LiU-ITN-TEK-A--21/068-SE

Physically-based Real-time Glare

Julien Delavennat

2021-12-15

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A--21/068-SE

Physically-based Real-time Glare

The thesis work carried out in Advanced Computer Graphics
at Tekniska högskolan at
Linköpings universitet

Julien Delavennat

Norrköping 2021-12-15

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

Linköping University | Department of Science and Technology | Master's thesis, 30 ECTS | Advanced Computer Graphics 2021 | LIU-ITN/LITH-EX-A--21/068-SE

Physically-based Real-time Glare

Julien Delavennat

Supervisor : Mark E Dieckmann
Examiner : Jonas Unger

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

The theme of this master's thesis is the real-time rendering of glare as seen through human eyes, as a post-processing effect applied to a first-person view in a 3D application. Several techniques already exist, and the basis for this project is a paper from 2009 titled Temporal Glare: Real-Time Dynamic Simulation of the Scattering in the Human Eye, by Ritschel et al.. The goal of my project was initially to implement that paper as part of a larger project, but it turned out that there might be some opportunities to build upon aspects of the techniques described in Temporal Glare; in consequence these various opportunities have been explored and constitute the main substance of this project.

Acknowledgments

I'd like to thank Mark E Dieckmann and Jonas Unger, firstly for supervising my project, but also for the various bits of advice they've given me on how to organize my work, how to scope the project, on some useful theory points, as well as for valuable career insights. I extend similar thanks to Camilla Forsell for helping with my project registration, and also to the rest of the Advanced Computer Graphics teaching staff at LiU. I also want to thank my family for supporting me during my studies, and my dear Naelan for their incredible moral support during the writing of this report.

Contents

1 Introduction

1.1 Motivation

Most first-person video games render lens flare (see figure 1.1) to make bright highlights and light-sources appear brighter, but this is inconsistent with the fact you play as a human in a lot of them: if the goal is to create immersion, then maybe rendering glare as it looks through our own biological eyes would make more sense (see figure 1.2).

Figure 1.1: In *Battlefield 4* (DICE, 2013), bright lights are represented with anamorphic lens flare and not ocular glare.

Figure 1.2: Glare effect simulation by Ritschel et al. [1].

1.2 Aim

The main goal is to look at existing research to try implementing physically-based real-time glare rendering in a game engine.

1.3 Research questions

1. Can physically-based glare rendering be implemented in a game engine?
2. Can it be flexible enough to handle common use cases?
3. Can it render fast enough to be used in real-time?

1.4 Delimitations

If there is a tradeoff to make between performance and quality, performance will have priority, as long as a minimum satisfactory level of quality is met. For instance, effect resolution might need to be capped for performance reasons.

2 Theory

2.1 Glare

Figure 2.1 shows an example of what glare might look like:

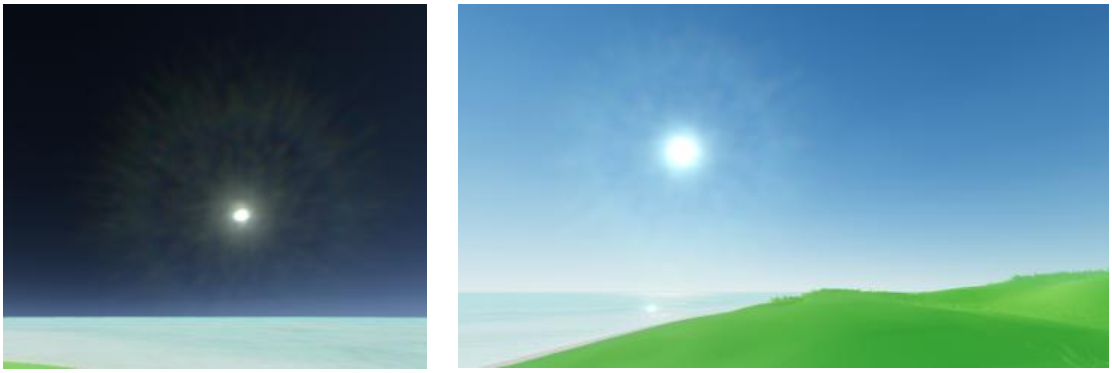


Figure 2.1: Examples of glare around a light source at night and during the day.

Glare happens when looking at a bright point or area. It is due to scattering and diffraction of light by anatomical components in the eye. Ritschel et al. [1] did a survey of relevant anatomical components, which singled out 4 significant glare components: the bloom, the ciliary corona, the lenticular halo, and eyelash streaks, shown in figure 2.2.

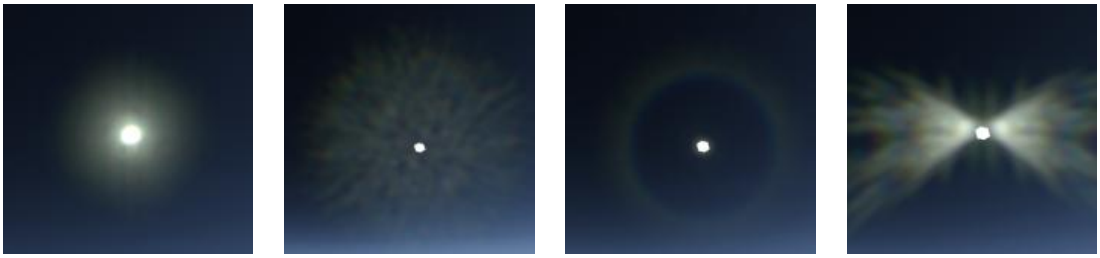


Figure 2.2: From left to right: bloom, ciliary corona, lenticular halo, eyelash streaks.

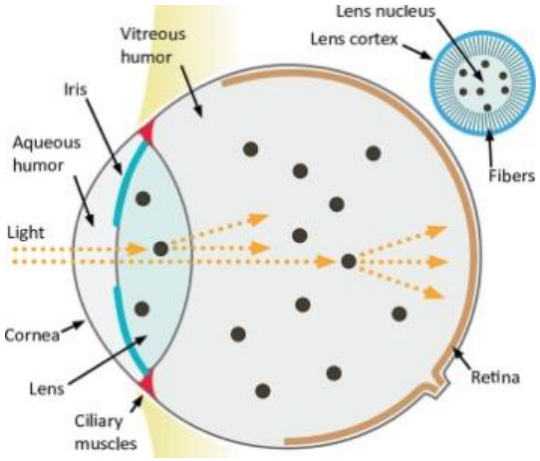


Figure 2.3: Anatomy of the human eye; in the upper right corner we have a front view of the lens [1].

Each of these glare components is caused by a corresponding anatomical component.

- Bloom is caused by the pupil, which is the aperture of the eye. A property of round apertures is they prevent the optical system from having perfect focus, due to diffraction. Even with no imperfections, the system is said to be "diffraction-limited". Circular apertures generate diffraction patterns known as Airy disks, which then appear around bright points. We see light from those points spread to relatively darker neighboring points, which blurs the image locally.
- The ciliary corona is caused by particles scattering light inside the eye, which creates a needle "crown" pattern [1]. Babinet's principle states that a small opaque obstacle that scatters waves will cause an interference pattern similar to what its inverted shape (i.e. a small opening of the same shape and size) would cause, except the phase would be reversed. We can use diffraction equations as with the other glare sources.
- The lenticular halo is caused by the edges of the lens, which have radially-arranged gratings that cause diffraction of incoming light [2]. See upper-right corner of figure 2.3.
- Streaks are caused by eyelashes scattering and diffracting light [1]. See figure 2.4.

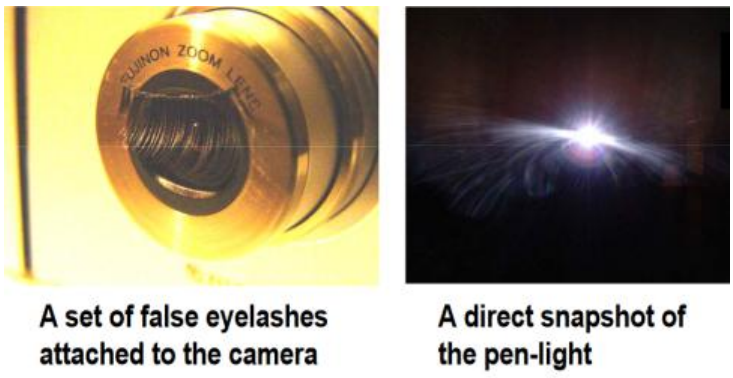


Figure 2.4: An artificial reproduction of eyelash streaks [3].

2.2 Diffraction

To simulate the effect of diffraction in the eye, Kakimoto et al. [3] and Ritschel et al. [1] suggest taking the Fourier transform of an aperture image which serves as a simplified input aperture plane for the diffraction equation (instead of accurately representing the entire 3D interior space of the eye), and then layering copies of the resulting pattern, each with a different color and with UVs scaled according to wavelength, to create the color dispersion effect of the diffraction.

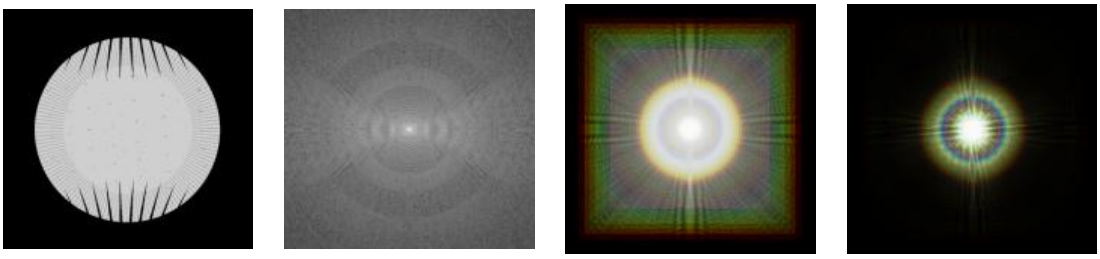


Figure 2.5: Left to right: 1) aperture image, modeled by approximating anatomy components with 2D shapes, made up of the pupil as a white background, the gratings on the edges of the lens as grey lines, the particles, and the eyelashes at the top and bottom, 2) its Fourier transform, 3) layered copies of a Fourier transform pattern with changing UV scale for each colored layer, 4) result with tone-mapping added.

Monochromatic and multi-color diffraction

To reach the point where we use Fourier transforms to calculate diffraction interference patterns, we have to look at how diffraction is modeled mathematically.

Diffraction is a phenomenon by which waves change shape as they cross an aperture gap. It applies to mechanical waves such as on the surface of water, but also to electromagnetic waves such as light.

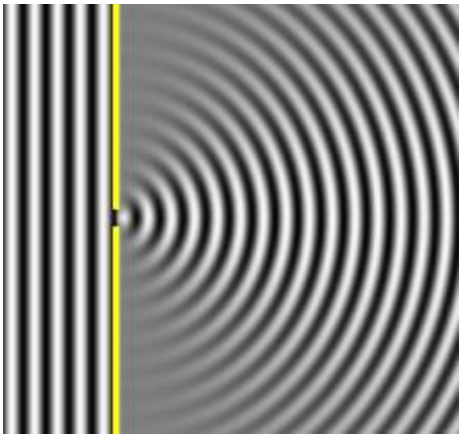


Figure 2.6: If aperture width is equal to wavelength, waves take a circular shape.

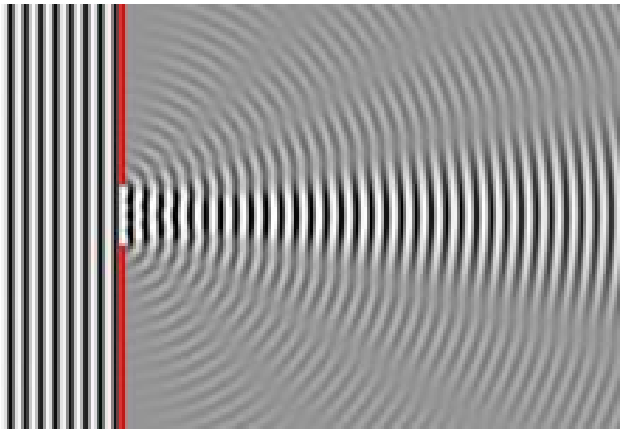


Figure 2.7: If aperture width is different to wavelength, waves form a more complex interference pattern.

Observing glare interference patterns will be done at some sort of receiving surface - in the case of the eye, it will be the retina, but we can approximate it with a plane (cf figure 2.8).

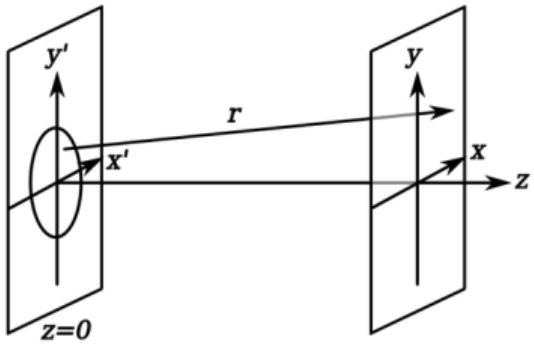


Figure 2.8: Waves pass through the circular aperture in the x'y' plane, travel a distance r, and get measured at the xy plane at a distance z from the aperture.

There exist several different equations that model this phenomenon. One of them is the equation of the Huygens-Fresnel principle. This principle states that points of a wavefront can be considered to also emit their own waves, and those new waves interfere with each other; the result of this interference is the wavefront at the next time-step.

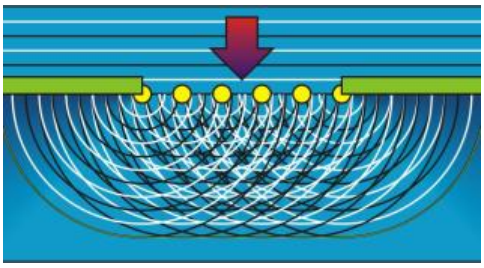


Figure 2.9: Points of the wavefront generate circular wavelets; the sum of all those wavelets interfering with each other forms the new wavefront.

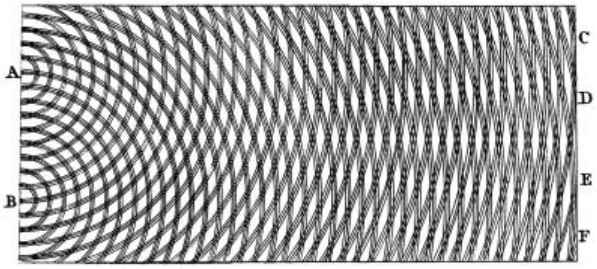


Figure 2.10: The resulting wavefront becomes more circular (with a lower curvature) once far enough from the aperture.

