**Implementation Notes: Physically Based Lens Flares**

Posted on [**January 19, 2015**](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/)

The Implementation Notes series of blog posts is an attempt to outline the process of implementing a research paper or technique from a presentation. I find that more often than not when trying to recreate others results that not all the dots are connected. It takes some time and digging to get the whole picture, this is my attempt at sharing that background work.

In this edition of Implementation Notes (actually the first edition) we will be looking at two papers relating to rendering lens flares.  
[Physically-Based Real-Time Lens Flare Rendering](http://resources.mpi-inf.mpg.de/lensflareRendering/pdf/flare.pdf)[1](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/#fn-326-1)  
[Practical Real-Time Lens-Flare Rendering](https://graphics.tudelft.nl/Publications-new/2013/LE13/PracticalReal-TimeLens.pdf)[2](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/#fn-326-2)  
Some of the foundation work from the first paper will be presented and then an implementation based on the second paper will be outlined.

**Background**

Cameras are comprised of many complex systems, each of which can introduce visual artifacts. If these visual artifacts are not modeled to a certain extent in our rendered images then they will produce suspiciously clean results and look artificial. However, if we introduce some of these artifacts then the images can appear more natural to the viewer. One such artifact is internal reflections, sometimes referred to as lens flares. These two papers explore modeling lens flares as the result of light reflecting off the surfaces of the optics inside of the lens, hence the “Physically-Based”.

[](https://placeholderart.files.wordpress.com/2015/01/141230-012.jpg)

Lens flares from an old Vivitar zoom lens

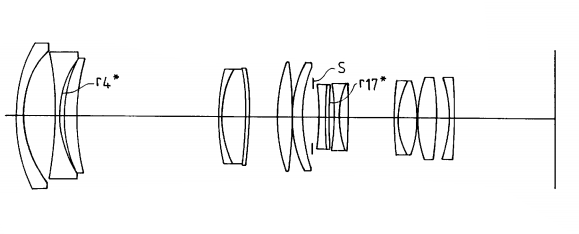
[](https://placeholderart.files.wordpress.com/2015/01/141230-020.jpg)

Lens flares from an old Vivitar zoom lens

The lens configuration inside most lenses is surprisingly complex and can consist of dozens of individual lens elements. As the light from the scene passes through each optical interface a portion is reflected and the rest is refracted. The reflected light then moves backwards through the optical system striking other optical interfaces along the way, the process then repeats. For every even number of bounces that occurs the reflected light is then directed back towards the image sensor; however, the light rays are no longer focused. This light represents a single lens flare, or ghost as referred to by the papers.

**Optical Systems**

This technique is heavily based off on the representation of the lens elements in an *Optical System.*  This referrers to the combination of lenses and empty spaces designed in such a way that light is focused on to an image plane. We begin with the assumption that the image plane *I* represents standard film (24x36mm) with an image circle diagonal of 43.3mm.

[](https://placeholderart.files.wordpress.com/2014/12/optical-system1.png)

Example Optical System (USPN5835272)

The optical axis of the system is the axis that runs through the center of the lens elements and intersects the center of the image plane. Lens elements are arranged along the optical axis either directly next to each other or separated by a small amount of air. The standard description of an optical system is a table of data that defines the interfaces between air and lens elements. The data looks something like this:

| # | r | d | n |
| --- | --- | --- | --- |
| 1 | 30.810 | 7.700 | 1.652 |
| 2 | -89.350 | 1.850 | 1.603 |
| … | … | … | … |

Here r is the radius of curvature for the interface, d is the distance to the **next** interface, and n is the refractive index of the lens element. The radius can be positive, indicating a convex surface, negative, indicating a concave surface, or 0, indicating a flat surface. Using this definition of an optical system allows us to use existing optical system data. The best place to find optical system descriptions is the patent database, be warned though I don’t know the legality of using the optical design data to reproduce the lens digitally.

When reading a lens patent there are a couple things to note. The first is that there is **a lot** of information in the patent! Most of it you don’t need and shouldn’t care about. The main thing you are looking for is the tabular data describing the lens system. In the table there is usually a header that describes the focal length of the lens and the f-number. This represents the field of view of the lens and essentially how much light the lens can let in respectfully. If you are a real ambitious go-getter you can use this to drive the settings on your [camera model](https://placeholderart.wordpress.com/2014/11/21/implementing-a-physically-based-camera-manual-exposure/).

