Supplemental Material — Physically-Based Real-Time Lens Flare Rendering

Matthias Hullin[1*] Elmar Eisemann[2†] Hans-Peter Seidel[1*] Sungkil Lee[3,1‡]
[1] MPI Informatik [2] Télécom ParisTech/Saarland University [3] Sungkyunkwan University

Ray Tracing Code

In this section, we provide C-like example code for our basic ray tracing scheme, which can be run on a modern GPU shader.

The three main data structures are `LensInterface`, `Ray`, and `Intersection`. `LensInterface` describes a single optical surface and is defined as follows.

```
struct LensInterface {
    vec3    center; // center of sphere/plane on z-axis
    float   radius; // radius of sphere/plane
    vec3    n;      // refractive indices (n0, n1, n2)
    float   sa;     // nominal radius (from opt. axis)
    float   d1;     // coating thickness=lambdaAR/4/n1
    bool    flat;   // is this interface a plane?
} LensInterface INTERFACE[NUM_INTERFACE];
```

In our coordinate system, the origin is the center of sensor plane, and the optical axis is aligned along with z-axis; an input ray travels in negative z direction from the entrance plane to the sensor. Setting `n` to different values (a set of values per wavelength) provides spectral dispersion. `n0` and `n2` are the refractive indices of the first and second media, and `n1` the refractive index of an anti-reflective coating. The notation of the subscripts 0, 1, and 2 are used consistently throughout the program to indicate a medium before intersection, anti-reflective coating, and a medium after intersection. The array `INTERFACE` describes the lens system as a series of interfaces. The order is defined from the entrance plane (`INTERFACE[0]`) to the sensor plane (`INTERFACE[NUM_INTERFACE-1]`).

A `Ray` has a position and direction, plus some additional properties.

```
struct Ray {
    vec3 pos, dir;
    vec4 tex;   // (aperture tex coord, r_rel, intens)
};
```

`tex` encodes three elements: the texture coordinate on the aperture (`tex.xy`), the cumulative radius r_{rel} (see paper, Section 4.1) during the tracing (`tex.z`), and ray's intensity (`tex.w`: 0 means an invalid ray). Whenever the ray hits a surface, we compute the relative radius, and maintain the maximum of the relative radii in `tex.z`

The structure `Intersection` is defined as follows.

```
struct Intersection {
    vec3   pos;
    vec3   norm;
    float  theta;       // incident angle
    bool   hit;         // intersection found?
    bool   inverted;    // inverted intersection?
};
```

`inverted` is used to mark an inverted intersection, explained in the paper.

In addition, we define two more variables to control the fixed-order intersection tests. The structure `BOUNCE` and `LENGTH` is defined as follows.

```
int2 BOUNCE[NUM_BOUNCE];
int  LENGTH[NUM_BOUNCE];
```

In the preprocessing stage, we enumerate all bounces, and record the two surfaces that reflect a ray into `BOUNCE` in order. We have three phases of tracing: (1) the entrance plane to the first reflection, (2) the first reflection to the second reflection, and (3) the second reflection to the sensor. `LENGTH` contains the number of interfaces where a ray passes through to the sensor plane.

Here is the main ray tracing routine.

```
Ray trace (
    int bid,      // index of current bounce/ghost
    Ray r,        // input ray from the entrance plane
    float lambda  // wavelength of r
) {
    int2 STR = BOUNCE[bid]; // read 2 surf.s to reflect
    int  LEN = LENGTH[bid]; // length of intersections

    // initialization
    int PHASE = 0;  // ray-tracing phase
    int DELTA = 1;  // delta for for-loop
    int T     = 1;  // index of target intrface to test

    int k;
    for( k=0; k < LEN; k++, T += DELTA )
    {
        LensInterface F = INTERFACE[T];

        bool bReflect = (T==STR[PHASE]) ? true : false;
        if (bReflect){ DELTA=-DELTA; PHASE++; }

        // intersection test
        Intersection i = F.flat ? testFLAT(r,F) : ↩
             testSPHERE(r,F);
        if (!i.hit) break; // exit upon miss

        // record texture coord. or max. rel. radius
        if (!F.flat)
            r.tex.z = max(r.tex.z,
                          i.pos.xy.length() / F.sa);
        else if(T==AP_IDX) // iris aperture plane
            r.tex.xy=i.pos.xy/INTERFACE[AP_IDX].radius;

        // update ray direction and position
        r.dir = normalize(i.pos-r.pos);
        if (i.inverted) r.dir *= -1; // correct an ↩
             inverted ray
        r.pos = i.pos;

        // skip reflection/refraction for flat surfaces
        if (F.flat) continue;

        // do reflection/refraction for spher. surfaces
        float n0 = r.dir.z < 0?F.n.x:F.n.z;
        float n1 = F.n.y;
        float n2 = r.dir.z < 0?F.n.z:F.n.x;

        if (!bReflect) // refraction
        {
            r.dir = refract(r.dir,i.norm,n0/n2);
            if(r.dir==0) break; // total reflection
        }
        else // reflection with AR Coating
        {
            r.dir = reflect(r.dir,i.norm);
            float R = ↩
                 FresnelAR(i.theta,lambda,F.d1,n0,n1,n2);
            r.tex.a *= R;       // update ray intensity
        }
    }

    if (k<LEN) r.tex.a=0; // early-exit rays = invalid
    return r;
}
```

*{hullin|hpseidel}@mpi-inf.mpg.de
†elmar.eisemann@telecom-paristech.fr
‡sungkil@skku.edu. Corresponding author.