The Huygens-Fresnel principle is described by the following formula [1] (terms on next page):

$$E\left(x,y,z\right) = \frac{1}{i\lambda}\iint_{-\infty}^{+\infty} E\left(x',y',0\right)\frac{e^{ikr}}{r}\frac{z}{r}dx'dy' \tag{2.1}$$

It outputs the distribution of complex amplitudes [2] for wave points in the space after the aperture, based on the complex amplitudes of the points of the incoming wavefront *at* the aperture, and a given wavelength.

Note: Euler's formula states that $e^{ix} = \cos x + i\sin x$.

[1] From "Fresnel Diffraction." In Wikipedia (September 27, 2021). `https://en.wikipedia.org/w/index.php?title=Fresnel_diffraction&oldid=1046835387`.

[2] A complex amplitude value is actually two values: amplitude and phase, stored respectively in the real and imaginary parts of a complex number, which we call a phasor. Amplitude only tells you the highest and lowest values possible for points of a wave, not the specific value for individual points on the wave; to sample a sine curve, you also need phase, to know at what point in its cycle you're sampling. This is why the phase is needed.

Terms:

- $E(x, y, z)$ describes complex amplitude at a point (x, y, z). (See note[3])
- λ is a wavelength
- i is the imaginary number unit
- (x, y, z) is the point after the aperture we're computing a value for
- $(x', y', 0)$ is a point at the aperture we're integrating over
- $k = 2\pi/\lambda$
- r is the distance between (x, y, z) and $(x', y', 0)$, given by the expected pythagorean formula $r = \sqrt{(x - x')^2 + (y - y')^2 + (z - 0)^2}$

From the resulting complex amplitudes, we can calculate radiance using the relationship:

$$intensity(x, y, z) = |complex\_amplitude(x, y, z)|^2 \tag{2.2}$$

The equation for the Huygens-Fresnel principle can be made more practical to work with analytically [4][5], and so for convenience purposes, it is common to instead work with approximations of that equation. Technically, if our work is done numerically, the Huygens-Fresnel equation should be possible to compute directly without dealing with analytical approximations, however, the fact one of the approximations lets us represent the calculation as an FFT can be quite practical too in the case we have a performant off-the-shelf FFT2D implementation available, since that can simplify implementation without costing significant accuracy.

There are two common approximations: the Fraunhofer approximation, which was used by Kakimoto et al. [3] for example, and the Fresnel approximation, used by Ritschel et al. [1].

The Fresnel diffraction equation is an approximation used to calculate a diffraction pattern viewed from an area relatively close to the aperture, called the near-field. In contrast the Fraunhofer diffraction equation gives us the diffraction pattern in the far field. The Fresnel approximation is used for near-field calculations because it is more accurate, since near-field calculations require that extra accuracy, and Fraunhofer diffraction is actually a special case with less demanding validity conditions.

The equation for the Fresnel approximation is the following, with the same terms as above:

$$E(x, y, z) = \frac{e^{ikz}}{i\lambda z} \iint_{-\infty}^{+\infty} E\left(x', y', 0\right) e^{\frac{ik}{2z}\left[(x-x')^2+(y-y')^2\right]} dx'dy' \tag{2.3}$$

The derivation of the Fresnel approximation and its validity conditions are given in a dedicated appendix section.

[3] We can see how representing the whole aperture as one image plane can make it fit conveniently into the input E(x', y', 0) term of the diffraction equation. However, since diffraction depends on depth/distance, representing the whole interior space as a flat plane is going to be incorrect because it doesn't account for particles that can be found all along the way from the aperture to the retina properly. Ritschel et al. [1] compare single-plane approximation with multiple-plane approximation, and conclude the image difference between the two is fairly minor, while the performance cost of multiple-plane is high. They still simulate the movement of the particles as multiple planes or layers, but the aperture image that gets diffracted is just one plane.

As mentioned in a previous section, this equation can be computed using a Fourier transform, but first the terms of the Fresnel Approximation need rearranging:

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{ik\frac{\left(x-x'\right)^2+\left(y-y'\right)^2}{2z}}dx'dy' \tag{2.4}$$

We can expand the $(x-x')^2$ and $(y-y')^2$ terms:

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{ik\frac{\left(x^2+x'^2-2xx'\right)+\left(y^2+y'^2-2yy'\right)}{2z}}dx'dy' \tag{2.5}$$

We can rearrange, and then split the highlighted part,

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{\frac{ik}{2z}\left(x^2+x'^2-2xx'+y^2+y'^2-2yy'\right)}dx'dy' \tag{2.6}$$

into several exponentials:

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{\frac{ik}{2z}\left(x^2+y^2\right)}e^{\frac{ik}{2z}\left(x'^2+y'^2\right)}e^{\frac{ik}{2z}\left(-2xx'-2yy'\right)}dx'dy' \tag{2.7}$$

We can rewrite k as $\frac{2\pi}{\lambda}$, and rewrite the last term to look like the one we'd find in a 2D Fourier transform:

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{\frac{i\pi}{\lambda z}\left(x^2+y^2\right)}e^{\frac{i\pi}{\lambda z}\left(x'^2+y'^2\right)}e^{\frac{i\pi}{\lambda z}\left(-2xx'-2yy'\right)}dx'dy' \tag{2.8}$$

We can move the exponential with x^2+y^2 in front of the integral:

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{\frac{i\pi}{\lambda z}\left(x^2+y^2\right)}e^{\frac{i\pi}{\lambda z}\left(x'^2+y'^2\right)}e^{-i2\pi\left(\frac{x}{\lambda z}x'+\frac{y}{\lambda z}y'\right)}dx'dy' \tag{2.9}$$

$$E\left(x,y,z\right)=\frac{e^{ikz}}{i\lambda z}e^{\frac{i\pi}{\lambda z}\left(x^2+y^2\right)}\iint_{-\infty}^{+\infty}E\left(x',y',0\right)e^{\frac{i\pi}{\lambda z}\left(x'^2+y'^2\right)}e^{-i2\pi\left(\frac{x}{\lambda z}x'+\frac{y}{\lambda z}y'\right)}dx'dy' \tag{2.10}$$

What we have obtained is the Fourier transform of $[E(x',y',0)e^{\frac{i\pi}{\lambda z}(x'^2+y'^2)}]$, with frequency inputs $x/\lambda z$ and $y/\lambda z$, and a couple terms in front of the integral relating to phase.

For our purposes, parts of those terms relating to phase can be removed [5] because we are interested in radiance, and phase doesn't impact total energy -radiance-:

$$E(x,y,z)=\frac{e^{ikz}}{i\lambda z}e^{i\frac{\pi}{\lambda z}\left(x^2+y^2\right)}\mathcal{F}\left\{E(x',y',0)e^{i\frac{\pi}{\lambda z}\left(x'^2+y'^2\right)}\right\}\Big|_{p=\frac{x}{\lambda z},\,q=\frac{y}{\lambda z}} \tag{2.11}$$

$$E(x,y,z)=\frac{1}{i\lambda z}\mathcal{F}\left\{E(x',y',0)e^{i\frac{\pi}{\lambda z}\left(x'^2+y'^2\right)}\right\}\Big|_{p=\frac{x}{\lambda z},\,q=\frac{y}{\lambda z}} \tag{2.12}$$

After computing equation 2.12, we use equation 2.2 to get radiance for glare pattern points.

2.3 Multi-color diffraction

What Kakimoto et al. [3] and Ritschel et al. [1] propose to actually run this, is have the aperture image corresponding to the light passing through the aperture (explained later in section 2.4), to multiply it by the complex exponential $e^{\frac{i\pi}{\lambda z}(x'^2+y'^2)}$, to run it through an FFT2D, and for each wavelength to:

- scale image coordinates - texture UVs in practice - based on wavelength ratios; we can use $\lambda_1 = 575nm$ as the reference wavelength which has scale 1, and then scale the pattern UVs for other wavelengths λ_2 by $\frac{\lambda_2}{\lambda_1}$ [1],
- layer the resulting pattern with the patterns for the other wavelengths

and then normalize the resulting values so they fit in [0;1], since for example, layering 32 images with additive blending can give us values up to 32, which would clip. Stacking scaled and tinted layers like this is correct when using Fraunhofer diffraction, but with Fresnel diffraction, we should actually recompute the FFT for each wavelength. This is because wavelength impacts the interference pattern, due to the presence of the lambda term in the complex exponential in the input of the FFT.

To generate the rgb value that corresponds to the wavelength for each layer, we might need a system that takes a wavelength in nanometers as input, and outputs an rgb value. The XYZ color space is designed to serve as an intermediary for that purpose. There are correspondence tables, such as the CIE 1931 2° standard observer table, which gives XYZ values for a given wavelength. It can be found freely online. The values might looks like this:

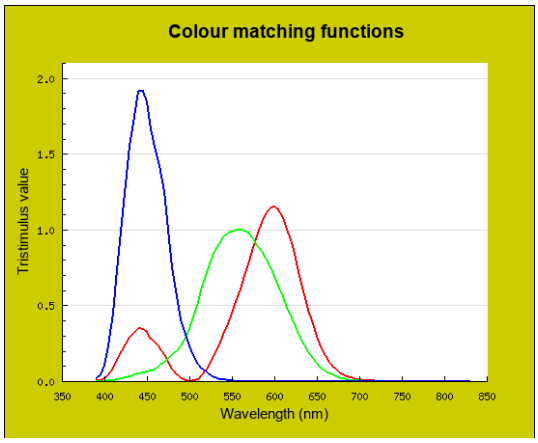


Figure 2.11: CIE 2° standard observer XYZ color-matching function curves.

Once we have XYZ values, we need to convert them to sRGB. Converting sRGB to XYZ involves matrix multiplication.

$$\begin{bmatrix} R_{\text{linear}} \\ G_{\text{linear}} \\ B_{\text{linear}} \end{bmatrix} = \begin{bmatrix} +3.2406 & -1.5372 & -0.4986 \\ -0.9689 & +1.8758 & +0.0415 \\ +0.0557 & -0.2040 & +1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Then, the following formula applies the missing gamma correction to get sRGB values.

$$C_{\text{srgb}} = \begin{cases} 12.92C_{\text{linear}}, & C_{\text{linear}} \leqslant 0.0031308 \\ (1.055)C_{\text{linear}}^{1/2.4} - 0.055, & C_{\text{linear}} > 0.0031308 \end{cases}$$

where C is R, G or B. These values are finally clamped to the range [0;1].

2.4 Modeling and animation

The four relevant anatomical components can be modeled as shown[4] in figure 2.12. Research groups seem to mostly design these manually, even if it's informed by actual medical research into anatomy. These components are fairly straightforward to shape, but Ritschel et al. recommend modeling the images using supersampling to smooth out some aliasing before feeding them to the FFT, since aliasing can cause unwanted artifacts in the frequency domain. Then, once we want to simulate their movements over time, various challenges appear. Exploring those challenges makes up the main substance of the Temporal Glare paper [1].

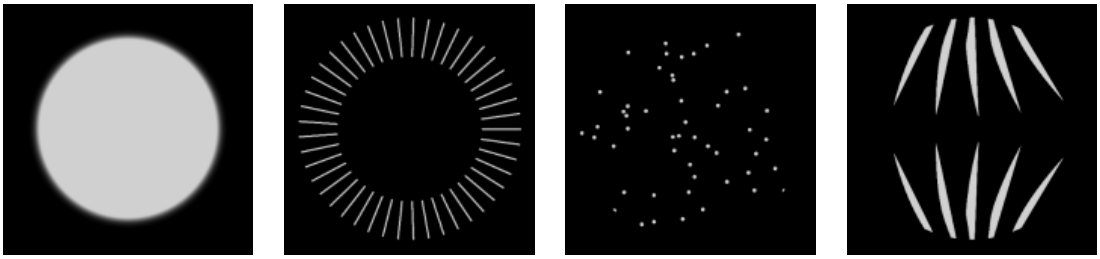


Figure 2.12: From left to right: pupil aperture, gratings in the lens, particles in the lens, vitreous humor and cornea, and then eyelashes.

Pupil

The pupil is the round opening of the eye. Its modeling is quite straightforward, but it has some nuance in its movement. The pupil is affected by the *pupillary hippus*: it contracts and expands over time depending on average scene luminance. Ritschel et al. [1] give equations 2.13 and 2.14 to describe the pupil diameter following those parameters.

$$h(t,p) = p + noise(\frac{t}{p})\frac{p_{max}}{p}\sqrt{1-\frac{p}{p_{max}}} \tag{2.13}$$

$$p = 4.9 - 3\tanh(0.4(\log L + 1)) \tag{2.14}$$

where $h()$ is the hippus function that gives the pupil diameter in millimeters, t is time in seconds, p is the mean pupil diameter, p_{max} is the maximum pupil diameter when expanded (they use 9mm), L is the scene luminance approximated by time-damped average screen intensity.

Lens gratings

The lens gratings are transparent and radially-arranged and expand from the center of the lens and up to its edges, but they have a higher refractive index near the edges, which is why there is more diffraction happening there. Ritschel et al. use 200 gratings, as a point of reference. The lens expands and contracts in a similar way to the pupil, and that motion is in fact dependent on the pupil's diameter, while still having its own fluctuations. Temporal Glare has the details [1]. To summarize quickly: those fluctuations include a low-frequency (<0.6Hz) noise-like variation and a small higher-frequency variation (between 1.3Hz and 2.1Hz), and the low-frequency one is stronger when the pupil diameter is larger.

[4] the last three components are computed with inverted color to make rendering them on black backgrounds separately possible; that's actually one of the techniques built for this thesis project, which is explained in the Method chapter.

Particles

Ritschel et al. [1] bring attention to two relevant types of particles that move inside the eye, those in the lens, and those in the vitreous humor, and one type of particle that doesn't move, in the cornea. They also model the contribution of the retina to the overall scattering by making those 3 other types of particles larger and less numerous than what physical measurements would recommend. The particles in the cornea are modeled as somewhat large, static and sparsely distributed. The particles in the lens are larger and more numerous than the ones in the vitreous humor. Particles have a somewhat uniform density distribution, while still randomly placed, for both the lens and the vitreous humor. They use 750 particles in the lens, as a point of reference.

To simulate the movements of the particles in the lens and vitreous, Ritschel et al. [1] use rigidbody and spring simulations that react to eye accommodation and head movements, respectively. For performance reasons, these options have not been explored for this project. To summarize quickly: particles in the lens follow its contractions and expansions, and particles in the vitreous humor all move together following rotational head momentum, spinning the particle volume around its center.