It took me a while to find a zoom lenses that had a focal length that represented a normal game field of view (~70 degrees) and had a low f-number, meaning that the lens allows in a lot of light. So I can save you the pain patent searching and point you at a couple options:

[Nikon 28-75mm f/2.8 (US5835272)](http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5835272)  
[Canon 28-80 f/2.8 (US5576890)](http://patft.uspto.gov/netacgi/nph-Parser?Sect2=PTO1&Sect2=HITOFF&p=1&u=/netahtml/PTO/search-bool.html&r=1&f=G&l=50&d=PALL&RefSrch=yes&Query=PN/5576890)[Canon 36-135mm f/3.5-f/4.6 (USPN4629294)](http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=4629294.PN.&OS=PN/4629294&RS=PN/4629294)

I have found that there is some information missing when reading the patent, namely the heights of the various elements. Luckily the height that matters the most is the *aperture stop*and this can be calculated using the effective aperture diameter (*D*)[3](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/#fn-326-5). The effective aperture represents what the physical aperture looks like if you were looking through the front of the lens. Since the elements in front of the physical aperture often have a magnification property it will differ from the physical aperture stop. The formula for this is based only on the focal length, f and the f-number N:

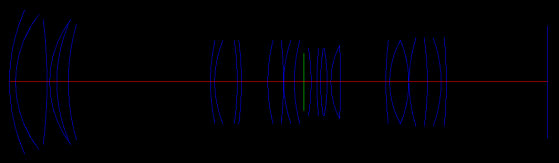
N=\frac{f}{D}

So for the Nikon zoom lens we can determine that at a focal length of 28mm and an f-number of roughly 2.8 that the effective aperture diameter should be 10mm. Once we are able to cast rays through the optical system this will be used to determine the size of the physical aperture.

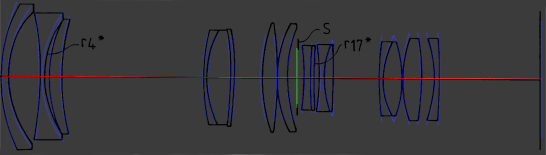
Quick note: On more modern lenses (1980s and onward) there are often aspherical lens elements in the optical system. Since the intent here is not to rely on the precise qualities of the optical system I chose to ignore these elements and instead treat them as spherical.

# Ray tracing an Optical System

The first step towards modeling lens flares is to establish the optical system we will be evaluating. This can be validated by simply drawing (in 2D) the profile of the optical system. To perform the drawing you simply walk over the elements of the optical system, drawing each and then increment the draw cursor down the optical axis by the *d* value for that element. Each interface is drawn as an angular patch using the radius of curvature (r) and angle \pm a given by a = asin(\frac{h}{r}). For the first pass I just took a guess at the various element heights making sure they didn’t intersect with the next elements. As mentioned before, we only care about the height of the aperture which will be computed. Here is what it the result should look like:

[](https://placeholderart.files.wordpress.com/2014/12/in-engine-optical-system-drawing.png)

In Engine Drawing of the optical system. Red – Optical Axis, Green – Aperture Stop, Blue – Optical Interfaces defined by the tabular patent data.

[](https://placeholderart.files.wordpress.com/2014/12/engine-optical-system-overlay.png)

Overlay of in engine representation and the patent diagram.

Next we will trace a single ray through the optical system considering only refraction, the pseudo code for this is:

OpticalInterface i = first;

while (i)

{

col = intersect(ray, i);

if (!col.valid)

break;

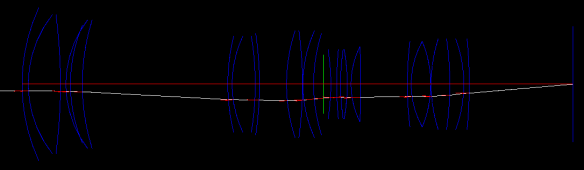
ray.pos += ray.dir \* col.dist;

ray.dir = refract(ray.dir, col.norm, i.n1/i.n2);

i = i.next;

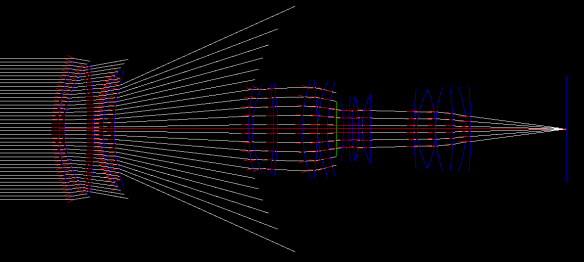
}

The result should look something like this:

[](https://placeholderart.files.wordpress.com/2014/12/single-ray.png)

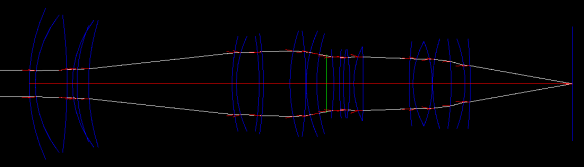
Single ray making it’s way through the optical system

Be mindful that there can be elements with radius 0, in these cases you should just do a ray plane intersection. Once we have a single ray going through we can cast a bunch of rays to see if they all converge on the focal plane. It is a good time to mention that the patents are specified assuming that you are focusing on an object at infinity. This means that all rays entering the lens are travelling in parallel.

