Kd = (Circle over (Bruns over (Coated over Base))) * Marks;
    return Kd * Ma +
        integrate(Cl * (Kd * Md * bumpdiff(marksbump, uv_marks, Lt)
                                    blend(ONE,SRC_ALPHA)
                                    Ms * bumpspec(marksbump, uv_marks,
Ht)));
}

#endif /* HAVE_BUMPOPS */

#ifdef HAVE_CUBEMAP

surface shader float4
cube_from_obj_normal (texref cube) {
    return cubemap(cube, {-1,-1,1}*__normal);
}

surface shader float4
poolball_with_cube (texref one, float4 uv, texref cube)
{
    float4 Ma = .5 * { 0.35, 0.35, 0.35, 1.00 };
    float4 Md = .5 * { 0.50, 0.50, 0.50, 1.00 };
    float4 Ms = .5 * { 1.00, 1.00, 1.00, 1.00 };
    float4 Me = .5 * { 0.00, 0.00, 0.00, 1.00 };
    float Msh = 127;
    float4 Cd = lightmodel_diffuse(Ma, Md);
    float4 Cs = lightmodel_specular(Ms, Me, Msh);
    matrix4 tm = invert(translate(0.35, 0.2, 0.0) * scale(0.3, 0.6, 1.0));
    float3 R = reflect(E,N);
    return Cd * texture(one, tm * uv) + Cs + 0.4 * cubemap(cube, {-1,-1,1}*R);
}

#endif /* HAVE_CUBEMAP */
```