Eyelashes

Nakamae et al. [6] give some ranges to describe eyelashes: between 4 and 8 eyelashes in front of our pupil, which, due to diffraction, generate between 6 and 20 visible light streaks; eyelashes grow at random angles with an average angle of 0° (i.e. perpendicular to the eyelid), and with a standard deviation in the range of 10-30°. Eyelashes might be more relevant from the upper eyelid, since eyelashes of the lower eyelid tend to angle down out of view.

We might simulate blinking by moving the image of the eyelids and eyelashes to close them, before feeding the image to the diffraction computation. There are possible use cases for blinking in first-person view, but there's also an argument to be made against having artificial blinking, since the users looking at the screen already have their own blinking happening in real life, and it might be jarring to have our virtual perspective cut for a couple frames every couple seconds. However, we might still display only the eyelash streaks without actually blocking the view, to give at least the feeling of blinking.

Similar to blinking, we can simulate squinting. This wouldn't block the view so it might be relevant for the goal of simulating our first-person virtual eyes reacting to strong lighting. A squinting animation might be combined with an actual lowering of the scene brightness, since that's what squinting is for in practice.

Another type of motion relating to eyelash streaks is how the streak pattern changes based on where a glare source is located in our field of view - in practice, in screen space. If we move our head, or if the glare source moves, the relative position changes, and it creates a scrolling effect. Kakimoto et al. [3] suggest precomputing a glare pattern for each possible viewing direction to account for the scrolling of the streak pattern, but acknowledge the necessary memory cost would be extremely high. That cost doesn't even include the fact that in practice, eyelash streaks can sometimes occupy the entire width of our field of view, and so those textures might need to be very large. This would also cause a high computation cost, since we'll be interested in compositing glare using convolution and our kernel would be the size of the viewport.

2.5 Compositing

Shinya et al. [7] describe how we can place glare highlights on bright pixels using a convolution, cf figure 2.13. First we might need to somewhat differentiate bright scene pixels from the others, otherwise everything might get blurred because anything that isn't completely dark would generate glare, cf figure 2.14. We can use a tone mapping operation that scales scene brightness before the convolution, to describe how much glare should be generated for each pixel. Glare amount can then be zero for pixel values up to a certain minimum brightness threshold, or at least very low so it's not noticeable, cf figure 2.15. The result of the convolution of the glare image with the scene view is then layered on top of the original scene view. Other ways of compositing have included placing billboards over bright pixels, which works well in some cases, but also suffers from potentially needing many billboards with alpha-blending (i.e. expensive rendering) if the scene includes many glare sources. Convolution over the scene image has a predictable computation cost, and also offers the advantage of natively placing the glare on the right pixels without help from another subsystem, specifically in a way where the glare pattern conforms to the shape of the glare source. In Temporal Glare [1], we have an example with a candle light, where the convolution smoothes the horizontal needles of the glare's ciliary corona, but keep the vertical needles, since it follows the vertical shape of the candle flame.

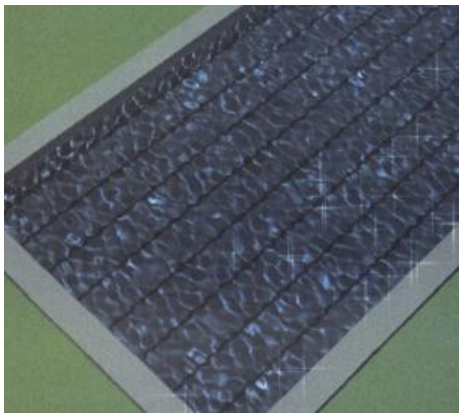


Figure 2.13: Cross-shaped sparkles are composited onto bright specular reflections via convolution, cf right side of the image [7]

Figure 2.14: Here, every pixel in this image has bloom, because even somewhat dark pixels are generating glare.

Figure 2.15: With the tone-mapping applied, we can describe how much each pixel should generate glare: only bright spots and light sources will contribute glare.

Fourier-domain convolution

We can either perform a typical convolution by running a kernel over the scene view image, or we can perform a convolution in the Fourier domain. Doing it in the Fourier domain is actually straightforward: it's a per-pixel complex number multiplication between the 2 images in the Fourier domain. Take 2 equal-resolution images, multiply each pixel of image 1 with the same pixel in image 2, while remembering FFT pixels are complex numbers, so we have to do a complex number multiplication, and we're done.

The main consideration for the choice of whether to use the regular convolution or the Fourier-domain convolution is performance of the implementation. The computation time of the Fourier version doesn't increase as much with the size of input images as the computation time of the brute-force regular convolution, but it costs a forward and an inverse Fourier transform of the scene view every frame in real time. Parallelization can offer some answers, but it's very implementation-dependent.

Now, the Fourier convolution might have one unintended effect: wrap-around. i.e. if a piece of one image lands on the edge of the other, it'll appear on both sides of it.

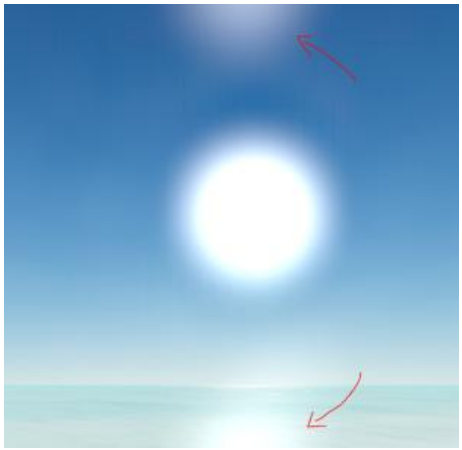


Figure 2.16: The sun in the middle of the image generates bloom around itself as expected, but a reflection on the water is causing bloom that wraps around to the top of the image.

What we can do to avoid this is to add "zero-padding", i.e. empty black pixels to the edges of the scene image: typically that quadruples the pixel count of the image.

Figure 2.17: We can double the image size and fill the edges with black pixels to let the excess bloom overflow into that buffer zone, and then crop to the original image size afterwards to remove the wrap-around.

HDR Tone-mapping

In order to be able to smoothly scale glare intensity based on scene radiance values, we might benefit from using HDR tone-mapping. HDR means "high" dynamic range, and LDR means "low" dynamic range. What is dynamic range? In the context of computer graphics, it's the number of different brightness levels we can encode, either on-screen, or in textures, etc.. It's typically how many levels our display can handle: for instance most screens have 256 levels (per color channel), while the original Gameboy had 4 levels of grey. What counts as low and what counts as high actually depends on context, but in general we'll consider 256 levels to be LDR. Because we might want to simulate the eye's adaptation to scene brightness, we have to keep a representation of the objective amount of light within our scene, and a separate representation of how bright we perceive that light. The objective difference between the darkest areas of a scene and the brightest areas could be much larger than a ratio of 1:255, and we have to decide how those areas will be displayed. We can let pixels with values lower than 1 and higher than 255 get clipped and lose the nuances outside of that range, or instead we can preserve those nuances, by using a tone-mapping curve to bring back all the brightness values in the scene within the range we can actually display. cf figure 2.18.

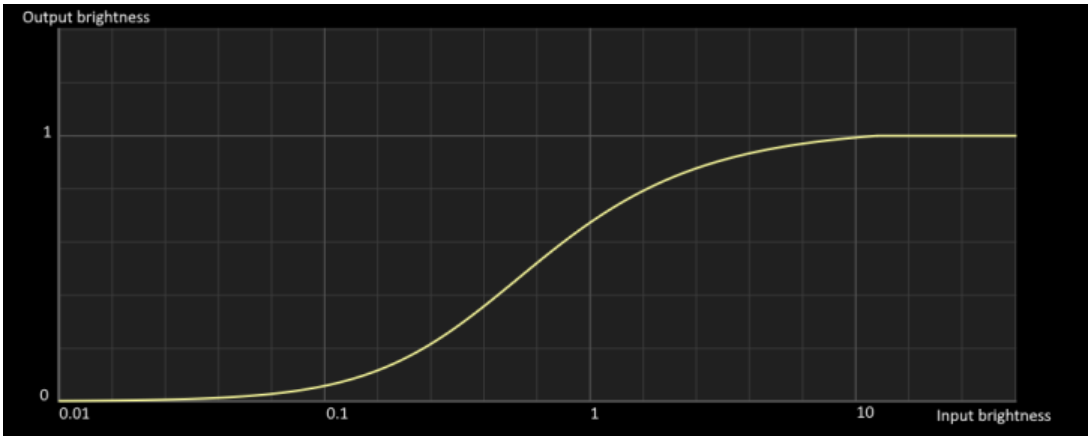


Figure 2.18: Tone-mapping, such as the ACES transform, reserves extra range for very dark and very bright values to avoid clipping.

HDR color blending

Beyond brightness and monochromatic compositing, another important aspect of the compositing is color. We want to be able to adapt glare color to source color. The way it's typically handled, is by splitting the multicolor glare pattern into its constituent rgb channels, and then compositing the red pattern onto the red channel of the scene view, green onto green, blue onto blue. A more involved system might look at breaking down the visible light spectrum into more channels corresponding to more frequency bands, but having 3 bands for rgb is fairly standard, especially in game engines. Now, since we're using HDR brightness levels, if we use naive RGB color, hue would behave incorrectly at high brightness levels. Hues tend to converge towards yellow, magenta and cyan when RGB color brightness is increased linearly by a high amount. For example, (255,1,0), which is basically red with a single bit of green, if multiplied by 1000, clips as (255,255,0), which is yellow, cf figure 2.20. The ACES color transform aims to preserve hue when brightness is very high (cf figure 2.21), as well as tonemap greyscale values to avoid clipping, cf figure 2.23.

Figure 2.19: Initial RGB colors.

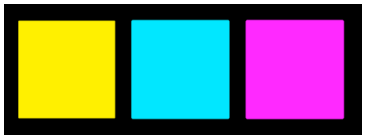


Figure 2.20: Same RGB colors intensified naively, which turns them yellow, cyan and magenta.

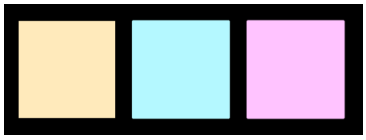


Figure 2.21: Colors intensified using the ACES color transform: hue is more accurate, and saturation decreases.

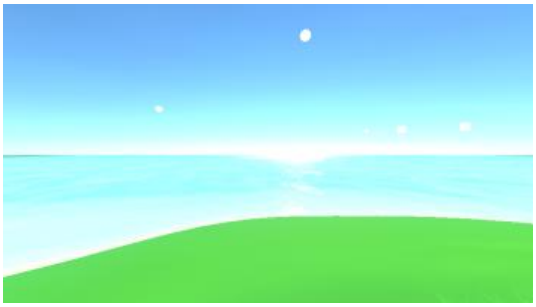


Figure 2.22: Without tone-mapping: lots of pure white and cyan.

Figure 2.23: With ACES tone-mapping: less clipping.

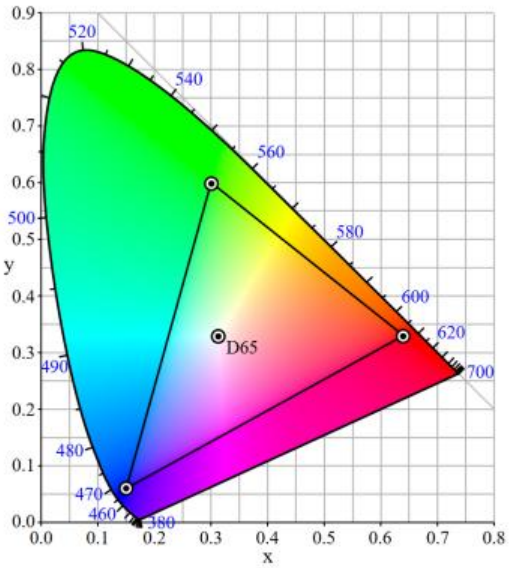


Figure 2.24: The black triangle delimits the sRGB color space.

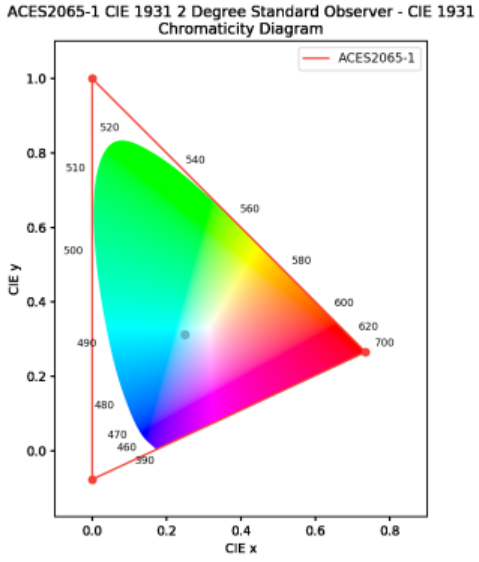


Figure 2.25: The outer red triangle delimits the ACES color space.

3 Method

This chapter explains the approach taken for answering the project's research questions. The main need was to test glare simulating and rendering techniques by implementing them, to see what could be done with them.

Initially, the work started in 2014 on a larger project about sky rendering. That initial project aimed to explore techniques for atmospheric scattering, day-night cycles, but also -more relevantly- sun rendering. The idea for rendering the sun, was to implement the technique presented in Temporal Glare [1], but it became obvious fairly quickly that the glare effect could constitute its own entire project. Since that initial project had to be dropped, I decided to focus on glare rendering when starting our current project in 2019.

3.1 First research question: glare rendering in a game engine

The initial goal was to test implementing glare rendering following instructions from the Temporal Glare paper [1] in a concrete use case; and looking at the techniques described in Temporal Glare, looking at its test cases, as well as previous works it referenced, it wasn't immediately obvious whether it could handle progressive changes in brightness, color, if it could render different types of glare sources on screen at the same time, or run at truly interactive framerates.

This meant the first research question revolved around the viability of glare rendering for real-time applications.

To look into this issue, the first step was to re-implement a simplified version of the technique from Temporal Glare [1]; a lot of progress had been made in that direction in the original 2014 project -written in C++ and GLSL- but for the current one, work was done inside Unity3D for the sake of convenience, so code had to be re-written (in C# and HLSL). The original project was cut short around the time I was working on acquiring an FFT implementation that could run on GPU to satisfy the performance needs of the technique, so that's where I started this time around.