[](https://placeholderart.files.wordpress.com/2014/12/multiple-rays.png)

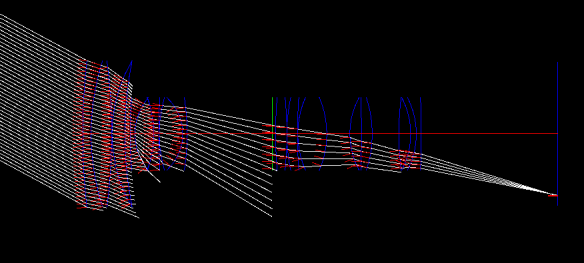
Rays focusing at the image plane

As you can see, the rays that make it through the optical system are focusing to a single point. The reason many rays are terminated prior to reaching the surface is that the effective aperture is much smaller than the first element. Recalling that we can compute the effective aperture by using the formula N=\frac{f}{D}, and that for the Nikon lens we found the D \approx 10mm. Now if we cast rays 5mm above and 5mm below the optical axis we can find determine the height of the aperture (green line) as the height of the ray when it passes through the aperture stop, in this case ~22mm diameter.

[](https://placeholderart.files.wordpress.com/2014/12/effective-pupil.png)

Rays cast at the edge of the effective aperture to compute the physical aperture size.

To further validate the optical system, we can sweep rays through the full FOV range specified by the patent[4](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/#fn-326-6). Remember it isn’t going to be perfect since we ignore aspherical elements but it should be relatively close. On a side note, this an interesting experiment because you can see that fewer rays make it through at steeper angles causing Vignetting[5](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/#fn-326-7).

[](https://placeholderart.files.wordpress.com/2014/12/fov-directional-rays.png)

Rays from a direction focusing on the focal plane (from the canon zoom lens)

Using this as the final test we can consider the optical system validated. If you have something horribly wrong with it you would notice it at this point. From here we can start to trace the actual flares.

# Flares/Ghosts

The first thing we need to do related to rendering flares is to enumerate the different paths that can end up back at the image plane. Since we lose a significant amount of intensity each time a ray reflects at an optical surface we will follow the papers and only consider rays which have bounced exactly twice. An additional consideration we can make is that a significant amount of light is lost when the light passes through the aperture stop. To factor this into the enumeration process we can say that we only care about light that passes the aperture stop exactly one time and bounces exactly two times. Another way of thinking about this is that we are only considering paths of light that bounce both times before the aperture or after the aperture. The pseudo code for enumerating the flares/ghosts is below:

a = getApertureIndex()

n = getNumElements()

for (i = 0; i < a; ++i)

for (j = i + 1; j < a; ++j)

OpticalGhost ghost;

ghost.\_reflections[0] = j;

ghost.\_reflections[1] = i;

ghosts.push\_back(ghost);

for (i = a + 1; i < n; ++i)

for (int j = i + 1; j < n; ++j)

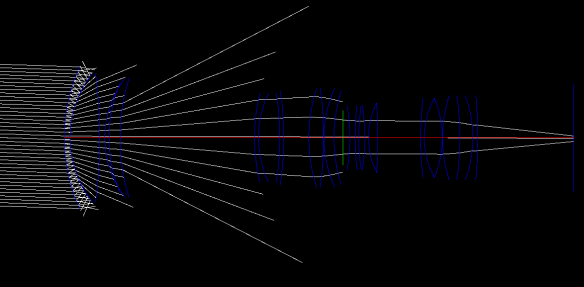
OpticalGhost ghost;

ghost.\_reflections[0] = j;

ghost.\_reflections[1] = i;

ghosts.push\_back(ghost);

Now that we have a list of all possible ghosts in the system we need to modify the ray tracing to account for reflecting at a certain interface. I’ll leave that as an exercise for the reader since it is fairly straight forward. The result of tracing a single ghost should look like this:

[](https://placeholderart.files.wordpress.com/2015/01/ghost-trace.png)

This result suggests that we will have a flare at the center of the screen and it will be clipped by the aperture plane (thus aperture shaped). This will be the end of the ray tracing approach presented by the first paper but we will use it to confirm an alternative approach, *Matrix Method.*

# Matrix Method

The matrix method is a general optics technique that approximates the effects of an optical system as a linear system. Since each optical interface is represented as a linear system the results can be concatenated into a single matrix describing the entire system. At the core approach of this is the paraxial approximation[6](https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares/#fn-326-3). This approximation states that for small angles (< 10 degrees) of \theta:

\sin \theta \approx \theta  
\tan \theta \approx \theta  
\cos \theta \approx 1

Using this approximation we can represent the transforms of our optical system, namely translation(T), reflection(L), and refraction(R), as 2×2 matrices functioning on the height and angle of a ray: \left[ \begin{array}{cc} H_{0}\\\theta_{0}\end{array} \right]. For full derivations of these matrices see [Matrix Methods in Paraxial Optics](http://www.erbion.com/index_files/Modern_Optics/Ch18.pdf).