This main program is called for each ray on the input grid. The intensity of output rays are first smoothed using neighboring rays before rasterization. Then, triangles, formed by three output rays on the sensor plane, are rasterized with additive blending. The intensity (color) of an incoming fragment is determined by its own intensity (`r.tex.a`), the wavelength, and the aperture texture (read from `r.tex.xy`). If `r.tex.z` of the fragment is greater than 1, we discard such fragments.

Here, we show example functions for intersection test with flat surfaces (e.g., entrance, aperture, and sensor planes) and spherical surfaces. We assume the input ray is normalized before the tests.

```
Intersection testFLAT( Ray r, LensInterface F ) {
    Intersection i;
    i.pos = r.pos
          + r.dir*((F.center.z - r.pos.z) / r.dir.z);
    i.norm = r.dir.z > 0 ? vec3(0,0,-1) : vec3(0,0,1);
    i.theta = 0; // meaningless
    i.hit = true;
    i.inverted = false;
    return i;
}

Intersection testSPHERE( Ray r, LensInterface F ) {
    Intersection i;
    vec3  D    = r.pos - F.center;
    float B    = dot(D,r.dir);
    float C    = dot(D,D) - F.radius * F.radius;
    float B2_C = B*B-C;
    if (B2_C < 0)
        // no intersection
        { i.hit = false; return i; }

    float sgn = (F.radius * r.dir.z) > 0 ? 1 : -1;
    float t = sqrt(B2_C) * sgn-B;
    i.pos = r.dir * t + r.pos;
    i.norm = normalize(i.pos - F.center);
    if (dot(i.norm, r.dir) > 0) i.norm = -i.norm;
    i.theta = acos(dot(-r.dir, i.norm));
    i.hit = true;
    i.inverted = t < 0; // mark an inverted ray
    return i;
}
```

Anti-reflective Coating

Finally, we provide a computation scheme for the reflectivity $R(\lambda, \theta)$ of a surface coated with a single dielectric layer. Although our overall model of the optical system assumes unpolarized light, we distinguish between p- and s-polarized light in the following computations, since light waves only interfere with other waves of the same polarization.

To simulate a quarter-wave coating optimized for wavelength `lambda0` under normal incidence, and given refractive indices `n0` (e.g. air) and `n2` (e.g. glass), `d1` and `n1` (thickness and refractive index of coating) would be chosen as

```
n1 = max(sqrt(n0*n2), 1.38); // 1.38=lowest achievable
d1 = lambda0 / 4 / n1;       // phase delay
```

```
float FresnelAR (
    float theta0,   // angle of incidence
    float lambda,   // wavelength of ray
    float d1,       // thickness of AR coating
    float n0,       // RI (refr. index) of 1st medium
    float n1,       // RI of coating layer
    float n2        // RI of the 2nd medium
) {
    // refraction angles in coating and the 2nd medium
    float theta1 = asin(sin(theta0)*n0/n1);
    float theta2 = asin(sin(theta0)*n0/n2);

    // amplitude for outer refl./transmission on
        topmost interface
    float rs01 = -sin(theta0-theta1)/sin(theta0+theta1);
    float rp01 = tan(theta0-theta1)/tan(theta0+theta1);
    float ts01 =
        2*sin(theta1)*cos(theta0)/sin(theta0+theta1);
    float tp01 = ts01*cos(theta0-theta1);

    // amplitude for inner reflection
    float rs12 = -sin(theta1-theta2)/sin(theta1+theta2);
    float rp12 = +tan(theta1-theta2)/tan(theta1+theta2);

    // after passing through first surface twice:
    // 2 transmissions and 1 reflection
    float ris = ts01*ts01*rs12;
    float rip = tp01*tp01*rp12;

    // phase difference between outer and inner
        reflections
    float dy = d1*n1;
    float dx = tan(theta1)*dy;
    float delay = sqrt(dx*dx+dy*dy);
    float relPhase = 4*PI/lambda*(delay-dx*sin(theta0));

    // Add up sines of different phase and amplitude
    float out_s2 = rs01*rs01 + ris*ris +
        2*rs01*ris*cos(relPhase);
    float out_p2 = rp01*rp01 + rip*rip +
        2*rp01*rip*cos(relPhase);

    return (out_s2+out_p2)/2;   // reflectivity
}
```