Fourier Transform Compute Shader

The FFT implementation used for this project is a Cooley–Tukey radix2 FFT, with a theoretical complexity of 5N.log(2N). It's implemented as a compute shader (cf appendix for an explanation of compute shaders). I run the math operations on all three rgb channels of the target textures at once. The implementation works in place and overwrites the input textures (the real part and imaginary part of the data). If the input is a typical spatial-domain texture, the imaginary part is just a black texture initially, and it gets filled with the imaginary data by the FFT. The same compute shader can also perform the inverse FFT by setting a flag, by taking in both a real and an imaginary part as input. It only runs on power-of-2 resolutions, typically 512p in our general case. The texture buffers used are float4 read-write HDR linear textures. I mentioned they were 512p, this is half the resolution of the 1820x1024p viewport's vertical resolution, for performance reasons. For cases where the image has padding, that means the effective image resolution that goes into the FFT is halved down to 256p (the padding implementation is explained later). The 1820*1024 viewport resolution was chosen in order to have an aspect ratio of 16:9, while having a vertical resolution that was a power of 2, in order to avoid stretching images to and from 1920*1080, which would've been a more common resolution.

Since the implementation of the 2D FFT algorithm can be represented as a series of 1D FFTs for rows of the image pixels, then 1D FFTs for the columns, I've split the compute shader in two passes: one that runs a kernel with thread groups of 32 rows per dispatch, one thread per row, and then another one that does the same for the columns. The dispatches run enough thread groups at once to cover the image resolution, so each dispatch is 32x16 == 512 threads per pass. The normalization of the values is done inside the kernels, once at the end of the row pass, and once at the end of the column pass. The FFT implementation can only handle square textures, so for example for a viewport resolution of 1820*1024, the viewport image texture can be squeezed horizontally to 1024x1024 before being fed to the FFT step (for performance reasons, the scene image is actually treated in half-resolution, so squeezed from 910*512 to 512*512). The threadgroup size of 32 has been found empirically, and gave better performance than groups of 8, 16, 64, 128. This lets the FFT be executed somewhat in parallel, and gives a very decent performance boost compared to a non-parallel one: an initial test ran 1 group of 1024 threads instead of 1024 groups of 1 thread (for both the row and column passes), and performance increased 4x; I then tried other variants, and settled for groups of 32 threads which was another 30% faster.

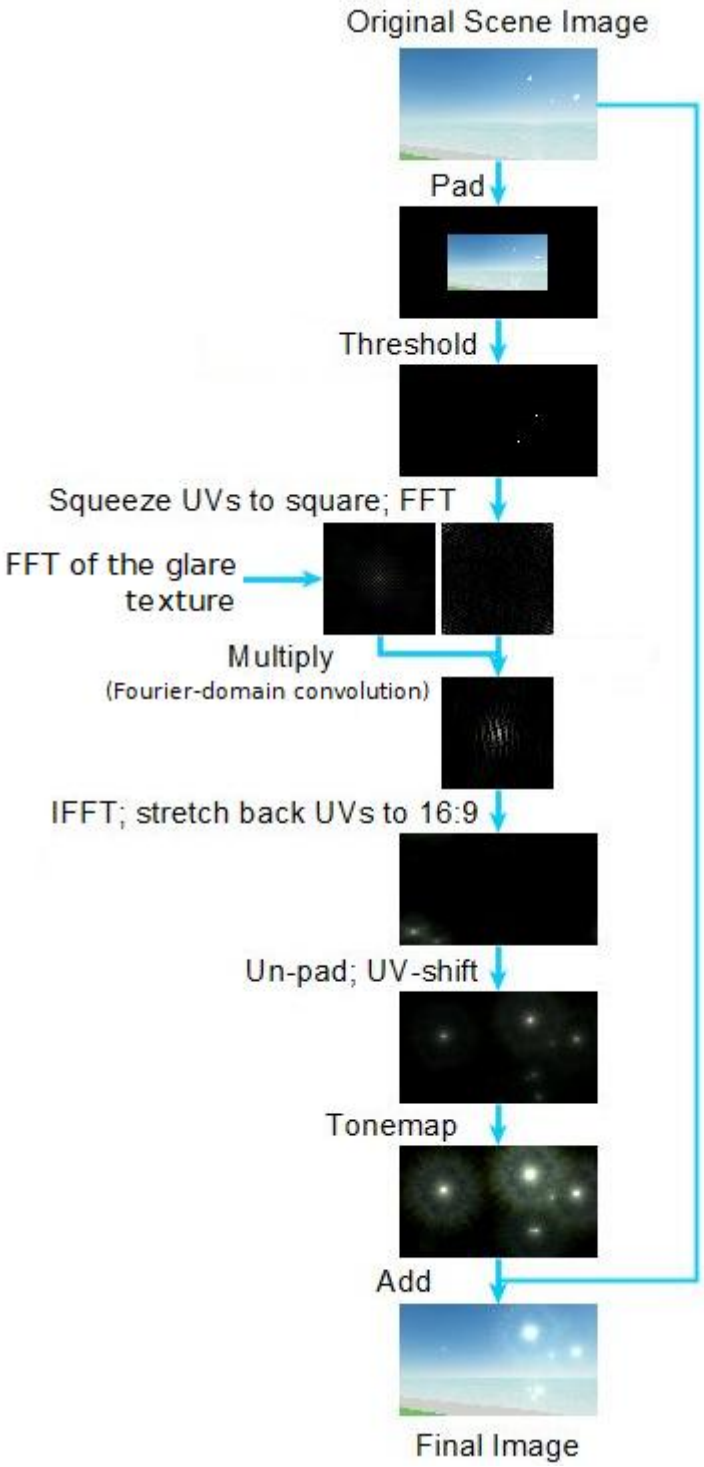


Figure 3.1: Summary of the effect compositing pipeline.

Basic compositing

Once the GPU FFT was implemented and tested, the next goal was to set up a simplified compositing test that would convolve a placeholder image onto the scene based on where the glare sources were. This required to implement a Fourier-domain convolution operation,

possibly as a compute shader too. Although, before that, implementing another compute shader for thresholding to isolate glare source pixels in the scene image would be necessary, and then even before that too, there was the need to implement a basic post-processing layer to render the effect over the scene through Unity's post-processing stack. The specifics of how the effect layer integrates into Unity's system isn't too interesting from an academic perspective, so we can move to explaining the rest of the basic compositing implementation.

Padding

As explained in section 2.5, in order to avoid wrap-around of the glare effect, we zero-pad the view of the scene before feeding it to our FFT step (which will put the image in the Fourier domain for the Fourier-convolution step). The padding is done by creating a 16:9 black texture, then placing the scene view in half-resolution in the middle of that. If we padded by expanding instead of shrinking, the extra texture resolution would drastically increase the workload for the FFTs and slow down rendering.

Thresholding compute shader

As explained in section 2.5, at that point, the thresholding operation takes in the original image of the scene, and returns a copy of that image, where pixels bright enough to pass the minimum brightness threshold for generating glare are white, and other pixels are black. Actually, to handle color, we perform this thresholding on each rgb channel separately, so for example, a strong red pixel, will put 1 in the red channel of the thresholded image, and 0 in the green and blue channels. The actual threshold value is 0.9. It's implied here, but we work with normalized float values for color.

Fourier-domain convolution compute shader

The Fourier convolution itself is a per-texel complex-number multiplication of the usual form:

$$\begin{aligned} result.real &= a.real * b.real - a.imaginary * b.imaginary; \\ result.imaginary &= a.real * b.imaginary + a.imaginary * b.real; \end{aligned} \tag{3.1}$$

Since Fourier-domain data has a real part and an imaginary part, and since we're storing Fourier data for each RGB channel, we need 6 channels, and since our texture format is float4 RGBA, i.e. it only has 4 channels, we need at least two textures to represent each image in the Fourier domain. So we need 4 textures: tex1_real, tex1_imaginary, tex2_real, tex2_imaginary. It's assumed that both input images being convolved together have the same dimensions. The result is stored in the first texture, so it's set to read-write and not just read-mode.

Re-scaling UVs for the Fourier-convolution-compositing step

The viewport is locked to a 16:9 resolution, but the FFT implementation only supports square images with power-of-2 resolutions. To comply with that requirement, the scene image gets squeezed into a 512x512 square before being fed to the FFT. Once the Fourier-convolution step is done, the output of the IFFT is stretched back to 16:9.

However, since the glare pattern is round with a 1:1 aspect ratio itself, if it gets composited onto a squeezed version of the scene image as-is, and that result then gets stretched back to 16:9 afterwards, then the glare in the final result is stretched horizontally from 1:1 to 16:9, which is elliptical instead of round. To avoid that scenario, the glare pattern is "anti-stretched" from 1:1 to 9:16 before being fed to the Fourier-convolution step. This means that

after the IFFT stretches the final image back to 16:9, the applied glare ends up round as 1:1, as it should be.

UV-shifting and compositing the glare layer

The result of the inverse FFT that gives us the final glare layer needs FFT-shifting. This operation is what you'd expect, it offsets UV by (0.5,0.5) and does modulo 1 so it loops around. The final shader that renders the glare layer does this UV-shifting, and also de-pads the image, since we convolved the glare onto a padded image of the scene, to absorb wrap-around. A final factor of 10 is applied to the values of the output, for tone-mapping purposes. The glare effect layer is transparent, and is simply added on top of the original scene image.

Modeling the glare pattern

Once we have a way to composite a placeholder onto the scene to represent glare, it's time to replace the placeholder with a proper glare pattern. As explained in section 2.2, building the glare pattern starts with anatomy components, that are rendered procedurally using implicit modeling, but drawing frames another way would also work. Taking example on Temporal Glare [1], at that point the glare pattern was computed by taking the image of the aperture with all anatomy components together, FFT-ing it to simulate monochromatic diffraction, performing what they call the Chromatic Blur operation (explained next) to get color diffraction, FFT-ing that again to have the image in the Fourier domain ready for convolving.

For the shape of the glare components, they mostly depend on the shape of the anatomy components that produce them, and how diffraction generates shapes that are hard to predict intuitively. The design of the glare effect from Temporal Glare [1] looks quite good, but was surprisingly hard to reproduce, even by approximating their anatomy design closely, which is why alternative designs were selected, as they were the next best thing results-wise.

Chromatic Blur - multicolor diffraction

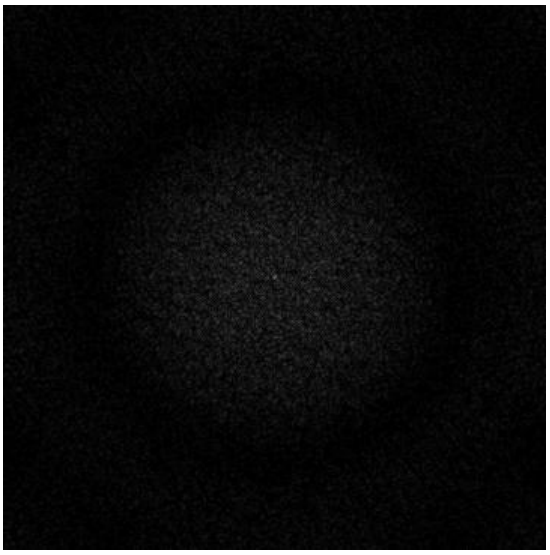


Figure 3.2: Monochromatic FFT of a particle component frame.

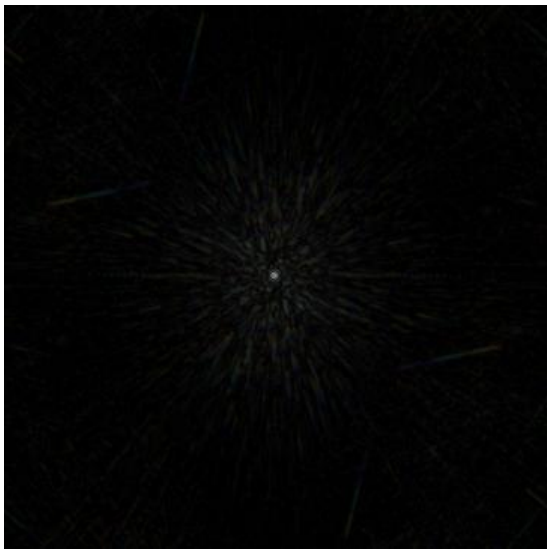


Figure 3.3: The result of the Chromatic Blur operation for that frame (with artifacts).

The multicolor diffraction operation can be summarized by these steps (details below):

1. Generating a list of 32 wavelength numbers - evenly spaced from 380 to 770nm.
2. Generating a list of RGB colors corresponding to wavelengths from step 1 - by using the conversion formulas from section 2.3.
3. Generating a list of corresponding UV-scaling factors for wavelengths from step 1 - using the wavelength ratio formula from section 2.3.
4. Bending UVs to add curvature to eyelash streaks.
5. Taking the monochromatic diffraction pattern input texture and layering 32 copies of it, each with a color from step 2 and the UV scale from step 3, divided by 32 to keep the final output within dynamic range (cf section 2.3).
6. Performing tone-mapping on each glare component before FFT-ing them again for storage in the Fourier domain.

Steps 1 and 2 - Color gradient of the visible light spectrum

We generate a list of 32 wavelength numbers evenly from 380nm to 770nm. We convert these wavelengths to XYZ, then to sRGB, using the method presented in section 2.3. cf figure 3.4.

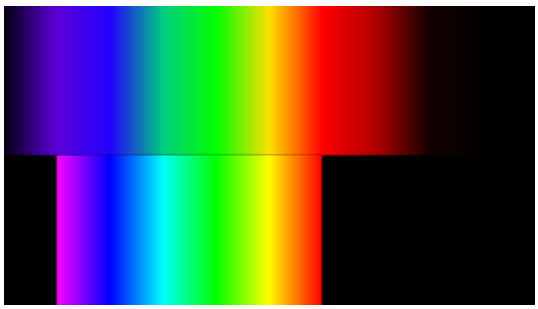


Figure 3.4: Top: XYZ gradient generated from 10 wavelength values and converted to sRGB. Bottom: HSL hue gradient from 0° to 300° (flipped horizontally to match the other gradient), for comparison. XYZ colors seem slightly less saturated overall, and don't include magenta, since it's not part of the visible light spectrum.

Step 3 - Wavelength-dependent UV-scaling

For our 32 wavelengths, we generate a list of UV scaling factors following the ratio given in section 2.3, i.e. $\frac{\lambda}{575}$. Here 575nm is the middle of our gradient, with scale 1; other wavelengths get lower and higher scales. The scale ratio might need inverting, depending on which order colors are generated in the gradient. What matters is that purple/blue rings end up closer to the center, and red rings end up further out, since longer wavelengths get diffracted wider.