T = \left[ \begin{array}{cc} 1&d \\ 0&1\end{array} \right]

R_{curved} = \left[ \begin{array}{cc} 1&0 \\ \frac{n_1 - n_2}{Rn_2}&\frac{n_1}{n_2}\end{array} \right]

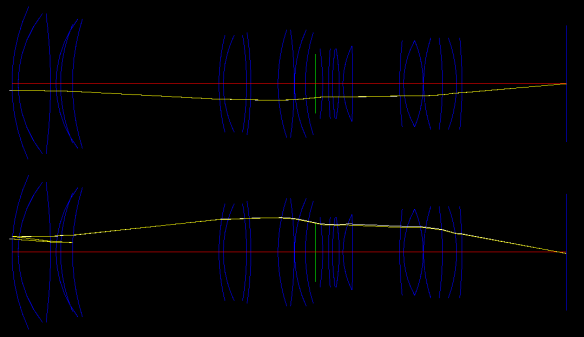
R_{flat} = \left[ \begin{array}{cc} 1&0 \\ 0&\frac{n_1}{n_2}\end{array} \right]

L_{curved} = \left[ \begin{array}{cc} 1&0 \\ \frac{2}{R}&1\end{array} \right]

L_{flat} = \left[ \begin{array}{cc} 1&0 \\ 0&1\end{array} \right]

So now instead of tracing N rays for each ghost we will simply concatenate the matrices that represent each optical interface and project a flare quad. Generally there will be pairs of transforms that translate to then refract by the interface, since we are talking about matrices the order is reverse R_n T_n. When we hit the interface that we should reflect at we multiply by L_{n} and then as we progress backwards through the system we apply the inverse of the above pairs, T_n R_{n}^{-1} (Translation is unaffected by the change in direction), until we hit the second reflection L_{n}^{-1}, at which point you resume concatenating the remaining pairs using the original R_n T_n. A simple example would be: M = R_2 T_2 R_1 T_1 L_{0}^{-1} T_1 L_1 T_1 R_0 T_0 representing a ray entering a 3 interface system translating to and refracting through the first element (R_0 T_0), translating and reflecting off the second (L_1 T_1), traveling back through the second (T_1), reflecting off the first and then propagating out of the system (R_2 T_2 R_1 T_1 L_{0}^{-1}).

**Important!**The inverse in this case doesn’t refer to the matrix inverse but rather the operation in the reverse direction i.e. R(n_1, n_2, r)^{-1} = R(n_2, n_1, -r), L(r)^{-1} = L(-r), and T(d)^{-1} = T(d). This isn’t mentioned in the original paper and if you use the standard matrix inverse the results will be quite a bit off.

[](https://placeholderart.files.wordpress.com/2015/01/matrix-method2.png)

While the matrix method (yellow) isn’t as accurate as ray tracing (white) it is pretty close in most situations we care about.

# Placing the ghosts

Using the matrices that describe the system, we generate a flare quad for the final flare rendering. To do this we actually need 2 matrices, the first is the matrix that describes the optical system up to the aperture stop (M_a), the second describes the optical system from the aperture stop to the sensor plane (M_s). M_a will be used to find the entrance pupil of the ghost and M_s will be used to find the projected aperture. If you recall from earlier, the entrance pupil is the aperture as seen from the front of the lens following the path of light that we care about. Essentially we want solve for the height of the entrance pupil H_{e_1} and H_{e_2} using the direction of the light \theta_e, the height of the aperture h_a, and the optical system matrix of the lens prior to the aperture M_a.

\left[ \begin{array}{cc} H_{a_1} \\ \theta_{a_1}\end{array} \right] = \left[ \begin{array}{cc} H_{e_1} \\ \theta_{e}\end{array} \right] \times M_{a}

Disregarding \theta_{a_1} since we only care about the height of the entrance pupil we can expand the matrix multiply:

H_{a_1} = H_{e_1} M_{a00} + \theta_{e} M_{a10} (column major)

solving for H_{e_1}:

H_{e_1} = \frac{H_{a_1} - \theta_{e} M_{a10}}{M_{a00}}

Using this equation and plugging in \pm H_{a} we can find tightly fitting bounds for the aperture. These bounds are then projected through the entire optical system (M_s \times M_a) to find the final flare quad position on the sensor. As a final step the positions are projected on the sensor’s image circle, which for 35mm film is \approx 43.3mm diameter, to get the final normalized coordinates and size.

These values represent offets along the vector from the center of the screen to the light position (in my case the sun). I found the screen space position of the sun by projecting a point a far distance from the camera in the direction of the sun although I’m sure there are more accurate ways. The starting angle \theta_e is simply \theta_e = \cos^{-1} (E \cdot L) where E is the forward vector of the camera in world space and L is the direction of the sun. The resulting pseudo code looks like this:

Theta\_e = acosf(dot(E, L));

Axis = Project(CameraPos + L \* 10000.0).Normalize();

H\_a = GetApertureHeight();

foreach ghost

// Aperture projected onto the entrance

H\_e1 = (H\_a - ghost.M\_a[1][0] \* Theta\_e) / ghost.M\_a[0][0];

H\_e2 = (-H\_a - ghost.M\_a[1][0] \* Theta\_e) / ghost.M\_a[0][0];

// Flare quad bounds projected through the system

H\_p1 = ((ghost.M\_s \* ghost.M\_a) \* Vector2(H\_e1, Theta\_e)).x;

H\_p2 = ((ghost.M\_s \* ghost.M\_a) \* Vector2(H\_e2, Theta\_e)).x;

// Project on to image circle

H\_p1 /= 21.65;

H\_p2 /= 21.65;

Center = (H\_p1 + H\_p2) / 2;

Radius = abs(H\_p1 - H\_p2) / 2;

Transform = CreateScale(Radius, Radius \* AspectRatio());

Transform \*= CreateTranslation(Axis \* Center);

RenderFlare(Transform);

# Intensity and Color

Once we have the placement of the ghost we need to determine what the intensity and color should be. I chose to deviate from the papers slightly for both of these values. For the intensity I chose to make the intensity relative to that of the optical system’s effective aperture. I calculate it by determining M_a for the optical system with out introducing any reflections. Then I calculate the projected size of the aperture with \theta_e = 0, since it is symmetrical you only need to do this once:

e = \pm \frac{H_{a}}{M_{a00}}

Then intensity, I, is initialized to the ratio between surface area of the ghost’s effective pupil vs the surface area of the system’s effective pupil. The intensity is then scaled by the final projected area of the flare quad. The final intensity represents the relative intensity compared to an in-focus pixel so you would multiply the light color by it to get your final pre-exposed color.

entrancePupil = H\_a / system.M\_a[0][0];

Intensity = Sqr(H\_e1 - H\_e2) / Sqr(2 \* entrancePupil);

Intensity /= Sqr(2 \* Radius);

[](https://placeholderart.files.wordpress.com/2015/01/flare-intensity.jpg)

Comparison of flare intensity at f/8.0 and f/16.0. The perceived brightness of the flares increase at the higher f-number at the same exposure even though the aperture is smaller. This is represented by the ratio of flare/entrance pupil.