Step 4 - Streak UV-bending

Glare streaks caused by eyelashes are sometimes curved, depending on eyelash curve/length, glare intensity, and how far we're blinking, cf figures 3.5 and 3.6.

Step 5 - Layering copies of the diffraction pattern

We layer copies of the monochromatic diffraction pattern with simple additive blending; copies are colored and UV-scaled, using colors and UV-scaling ratios from steps 2 and 3. To avoid brightness clipping, we divide the values of each layer by 32, the number of layers.

Figure 3.5: Real-life photo of a light seen through artificial eyelashes. [3]

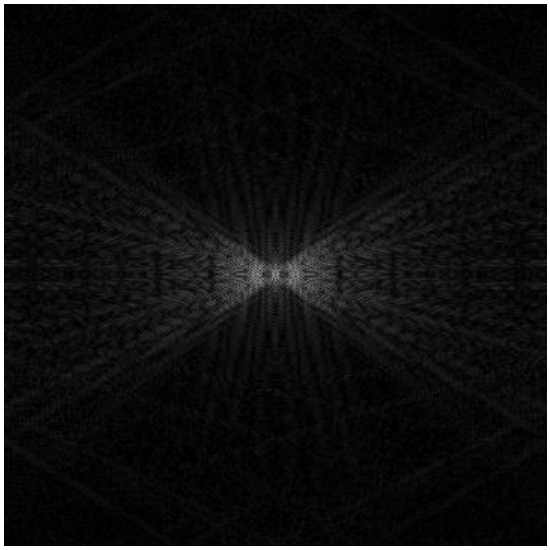


Figure 3.6: The default diffraction pattern obtained via FFT of our eyelash image has straight streaks, so we need to add curvature cf figures 3.7 and 3.8

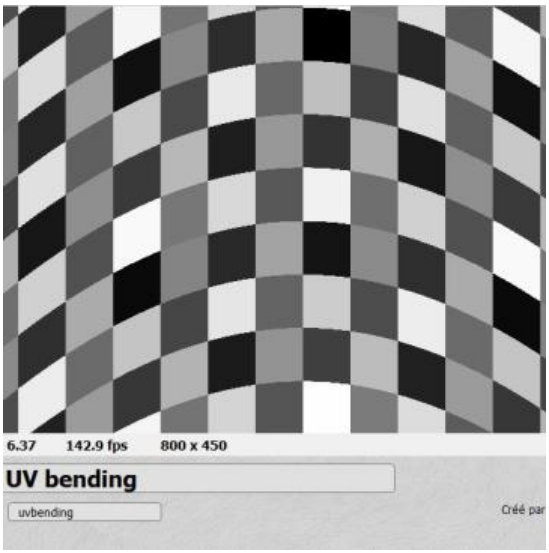


Figure 3.7: To bend UVs, we can use y = sin(x), for x in [-1;1], and then add that value as a vertical offset to the UVs based on how far from the center we are in the x coordinate.

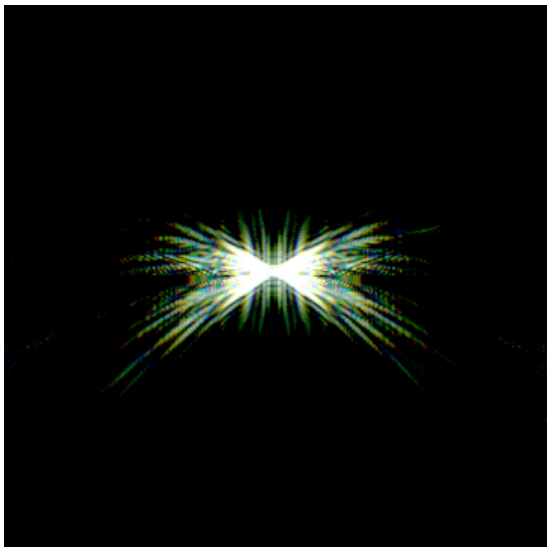


Figure 3.8: Streak texture after the uv-bend, the chromatic blur, the uv-shift and some tone-mapping.

Step 6 - Tone-mapping before FFT for storage

The glare components get tone-mapped for two reasons: to counter the fact the output of the FFTs have extreme contrast in values between the lowest frequencies in the center, and the rest of the pattern that is very dark, and to remove unwanted noise. Each texture has a power, a threshold, and factor applied. The power reduces the contrast. The threshold removes unwanted parts of the FFT pattern. The factor does most of the scaling. Numbers are given as-is in table 3.9, however their relevance is dependent on the specific FFT implementation.

	Power	Threshold	Factor
Bloom	0.45	0.00010	2000
Lenticular halo	0.45	0.00010	1000
Ciliary corona	0.42	0.00017	2000
Eyelash streaks	0.36	0.00025	500

Figure 3.9: Tone-mapping operator values for glare components.

The numbers look odd because FFT output ranges and value distributions aren't easy to predict, and choices were made via trial and error. The tone-mapped glare, even though originally computed via FFT to simulate diffraction, isn't actually in the Fourier domain, since it's an image in itself. This means we have to FFT it again for use in compositing.

Half-texel offsets

The FFT is sensitive to aliasing in input images, so anatomy components can be rendered with super-sampling, blurring or rotation, but also by taking into account half-texel offsets:

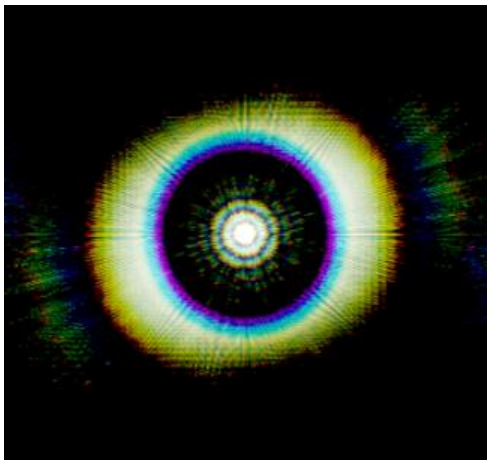


Figure 3.10: Incorrect diagonal bias.

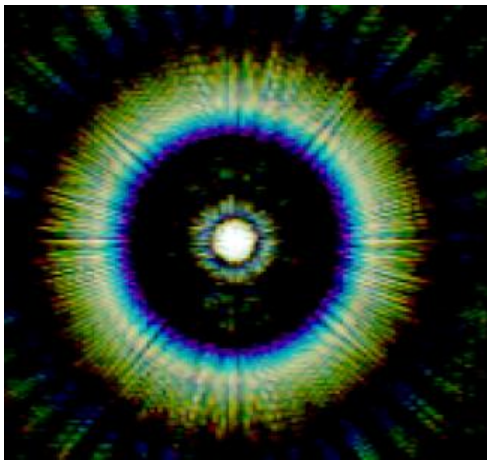


Figure 3.11: More correct.

```
//wrong:
float DistanceToCenter = distance(i.uv.xy , float2(0.5,0.5));

//right:
float2 HalfTexel = float2(0.5 / ViewportWidth, 0.5 / ViewportHeight);
DistanceToCenter = distance(i.uv.xy - HalfTexel, float2(0.5,0.5) + HalfTexel);
```

3.2 Second research question: viability from a flexibility standpoint

The second research question surrounding glare rendering appeared while implementing sun rendering in the initial 2014 project: the implementation of glare rendering failed to adapt to progressive changes in brightness and color of the sun during the sunset. During sunset, brightness decreases as the atmosphere absorbs and scatters more light away due to sun rays traversing a longer distance in the atmosphere (due to the incident angle of sunlight with the earth's surface), and the remaining light spectrum also turns more orange as blue light gets stripped by the extra scattering. This requires the glare effect to decrease in intensity progressively, as well as transition from white to orange smoothly as the sun descends below the horizon.

Essentially the problem was that since the scene image needs thresholding to differentiate pixels that should generate glare, and pixels that shouldn't, maximum-intensity glare would simply pop in and out of view instantly as soon as a pixel brightness went above or below the threshold; and what this meant for color, was that we could only ever have a given color channel present or absent from the glare, which would lock possible colors to red, green, blue, yellow, cyan, magenta and white. This was obviously an issue since we needed to render the sun's glare as orange, and also progressively blend between hues, not just snap from one major hue to the next.

Real-time rendering (4th edition) [8] has a section on lens flares and bloom, and on page 527, it gives a solution to the lack of flexibility of my initial thresholding implementation: "Instead of thresholding, high dynamic range imagery can be filtered for a better result", as well as "Any bright pixels are retained, and all dim pixels are made black, often with some blend or scaling at the transition point".

It seemed very obvious in retrospect, but instead of thresholding brightness values in a way that would either select or not select pixels to generate glare, brightness values needed to ramp upwards from 0 glare intensity after passing the threshold, rather than going from 0 to full glare intensity as soon as a threshold is passed.

This solved the question of whether glare rendering could adapt to progressive brightness changes, but it also solved the question of how to adapt to progressive color changes, since now, brightness values for each rgb color channel could now ramp up separately in a way that preserves the ratio of values between those channels. So, red and green channels could maintain a 2:1 ratio to keep the glare effect orange as brightness changes. This would also let glare sources take intermediate hues, as well as change hue over time, since those intermediate ratios could now be expressed. We could have glares of different colors and brightness on screen at the same time from the start too, it turns out.

So that answers the second research question about the viability of rendering glare in a game engine from a flexibility standpoint, and specifically for our test case of sun rendering during a day-night cycle. This has a couple consequences on the implementation, let's cover those next.

Improved thresholding

The original scene image is thresholded and tone-mapped to select which pixels will trigger glare and how much. The threshold value is at 0.9, and the remaining values are multiplied by 10,000:

$$float3\ rgb\_channels = (max(0, (pixel.rgb - 0.9) * 10000)); \tag{3.2}$$

The multiplication by 10000 is done to compensate for the effect of the following FFT on the values. Also, it's important that the threshold is subtracted from the values (without ending up negative), and values below it aren't just set to 0 while values above stay at 0.9, otherwise glare will pop in and out of view instantly instead of fading in and out smoothly, and it would also break hue blending since individual rgb channels would pop in and out of the glare pattern.

Color and ACES tone-mapping

Using a progressive brightness threshold for compositing glare helps with brightness scaling, as well as color blending, however, the RGB format still suffers from its own limitations, and we'll want to use the ACES color transform. The implementation details of the ACES color transform haven't been investigated during this project. An implementation of the ACES transform is provided by Unity's Post-Processing v2 stack, and is what has been used for this project. It is applied both on the values of the original scene image before we do any computation with it ourselves, and then the final result of our glare effect layer is also passed through the ACES transform.

3.3 Third research question: viability from a performance standpoint

The third main research question was whether the Temporal Glare technique could even work in real-time with acceptable performance, and how. Temporal Glare reported in 2009 that without pre-computing anything, and when using full-screen Fourier-domain convolution for compositing (which is more expensive that placing billboards in simple scenes with few, or small glare sources), they reached a framerate of 30fps. It seemed reasonable to assume performance would be better with more modern hardware and by pre-computing some of the effect, but how to pre-compute the effect while keeping it flexible, and whitout it being a simple uninteractive repeating loop needed to be figured out. The doubt surrounding that question was initially also tied to initial naive brute-force implementation ideas for the color-blending and brightness-scaling that seemed extremely expensive at run-time without incredible optimization (the initial worry was that many fullscreen FFTs per frame would be needed, possibly to composite glare for more color bands than the 3 rgb ones, and for several brightness bands; this idea turned out irrelevant). However after the thresholding issue was resolved, good performance seemed within reach, since brightness-scaling and color-blending would be cheap.

An early idea for brightness-scaling purposes was to construct the glare layer by layer, however, this idea ended up being even more relevant for real-time animation. The idea was that we could reveal new layers as brightness increased, since for instance, the lenticular halo only seems to become noticeable for quite strong glare, while bloom appears first at much lower source brightness levels. If each glare component (bloom, corona, halo, streaks) corresponds to a different effect layer, then the rendering of each component might be controllable with different parameters, and that's where the animation comes in.

The first tests for pre-computing the glare effect actually precomputed an animation loop using a different flipbook texture for each glare component, so they could each be on their own layer. I then realized, instead of animating using a loop that simply repeats, we could instead use the flipbook textures to store the possible range of motion for the anatomy components (pupil, eyelashes, lens gratings, particles), and animate by selecting frames based on animation parameters. For example, instead of animating the pupil expanding and contracting and looping that, we could instead pre-render the pupil at every possible diameter, and then pick the pupil frame that corresponds to the relevant diameter at render time, since pupil expansion depends not just on time, but also scene brightness, which is an unpredictable

parameter that will react to users moving the camera around. Eyelash blinking and squinting might also depend not just on time, but also scene brightness, and eyelashes can be rendered as a range of motion, from fully open to fully closed. Lens gratings will also expand and contract like the pupil, with slightly different parameters (cf section 2.4). Particles aren't as easy to animate parametrically since there are at least 3 different types of particles: those in the cornea (static), those in the vitreous humor that might need representing as two layers that move based on head momentum, and those in the lens, that mostly squeeze and stretch towards and away from the center of the aperture, which might be pre-computable as a range of motion, from fully stretched, to fully squeezed. These options cited for animating the particles have been unexplored in the actual project, and particles have been kept as one layer with an animation loop.

For compositing the glare layers, I initially thought I would need to perform a convolution for each effect layer that needed compositing onto the image, but then I realized, I could first add all the layers together directly in the Fourier domain, and perform a single convolution for the summed effect layers. Some tests confirmed this works because the sum of FFTs gives the same results as the FFT of the sum of the images. At this point, at pre-computation time, we have all glare components stored as flipbook textures, and at run-time, we mostly just need to sum the selected frames for each glare component, and multiply the result with the scene image's FFT to convolve the glare with the scene image. We have removed a lot of excess computation operations compared to earlier implementations, though at the cost of a non-trivial chunk of memory for flipbook texture storage, but this version ran the scene at about 70fps, which was above the initial target of trying to run the effect at above 60fps for a 1080p viewport.

This meant that performance would be at least somewhat acceptable, which answered the research question: yes, glare can be rendered with usable performance in real-time, under certain conditions. The framerate was *only* 70fps, and that was because the glare effect pipeline took about 14ms to compute per game frame, for a 512x512 glare on a 1820x1024 viewport, where 12ms were taken up by the Fourier-domain roundtrip the scene image does (forward, and inverse FFT), which represents practically the entire rendering budget of 16.67ms. The test scene had a couple other moderately expensive shaders (somewhat realistic water, cartoon grass), so the glare effect would run in a scene with at least some kind of nominal computational load, rather than an entirely empty scene. Implementation details for the pre-computation and runtime flipbook animation are given below.