With the intensity calculated the color must be determined. Once again I deviated slightly from the papers to do this. The main contributor to color for a flare is the anti-reflective coating on the lens. These coatings are designed to create an interference pattern and cancel out light that is reflected at each interface similar to noise cancelling headphones. The shift in color occurs because the coatings are designed to minimize reflections at a specific wavelength, generally a mid length wavelength like 500nm, this results in non-uniform reflection across the various wavelengths. I found the best explanation and reference [here](http://www.gen-opt.com/lamda_admin/Lamda_Edit/UploadFile/2011222112937198.pdf), this is the same as is presented in the papers. I deviate from the authors because rather than trace a single ray to determine the color, I chose to evaluate the reflectance at random angle per ghost (once for red, green, and blue). This produces a stable color per flare and makes it easier to ensure that the angle doesn’t exceed the critical angle of the system, resulting in invalid reflectance values.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | // <http://www.gen-opt.com/lamda_admin/Lamda_Edit/UploadFile/2011222112937198.pdf>  // lambda   - Wavelength of light being tested for reflectance  // d        - Thickness of the anti-reflectance coating (lambda0 / 4.0f / n1) where  //              lambda0 is the wavelenght of light the coating was designed for, typically  //              a midrange wavelenght like green (550 nm)  // theta1   - Angle of incidence at the edge of the coating  // n1       - The Index of Refraction for the incoming material  // n2       - The Index of Refraction for the coating, max(sqrt(n1 \* n2), 1.38) where 1.38  //              is the index of refraction for a common magnesium floride coating.  // n3       - The Index of Refraction for the outgoing material  **static** **float** Reflectance(**float** lambda, **float** d, **float** theta1, **float** n1, **float** n2, **float** n3)  {      // Apply Snell's law to get the other angles  **float** theta2 = asinf(n1 \* sinf(theta1) / n2);  **float** theta3 = asinf(n1 \* sinf(theta1) / n3);    **float** cos1 = cosf(theta1);  **float** cos2 = cosf(theta2);  **float** cos3 = cosf(theta3);    **float** beta = (2.0f \* MATH\_PI) / lambda \* n2 \* d \* cos2;        // Compute the fresnel terms for the first and second interfaces for both s and p polarized      // light  **float** r12p = (n2 \* cos1 - n1 \* cos2) / (n2 \* cos1 + n1 \* cos2);  **float** r12p2 = r12p \* r12p;    **float** r23p = (n3 \* cos2 - n2 \* cos3) / (n3 \* cos2 + n2 \* cos3);  **float** r23p2 = r23p \* r23p;    **float** rp = (r12p2 + r23p2 + 2.0f \* r12p \* r23p \* cosf(2.0f \* beta)) /          (1.0f + r12p2 \* r23p2 + 2.0f \* r12p \* r23p \* cosf(2.0f \* beta));    **float** r12s = (n1 \* cos1 - n2 \* cos2) / (n1 \* cos1 + n2 \* cos2);  **float** r12s2 = r12s \* r12s;    **float** r23s = (n2 \* cos2 - n3 \* cos3) / (n2 \* cos2 + n3 \* cos3);  **float** r23s2 = r23s \* r23s;    **float** rs = (r12s2 + r23s2 + 2.0f \* r12s \* r23s \* cosf(2.0f \* beta)) /          (1.0f + r12s2 \* r23s2 + 2.0f \* r12s \* r23s \* cosf(2.0f \* beta));    **return** (rs + rp) \* 0.5f;  } |

Anti-reflection coatings is one of the areas where lens making has made significant strides in the past 20 years. Below is a comparison between a lens from the 80s and a lens from today.

[](https://placeholderart.files.wordpress.com/2015/01/coating-comparison.jpg)

Top: 1980’s Vivitar Zoom  
Boom: 2014 Sony FE 28-70  
One of the major differences between legacy and modern lenses is the quality and characteristics of anti-reflection coatings. Both images were taken with similar settings.

# Flare Texture

Now that we have a color, intensity, and placement for the flare we need to generate a texture. This would be an ideal place to hand it over to an artist to whip something up but I don’t have that luxury at the moment so we will just have to make due. The flare texture should be a projection of the aperture’s shape. On most lenses this would be an n-gon, where n is the number of aperture blades. On higher end lenses a lot of effort goes in to maintaining a circular aperture shape at all f-numbers. The simplest way to do either of these is to generate a small distance field texture, 64×64 in my implementation. If you simply want a circular aperture then the distance is just:

distance = length(In.Texcoord \* 2.0f - 1.0f);

If you want to have an n-gon then you can calculate the distance as the max distance of the coordinate dotted with each axis, where the axis is a vector that is perpendicular to the aperture edge:

distance = 0

coord = In.Texcoord \* 2.0f - 1.0f;

foreach axis

distance = max(distance, dot(axis, coord));

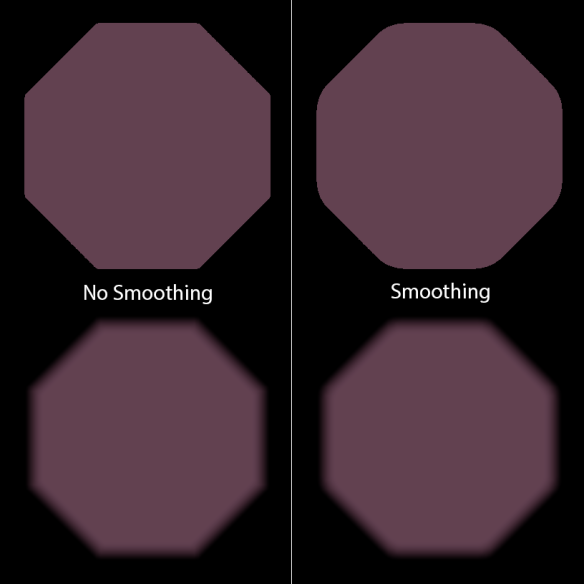
This is the most flexible representation but you could also exploit symmetry to execute less dot products. For example, an octagon can be represented with 1 dot and 2 max operators instead of 8 dot products and 8 max operators.