Motion for each anatomy component

Figure 3.12: Range of motion of the eyelashes.

Instead of simply rendering an entire animation loop for the whole glare effect, the motion of the various anatomy components is rendered separately into flipbook textures; but also, instead of simply rendering an animation loop for each component, only their possible range of motion is rendered into the flipbook textures (cf figure 3.12), and the actual animation is created by deciding in real-time which frame of the range of motion to render for the four components, and then adding those 4 frames together to create the glare effect for display. In the implementation, each flipbook texture stores 60 frames. Since at run-time, we composite

them onto the scene via Fourier-domain convolution, it saves time to store them directly in the Fourier domain so we don't need to re-FFT them every frame.

The pupil and gratings can contract and expand over time, and the eyelashes can open and close. What is used is an animation where the pupil's diameter follows a sine pattern, but only the expanding part of the loop was kept, such that it can just be played backwards to get the contraction part of the animation. This covers the necessary range of motion from fully contracted to fully expanded. The movement for the particles however is still a looping animation, since the principle of a range of motion for random particles would've been trickier to implement.

Measuring average scene brightness

One possible technique for computing average scene brightness to decide blink extent is to successively copy the scene image to lower and lower resolution buffers -512 into 256 into 128 etc.- such that we're left with a 1x1 texture that represents the average scene color thanks to the bilinear filtering applied at each step. This can then be used as an animation parameter, cf following sections.

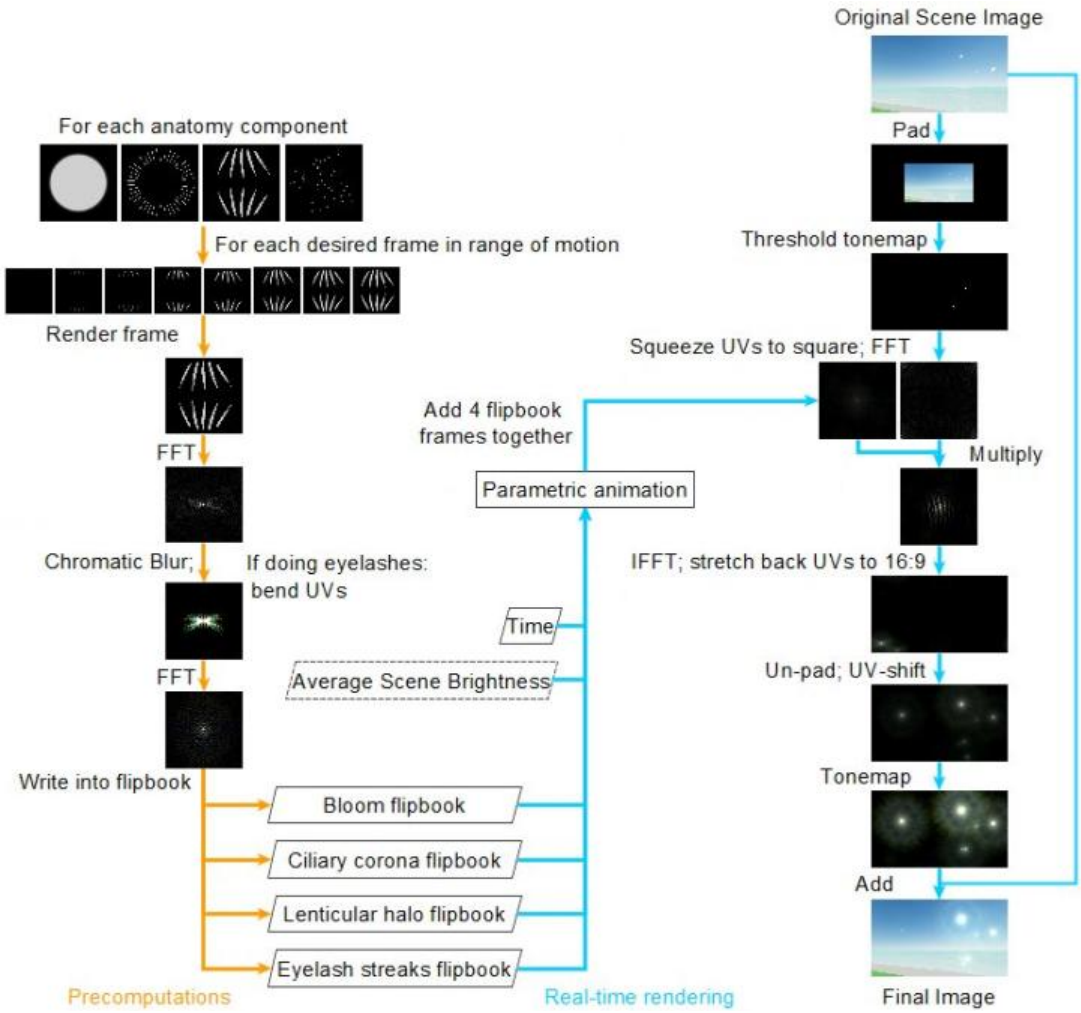


Figure 3.13: Summary of the overall effect pipeline.

Rendering Anatomy Components Separately

Anatomy components are animated individually and stored separately, which gives us the option, at runtime, to dynamically pick and choose which specific frame to render for each component based on animation parameters such as time and scene brightness. However, separating components like this means we don't get to use the pupil as a white background for the other components (eyelashes, particles, gratings), since, when we separate them it causes to render them on an empty background, i.e. black on black. To deal with this issue, components are all rendered white on a black background instead - which means the eyelashes, particles and gratings are rendered in inverted colors. This works because the FFT of an image in inverted colors gives the same frequency spectrum as for the original image, but with inverted phases, however the phases are discarded in the multicolor diffraction (called Chromatic Blur in Temporal Glare [1]), so we're not changing anything by inverting values at this point.

Selecting and combining flipbook frames

Since the flipbooks are stored as 3D textures, and assuming each flipbook stores n frames, then the 3rd UV coordinate (i.e. "UV.Z") we use to sample frames would be

$$UV.z = index * (1/n) + (0.5/n); \tag{3.3}$$

since UVs are expressed in the interval [0;1]. The selected frame index depends on which component we're rendering. That's because the different glare components react to different parameters, for instance, the eyelashes and the pupil contract if the scene brightness is strong, while the particle movements are animated as a loop at a constant rate. The index for the pupil diameter can move back and forth from 0 to n at a constant rate. The index for the lens gratings' diameter follows the same pattern, with a fixed offset in time of 5 frames. The index math for the eyelashes regular blinking is a bit simplistic and doesn't represent accurate blinking behaviour with quick blinks every couple seconds, and instead uses the same index math as the one for the pupil, but at a different frequency. The index for the particles just loops forward from 0 to n at a constant rate.

Once the 4 flipbook frames are selected, they're added together and the combined texture is stored into a float texture. The different components are mixed with different brightness scales for rendering:

$$sum = bloom * 1 + halo * 0.75 + streaks * 0.5 + corona * 1 \tag{3.4}$$

Combining the frames at runtime has to be done in the Fourier domain, since they're stored in the Fourier domain in the flipbooks, and we only leave the Fourier domain after the glare has been convolved onto the scene view.

4 Results

4.1 Performance

Computation Time

Performance-wise, the results of the project are given below.

	512p viewport	1024p viewport
512p glare	12.5ms	14ms
1024p glare	33ms	34ms

Figure 4.1: Execution time for the entire glare effect pipeline every frame.

Most of the time is taken by the the forward FFT2D and inverse FFT2D steps. The remaining 0.5-2ms of computation time is taken by the rest of the rendering code, depending on viewport resolution.

	FFT	IFFT
512p glare	6ms	6ms
1024p glare	16ms	16ms

Figure 4.2: Execution time for the FFT kernel.

Memory Usage

The memory footprint of the final build is described below, but the bottom line is: our implementation takes up to about 1GB of GPU memory, of which 970MB for real-time use.

Pre-computation buffers, which can be freed once the pre-computations are finished:

- One 4x resolution texture for rendering anatomy components in high-resolution
- Four other generic 1x resolution buffers for pre-computations. Generic means that instead of allocating one buffer for each step, buffers are overwritten as their content becomes redundant. 1x means that if the glare is 512x512, then the buffers are 512x512.

Real-time computation buffers:

- Eight 1x resolution Fourier-domain flip book textures with 60 frames each
  - 4 to store real values / magnitude
  - 4 to store their imaginary values / phase
- Five 1x resolution RenderTextures for real-time computation steps:
  - Thresholded version of the scene view
  - 4 generic 1x buffers for holding 2 complex images:
    - real 1
    - imaginary 1
    - real 2
    - imaginary 2

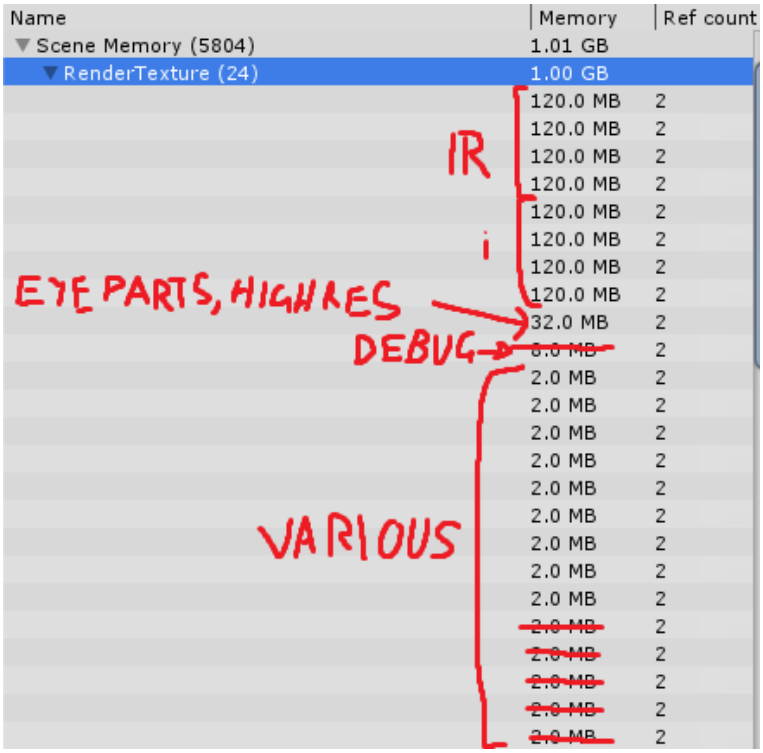


Figure 4.3: Buffers shown in Unity's memory profiler.

4.2 Test cases

Sun / Large glare sources

Attached to this report should be various videos. The sun can be seen in real time in two different video files: one with water specular highlights, and another one that displays a sunset animation.

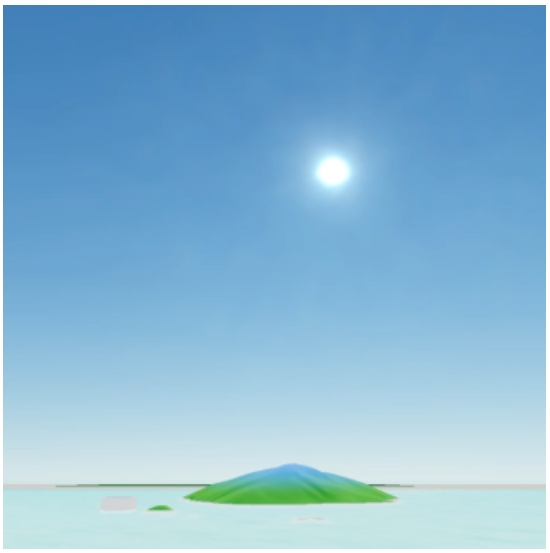


Figure 4.4: Large glare source on a bright background.

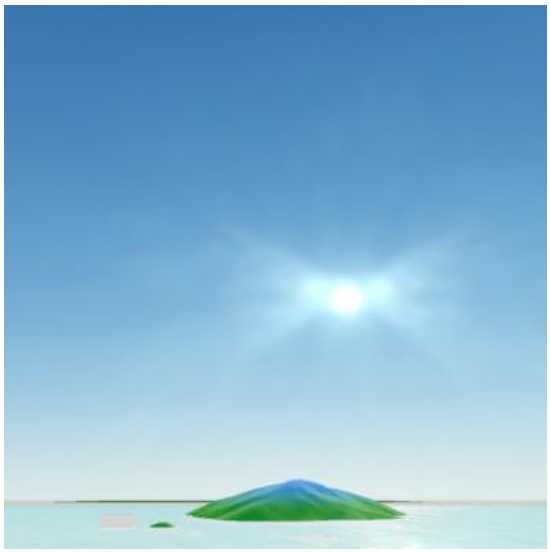


Figure 4.5: Large glare source on a bright background, while blinking.

Figure 4.6: Large glare source surrounded by a darker context.

Figure 4.7: Large glare source surrounded by a darker context, while blinking.

Stars / Small glare sources

While these screenshots might make it look like the star rendering works to some extent, due to aliasing, when viewed in real time, the glare flickers very unpredictably for very small glare sources, and not every star triggers glare even if they have the right brightness. cf dedicated video file.

Figure 4.8: Small medium-intensity glare sources.

Figure 4.9: Small but very bright glare source (left).

Figure 4.10: Small medium-intensity glare sources on a dark background.

Figure 4.11: Small medium-intensity glare sources on a dark background, while blinking.

Source color variations

There is also a dedicated video file that shows the hue shifting over time.

Figure 4.12: Glare with colors that typically break without HDR: azure blue, orange, apple green.

Figure 4.13: Color variations applied to sunlight.

Source size and brightness variations

The two dedicated video files show two animations: one changes the size of the glare sources over time, the other changes their brightness over time.

Figure 4.14: 3 sources of equal size, but different brightness.

Figure 4.15: 3 sources of equal size, but different brightness, larger.

Figure 4.16: 3 sources of equal size, but different brightness, even larger.

Figure 4.17: 3 sources of equal brightness, but different sizes.

Figure 4.18: 3 sources of equal brightness, but different sizes, brighter.

Figure 4.19: 3 sources of equal brightness, but different sizes, even brighter.

There are few main take-aways here:

- Increasing the size of a glare source makes the overall effect appear brighter/more opaque, since more copies of the texture get overlapped by the convolution.
- For single-pixel glare sources, only one "copy" of the glare texture is visible, and it lets the aliasing of the glare pattern appear visibly. This is expected since the glare pattern was purposefully computed in low-resolution for performance reasons. What's interesting is that once the glare source is larger than a handful of pixels, the convolution blurs the glare pattern, and the aliasing becomes much less noticeable.

Water / Dynamic glare sources

Obviously, this is quite a dynamic animation in real time, and the results are best observed in the dedicated video file.

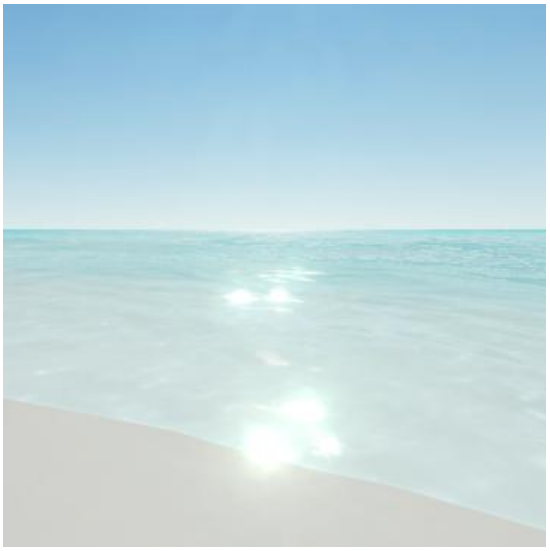


Figure 4.20: Specular highlights on water.

Figure 4.21: Specular highlights on water, while blinking.

Figure 4.22: Specular highlights on water, from a further viewpoint.

In the meantime, because the contrast in the scene isn't very high, it might not be obvious enough from the screenshots that the glare seems to gets tinted cyan, instead of having the full color diffraction gradient in the effect here, as we'd expect from a white light source. However, we actually have a mix of white and cyan pixels generating glare, and on some screenshots we can see the specular reflection highlights from the water be pure white, cf figures 4.23 and 4.24.

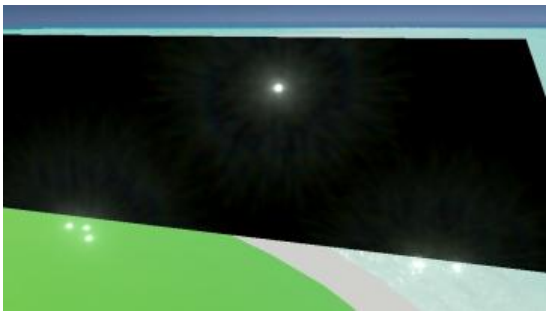


Figure 4.23: Comparing glare from white lights (left, center), to glare from water highlights (bottom right). Here water highlights seem identical to those of the white lights.

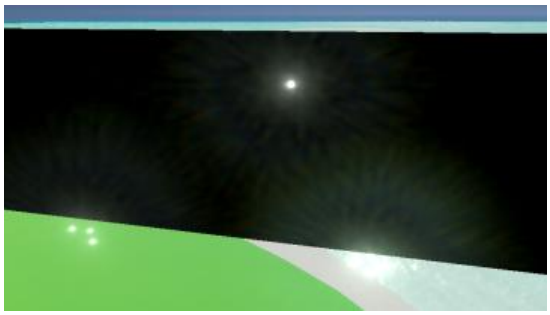


Figure 4.24: On this screenshot however, water highlights appear noticeably green/cyan in comparison to the glare from the white lights.

Partial source occlusion

Figure 4.25: Glare source partially occluded by an opaque obstacle.

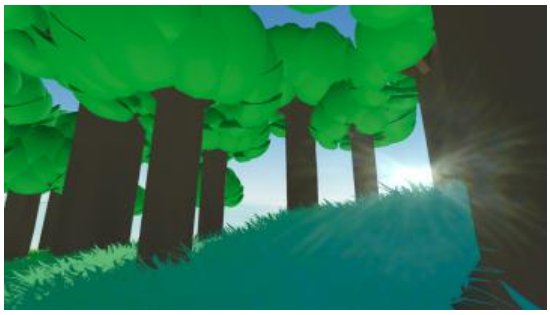


Figure 4.26: Glare source partially occluded by a cut-out obstacle.

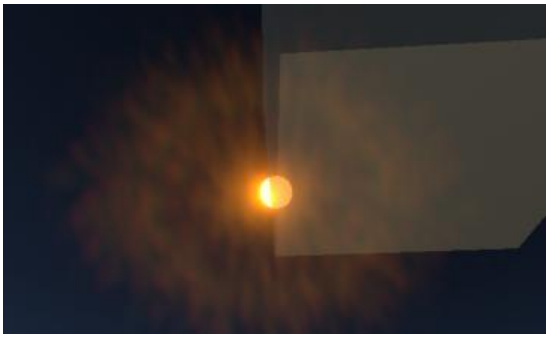


Figure 4.27: Glare source seen partially occluded by a transparent obstacle: partial opacity of the obstacle reduces the brightness of occluded pixels.

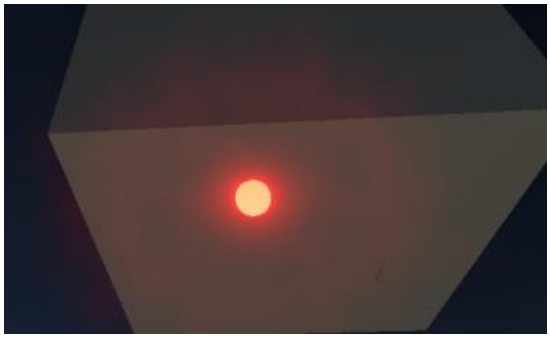


Figure 4.28: Orange light seen entirely through the obstacle: the light turns red, because the obstacle pushes the lower green value below the glare threshold, but not the red one.

5 Discussion

5.1 Results

Quality

Some implementation issues have been left unresolved, for example:

- The size of the glare effect on screen changes based on how the viewport resolution gets downscaled internally to save performance. If the viewport is 1820*1024, the glare image has a certain size once composited onto the scene, but if the viewport is treated as half that resolution to reduce the cost of the FFT and IFFT, i.e. the viewport gets treated as being 910*512, then the final size of the glare effect on screen will double. This shouldn't be too hard to fix, and doesn't cause significant issues research-wise so it didn't have priority.
- There is also currently no step in place to adapt image to viewport resolutions other than 16:9 aspects with power-of-2 vertical resolution, i.e. 1820*1024 and 910*512. It might've been interesting to see whether simply stretching the glare layer to fit would've looked fine or not, or if it would've required extra work to avoid noticeably downgrading image quality in the process.
- Perspective near the edges of the viewport stretches glare sources since perspective deformation is stronger (cf figure 5.1), which increases the number of pixels generating glare, which means more copies of the glare pattern will be convolved over them, which increases the opacity of the glare pattern when seen near viewport edges. This might be addressed by dimming brightness values around screen edges in the thresholding step to compensate.
- More work on the eyelash streak animations would've been very welcome, some missing features include:
  - Reducing the incoming brightness during blinking and squinting, since that's one of the main effects of eyelashes in real life.
  - Bending the glare streaks as the eyelids close further, since the upper and lower eyelashes should actually collide with each other.

Figure 5.1: The sun becomes elliptical when it is seen near viewport edges, due to perspective deformation.

  - Eyelashes shouldn't just produce line-shaped streaks, but also generate larger colorful brush patterns (cf figure 2.4). They might be caused by light being reflected/-transmitted by eyelashes, and depth-of-field causing those reflections to become blurry and then diffracted inside the eye again.
  - Blinking wasn't really animated properly, it happens slowly and at regular intervals, and instead should probably happen much faster, and at semi-random intervals.
  - When a glare source moves relatively to our field of view, it "scrolls" past different eyelashes, and the streak pattern should change to reflect that. This streak-scrolling animation has been theorized by Kakimoto et al. [3], and their idea was to pre-compute glare patterns for glare sources at every possible screen-space position. However they do mention this would cost a very large amount of memory. In our case, we have an extra complication, which is that we composite glare using Fourier-convolution, and so we have no way of feeding to that convolution which glare pattern to convolve on which part of the screen, other than maybe running many Fourier-convolutions for individual tiles of the scene image with a different glare image for each tile, however at that point we might as well just use a regular kernel convolution to do that. Which might turn out expensive since eyelash streaks are actually the largest part of the glare pattern, and might take up the entire width of the screen or more, and since convolving large textures might raise computation time a lot.

- Speaking of using kernel convolutions to composite different patterns on different parts of the screen, that same idea might be used for star rendering, and/or rendering of very small and low-intensity glare sources. There's a problem with our current implementation when it comes to rendering those, and it's that the particles that generate the needle patterns of the ciliary corona are one texture instead of several layers, and it means a small and dim glare source (a star for example), will trigger the entire corona pattern, instead of just the small needles in the center. Some efforts were made to make the center of the corona pattern brighter so that the inner needles would trigger at lower source

intensities than the outer needles, but the underlying inaccuracy remains, since a better solution might be to split the corona rendering into several layers anyway. An idea for rendering very small glare effects like this might be to use a regular kernel convolution, since convolving small patterns isn't too costly computationally, and it could let us render slightly different needle patterns for stars at different positions on screen instead of the exact same needle pattern for all of them. Another reason to split the glare rendering for small glare sources into a separate effect is that we currently reduce the internal resolution of the scene view for performance purposes during convolution, which means glare sources that are smaller than 2 pixels suffer drastically from flickering caused by aliasing, so star rendering would work a lot better in full resolution.

- As shown in figure 4.28, an orange light-source that gets dimmed by 50% -possibly due to being seen through a semi-opaque obstacle- won't stay orange if the green channel gets pushed below the threshold but not the red channel. This is a remaining limitation of the rgb system, even after the use of the ACES color transform: it doesn't represent the actual spectrum of the light. Nakamae et al. [6] investigated handling the actual spectrum of glare sources for rendering, but this aspect hasn't been investigated in our project.
- There are a handful of visual artifacts remaining that could be removed with better tone-mapping throughout the effect pipeline. There are some artifacts in the modeling of the glare components (cf figure 5.2), and there are some artifacts in the rendering of the scene (cf figure 5.3). Tone-mapping has been iterated on via ad-hoc trial and error, and to put some order in the tone-mapping, a survey was done of how brightness ranges and value distributions progress through the pipeline, and the main obstacle that was identified was predicting formally what value distributions the various FFT steps would give us. No further conclusions were reached, and trial and error continued.

Performance

The performance of the glare simulation given by Temporal Glare [1] in 2009 is 52fps, by computing a new glare pattern from scratch every frame, and without significant compositing/rendering costs (it's for just one billboard), which gives them 19ms of computation time for the glare pattern. A paper by the same lead author from 2016 [9] gives a new performance measurement of 10ms for the same simulation on more recent hardware. Since our runtime cost of building a glare pattern for a frame during runtime takes 2ms, that might count as quality and memory footprint tradeoffs that yield a 5x speed improvement, give or take the fact that our measurement is taken on late-2017 hardware; specifically, our measurements have been taken in Unity3D 2018.3.6, and the code has been written in C# and HLSL for Windows 10 on a mid-tier MSI laptop - GP72M 7REX Leopard Pro (i7 processor, NVidia GTX 1050-Ti GPU).

The performance target of 60fps on a 1080p viewport (or close) has been reached, but if we're discussing the viability of the technique, it might be relevant to take into account the performance requirements of the next couple years, and that includes larger screens and faster refresh rates. If we consider that 120Hz screens in 4k resolution might become the new norm, it remains to see whether available computing power will be able to keep up, if the algorithmic time complexity of a Fourier round-trip would become prohibitive, or if possible performance improvements would lower the current 12ms computation time for the Fourier transforms every frame, which is the single biggest bottleneck until a better way of compositing the glare is found - give or take the fact that rendering with billboards is still an option, but its cost can rise quickly for scenes with many pixels generating glare; however in a well-controled setting with a limited number of glare sources, our technique rendered on billboards would be quite fast.

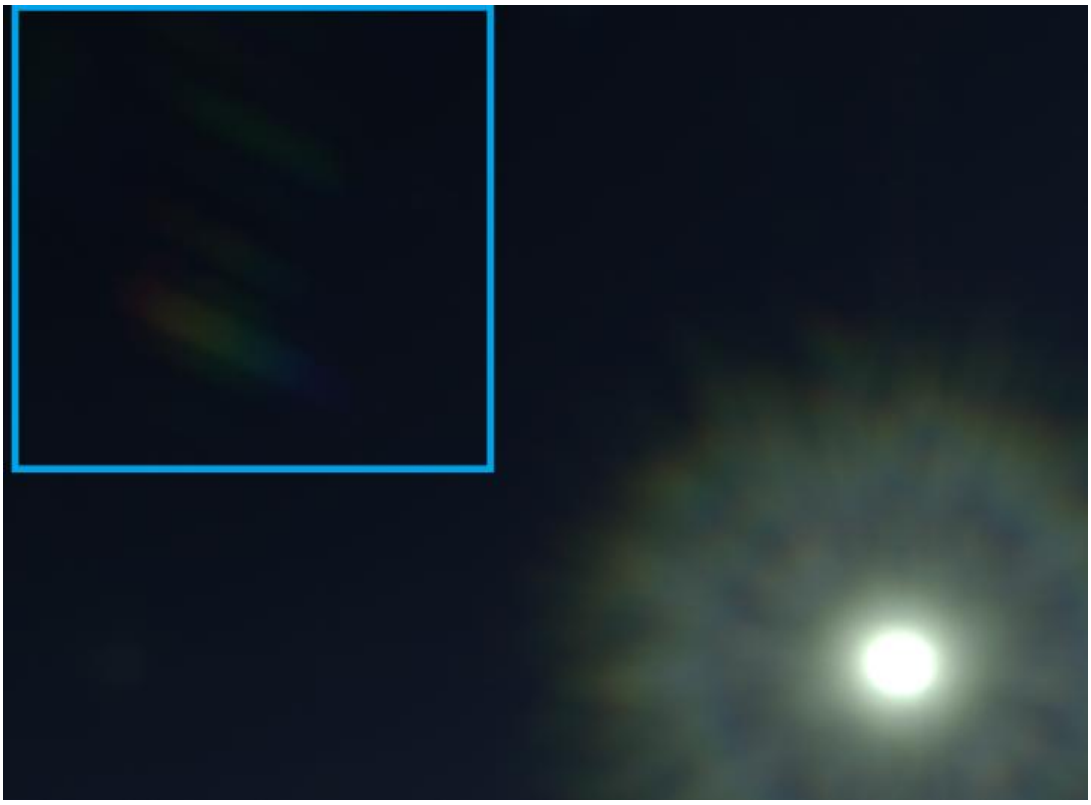


Figure 5.2: There are superfluous streaks in the corners of the halo component texture due to errors in modeling.