coord = abs(Texcoord \* 2.0f - 1.0f);

distance = max(coord.x, coord.y);

distance = max(distance, dot(coord, float2(0.70710678118f, 0.70710678118f)));

## Smoothing the result

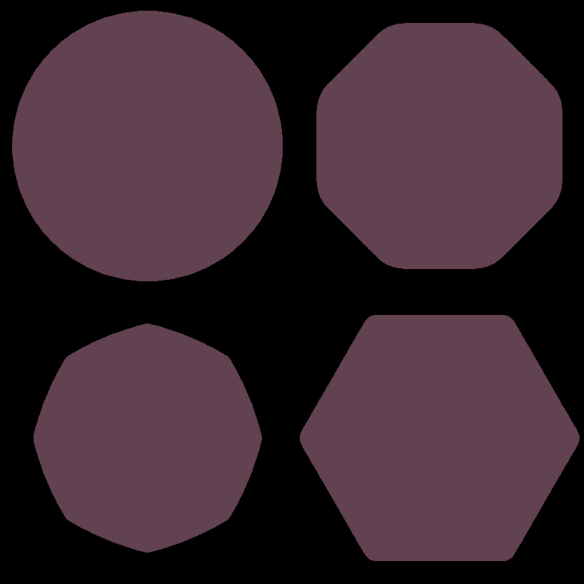


Application of smoothing removes the peaks at the corners in the faded version of the aperture. k = 0.1 for the right hand side.

Directly using this shape will give acceptable results but when applying a falloff based on the distance field some peaks at aperture edges can be seen. To eliminate these I simply applied the smooth max operator, adapted from [Inigo Quilez’s Smooth Min](http://www.iquilezles.org/www/articles/smin/smin.htm) article and the identity max(a,b) = -min(-a, -b).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // polynomial smooth min  // from: <http://www.iquilezles.org/www/articles/smin/smin.htm>  **float** smin(**float** a, **float** b, **float** k)  {  **float** diff = b - a;  **float** h = saturate(0.5 + 0.5 \* diff / k);  **return** b - h \* (diff + k \* (1.0f - h));  }    **float** smax(**float** a, **float** b, **float** k)  {  **float** diff = a - b;  **float** h = saturate(0.5 + 0.5 \* diff / k);  **return** b + h \* (diff + k \* (1.0f - h));  } |

Using this technique you can get lots of different source aperture shapes.



Various aperture shapes obtained using the smoothed distance field approach

# Entrance Clipping

To get the final shape of the flare we need to account for the case when our effective pupil lies outside the physical entrance heights. In this case we should be clipping the aperture shape with the entrance shape. Incorporating this effect is relatively simple and can break up the otherwise repeating effect. Pseudo code follows:

...

H\_e = GetEntranceHeight();

foreach ghost

...

// Entrance projected on to the aperture

E\_a1 = (ghost->\_ma \* Vector2(H\_e, thetaE)).x;

E\_a2 = (ghost->\_ma \* Vector2(-H\_e, thetaE)).x;

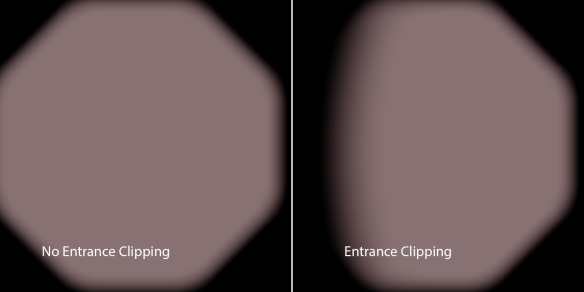
// Move the projection into aperture space as a position and radius

float C\_e = ((e\_a1 + e\_a2) \* 0.5) / H\_a - Center;

float R\_e = (abs(E\_a1 - E\_a2) \* 0.5) / H\_a;

These two parameters are uploaded along with the screen space light direction and are combined with the distance field using the same smooth max approach as before:

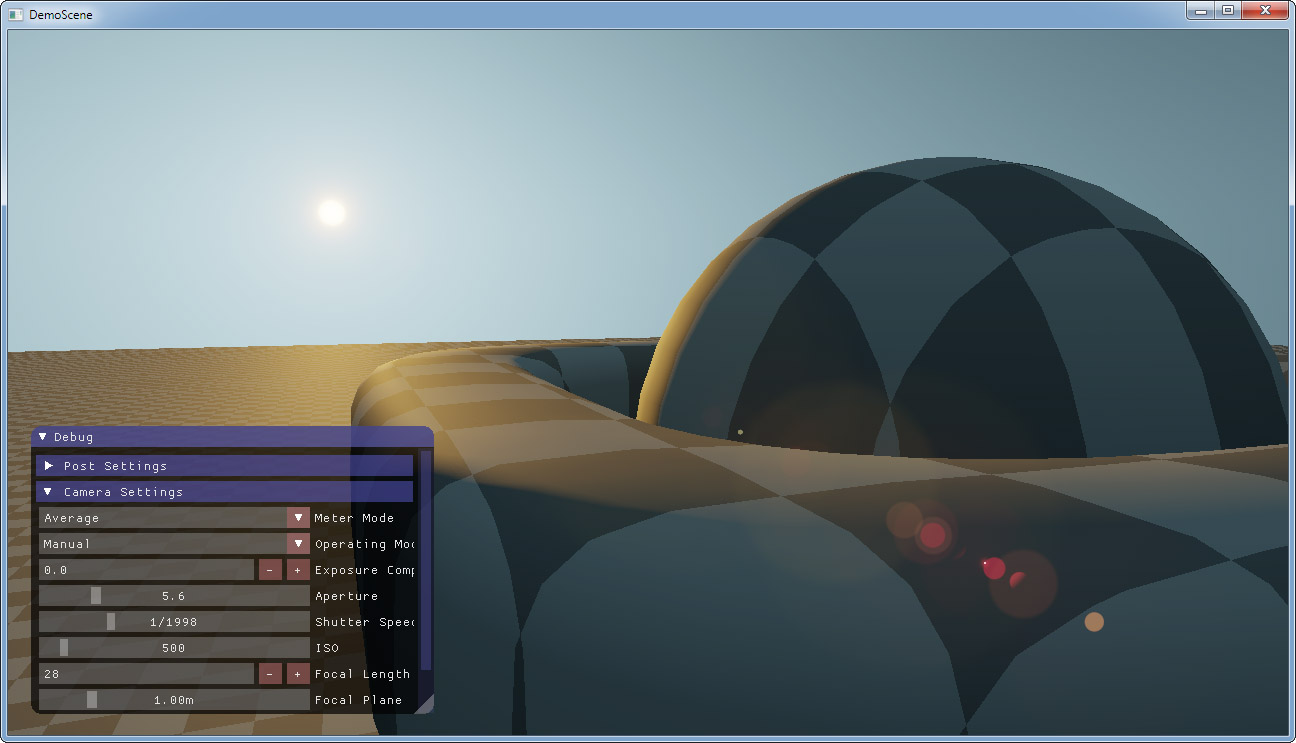
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Sample the distance field  **float** fDistance = FlareTexture.SampleLevel(LinearSampler, Texcoord, 0.0f).x;    // Clip against the entrance in aperture space  float2 coord = Texcoord \* 2.0f - 1.0f;  fDistance = smax(fDistance, length(coord - vEntranceCenterRadius.x \* vViewspaceDirection) / vEntranceCenterRadius.y);    // Finally fade the edge to get the final aperture mask  **float** fApertureMask = smoothstep(0.95, 0.80, fDistance); |

[](https://placeholderart.files.wordpress.com/2015/01/entrance-clipping.png)

Effects of entrance clipping on a single flare.

# Final Results

This brings us to the end of what I decided to implement from the papers. Here are some final results using the info presented in the post.



# Optimizations

The authors present some recommendations for performance such as culling flares whose intensity falls below a certain value or only rendering the top 40% flares. These ideas could be extended with rendering out the flares at multiple resolutions to avoid the fill rate cost of the larger flares that cover a majority of the screen.On the CPU side the system matrices only need to be computed once so there is no need to precompute their values.

# Conclusion

The papers present an interesting method for representing a “Physically Based” lens flare model. If I were to implement it in a production engine I would leave the flare shape and colors as artist configurable and simply implement the flare placement and entrance clipping. The method outlined in this post has it’s limitations since it’s still representing the entire optical system with a linear approximation. Perhaps an interesting approach would be to apply this method to a majority of the flares in a system and then use the more expensive ray-tracing technique proposed by the first paper to represent complex flares. Well, I hope you enjoyed the first installment of Implementation Notes. If you liked it let me know and if you do this type of thing in your own time you should write about it!

1. <http://resources.mpi-inf.mpg.de/lensflareRendering/pdf/flare.pdf>
2. <https://graphics.tudelft.nl/Publications-new/2013/LE13/PracticalReal-TimeLens.pdf>
3. <http://en.wikipedia.org/wiki/F-number>
4. <http://en.wikipedia.org/wiki/Angle_of_view>
5. <http://en.wikipedia.org/wiki/Vignetting>
6. <http://en.wikipedia.org/wiki/Paraxial_approximation>