There are also a couple unexplored options for improving performance that stand out:

- There has been no test of performance of parallelized basic kernel convolution to compare with parallelized Fourier convolution during this project. In the case of this project being carried out further, this would be the very first thing that would need doing.
- There is the possibility of using more half-precision float values rather than regular float values in the effect pipeline. The reason this might be relevant is that Unity's documentation indicates its HDR texture format uses an ARGBHalf format, and so doing our computations with regular float values might cost unnecessary performance, though this needs investigating. Using Half-floats might save us the cost of converting from 32 to 16-bits since we're using float4 arrays for compute shader buffers, while the data itself is stored as half4. Half floats might also save cache space and reduce cache reloads, as well as reduce data transfer times if there are any in our system.
- Another possible opportunity for improving performance might be to investigate modifying the GPU FFT implementation so it can work with real-valued input and output textures (i.e. not complex). This might save computation time and storage space since the scene image is real-valued, the final output only needs to be real-valued, and some of the intermediate computations might be wasting performance handling imaginary-valued phase information in Fourier-domain data.
- The GPU memory footprint is very large, at about 1GB. Some of this could possibly be reduced by exploiting the fact some anatomy components (pupil, gratings, some particles) are radially symmetrical, which means we could maybe store only a quarter

Figure 5.3: There are sometimes remaining noise spots due to the Fourier round-trip used for compositing that don't get thresholded out of the image properly.

of their pattern, and recreate the full pattern at run time by simply using rotated copies of that quarter. Otherwise, we can use fewer frames in the flipbooks: initial tests had 30 frames instead of 60, and the effect was usable, just animated less smoothly; maybe some frame-interpolation or motion blur could arrange that.

5.2 Method

The method for picking sources and reference papers for this project was pretty typical: taking the more recent and relevant paper, and checking its own sources. Projects on glare rendering haven't been very numerous over the years, and most of them trace back the origin of the techniques to the same handful of papers. There was a SIGGRAPH course in 2015 on the topic of Glare Rendering by Kakimoto et al. [10], which is one of the most recent sources for our project, and it has a good summary of the research up to that point. The references given in that course made up the main group of references for our project. It turns out that when looking at the list of relevant sources from Temporal Glare [1], there is a large overlap with the list of references given by the 2015 course too. There are probably more recent research projects that have taken place since that course, however nothing significant came up when searching for them. The number of sources already available for our project seemed sufficient already, but more up-to-date sources would've been interesting.

However, some of the methodology for this project could've been better; one of the main issues for instance is that since the theory seemed initially slightly beyond my reach, I focused on implementing what I understood, and skipped over the more theory-heavy parts of the technique. This let me move forward with implementation faster, but once it was time to write this report, and especially the theory chapter, it turned out that there were a few

important insights in the theory, that I only learned after the project was already considered complete.

For example, the fact that the Fresnel approximation to the Huygens-Fresnel principle seems to mostly be used for analytical purposes, while we are working with numerical math, indicates to me that maybe for pre-computation purposes, running the original Huygens-Fresnel integral might've simplified some of the implementation by computing diffraction accurately directly rather than having a more complex implementation that involved more possible failure points such as FFTs that caused effect brightness values to have extreme distribution patterns during pre-computation, which made tone-mapping more difficult. I later wrote a shader to test using the Huygens-Fresnel equation directly, and in a dozen lines of code it made relevant interference patterns.

Another valuable theory point that was understood too late was the fact that there are actually more than one type of particle inside the eye, and that the various types require different animation parameters. Knowing this earlier would've let me implement a more complete animation system for the ciliary corona.

Ritschel et al. [1] also mention that they used time-damping on the values of the scene brightness for controlling squinting, and I missed that when working on eyelash animation. Moreover, after I finished implementing a simplified version of their technique to make sure glare rendering worked, not every further feature they suggest was implemented. For example, the noise pattern in the animation of the pupil and lens gratings wasn't used, and the Fresnel term in their diffraction equation (which they call E in their equation 3) hasn't been integrated in the code for this project.

Not every mistake due to a lack of familiarity with some of the theory involved glare-specific knowledge however. One mistake that was made concerned the use of padding to avoid wrap-around during convolution: the entire time, the project was done with padding on both the scene image and the glare pattern, while only the padding around the scene image was necessary. The result of this error is that every screenshot in this report has half the resolution possible for the glare effect. Thankfully the effect looked mostly presentable already, but the visual quality could've been better for free the whole time. Figure 5.4 is a screenshot taken with the improved resolution: the eyelash streaks and bloom look quite clean, but the halo and corona seem off. The tone-mapping of the components might need re-doing to account for this.

5.3 The work in a wider context

Glare rendering is a niche topic, but it can have a few uses.

In creative industries such as video games, or interactive artistic applications, any type of effect can have its use if they fit the creative direction for a given project. Uses could also be found in the movie industry nowadays, since previsualizing movie effects in realtime during production is becoming more common, and subjective glare effects might be relevant for creative reasons.

Motivations for using subjective glare effects might turn out to be numerous, but it seems natural that this type of effect would be attractive to whoever needs to evoke the feeling of seeing through the eyes of a character in a way that emphasizes the realism of their vision. An example of game that might have made relevant use of human-eye glare is *Amnesia: The Dark Descent* (Frictional Games, 2010), which is a first-person horror game where our

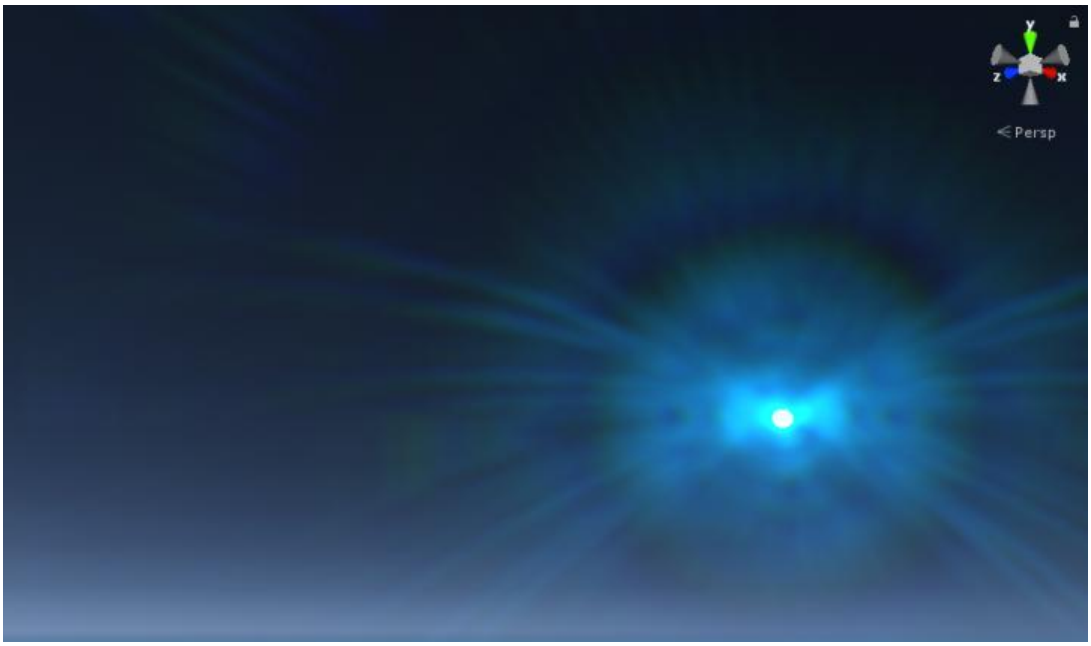


Figure 5.4: Final build of the glare effect with double the resolution, shown on an azure light source. The gizmo in the upper right corner is rendered without any anti-aliasing, and gives a point of comparison of how much smoother the high-resolution glare is in comparison.

view has visual hallucinations based on real perceptual distortion phenomena that humans sometimes experience, which indicates that making our protagonist feel more grounded and real was part of their creative direction to give the feeling that the danger to them (and us, the players) is more tangible. More outlandish uses could be found of course, surrealist art thrives on incongruously realistic elements and picking up such a niche technique might be relevant for that use.

The other main use would probably be scientific, and concern simulators that try to put users in immersive conditions, or for predicting how bright light sources might impact vision, in contexts where vision needs to be experimented with in real time. The use of glare rendering might also be adapted to simulate vision for non-humans in interactive experiments, maybe to simulate the temporary blinding impact of man-made structures either emitting or reflecting high radiance in the visual environment of endangered species for example.

6 Conclusion

The main question when starting this project was whether more realistic glare rendering techniques such as the one put forward by Temporal Glare [1] could replace lens flare rendering in video games. The initial doubts surrounding the viability of the technique concerned its flexibility and performance.

This project shows that physically-based glare rendering is viable for use in video games and interactive applications, but under certain conditions. The two most important restrictions are the performance of the compositing if we want the effect to render with a constant computation time regardless of the number of glare source pixels on screen, and the accuracy of the modeling and simulation when pre-computing for performance reasons, especially with regards to the eyelash streak animation and compositing.

It would make sense for future work to focus on trying to lift these two restrictions, possibly by finding different compositing approaches that would be better suited to the requirements of the various glare components, instead of trying to fit all the glare components through the same flipbook system. Bloom could be its own effect. At least one of the ciliary corona layers could be a separate kernel convolution used for rendering stars and other small glare sources. Eyelash streaks might also need to be rendered using a typical kernel convolution so we can decide which streak pattern to place based on the screenspace position of individual glares sources, or even placed by another type of technique than convolution. Concerning performance, there might still be low-hanging fruit optimizations possible that future work could identify.

7 Appendix

7.1 Deriving the Fresnel Approximation

The Fresnel diffraction approximation is derived by replacing occurrences of the term r in the Huygens-Fresnel equation. In one case, we replace it by an equivalent binomial series with only its first 2 terms, which is valid if further terms are negligible. In other cases, we replace it by z. Now, the term r isn't obviously a binomial, since it looks like this:

$$r = \sqrt{(x - x')^2 + (y - y')^2 + z^2} \tag{7.1}$$

We can separate terms of r that are x-and-y-related from its z-related term:

$$\rho^2 = \left(x - x'\right)^2 + \left(y - y'\right)^2 \tag{7.2}$$

$$r = \sqrt{\rho^2 + z^2} = z\sqrt{1 + \frac{\rho^2}{z^2}} \tag{7.3}$$

and following the rule of Taylor series,

$$\sqrt{1 + x} = (1 + x)^{\frac{1}{2}} = 1 + \frac{x}{2} - \frac{x^2}{8} + \cdots \tag{7.4}$$

we can rewrite r as a binomial (with a remaining z factor), like this:

$$r = z(1 + \frac{\rho^2}{z^2})^{\frac{1}{2}} = z + \frac{\rho^2}{2z} - \frac{\rho^4}{8z^3} + \cdots \tag{7.5}$$

For terms after the second one to be negligible, the third term has to be much smaller than the period of the complex exponential of the diffraction equation:

$$\frac{\rho^4}{8z^3} \ll 2\pi \tag{7.6}$$

This condition requires z to be very high, and/or ρ to be very low.

Now, this condition doesn't account for wavelength yet, so we can look, not just at r, but kr from the original equation instead:

$$E(x,y,z) = \frac{1}{i\lambda}\iint_{-\infty}^{+\infty} E(x',y',0)\,\frac{e^{i\,kr}}{r}\frac{z}{r}dx'dy' \tag{7.7}$$

$$k\frac{\rho^4}{8z^3} \ll 2\pi \iff \frac{2\pi}{\lambda}\frac{\rho^4}{8z^3} \ll 2\pi \tag{7.8}$$

which can be reduced to:

$$\frac{\rho^4}{z^3\lambda} \ll 8 \tag{7.9}$$

If we do the calculation in nanometers, λ impacts very little. We can assume the wavelengths we are dealing with are many orders of magnitude smaller than the measurements of the aperture or the distance to the retina: hundreds of nanometers for light waves, versus several millimeters for eye structures. We have to consider wavelengths up to 800nm, ρ values between 0 and 5mm maybe, and around 25mm for z, and since ρ and z get powered by 3 and 4 respectively, λ can be neglected in the expression, so the remaining validity condition is:

$$\rho \ll z \tag{7.10}$$

This enables us to make one further approximation, that is to treat the r in the denominator as being roughly equal to z, since r is the hypotenuse and z is its adjacent side, and thanks to the high z distance flattening the angle and its cosine, thus making r and z roughly equal :

$$E(x,y,z) = \frac{1}{i\lambda}\iint_{-\infty}^{+\infty} E(x',y',0)\,\frac{e^{ikr}}{r}\frac{z}{r}dx'dy' \tag{7.11}$$

$$E(x,y,z) = \frac{1}{i\lambda}\iint_{-\infty}^{+\infty} E(x',y',0)\,\frac{e^{ikr}}{z}dx'dy' \tag{7.12}$$

The validity of the approximation having been established, we can replace the remaining occurrence of r by the new value:

$$r \approx z + \frac{\rho^2}{2z} = z + \frac{(x-x')^2 + (y-y')^2}{2z} \tag{7.13}$$

The remaining occurrence is in the complex exponential:

$$E(x,y,z) = \frac{1}{i\lambda}\iint_{-\infty}^{+\infty} E(x',y',0)\,\frac{e^{ik\,r}}{z}dx'dy' \tag{7.14}$$

What follows is some of the terms being re-arranged, we can move $\frac{1}{z}$ in front of the integral:

$$E(x,y,z) = \frac{1}{i\lambda}\iint_{-\infty}^{+\infty} E(x',y',0)\,\frac{e^{ik(z+\frac{(x-x')^2+(y-y')^2}{2z})}}{z}dx'dy' \tag{7.15}$$

We can distribute the terms in the exponent of e:

$$E(x,y,z) = \frac{1}{i\lambda z}\iint_{-\infty}^{+\infty} E(x',y',0)\,e^{ik(z\,+\,\frac{(x-x')^2+(y-y')^2}{2z})}dx'dy' \tag{7.16}$$

We can move the term e^{ikz} in front of the integral:

$$E(x,y,z) = \frac{1}{i\lambda z}\iint_{-\infty}^{+\infty} E(x',y',0)\, e^{ikz} e^{ik\frac{(x-x')^2+(y-y')^2}{2z}} dx'dy' \tag{7.17}$$

$$E(x,y,z) = \frac{e^{ikz}}{i\lambda z}\iint_{-\infty}^{+\infty} E(x',y',0)\, e^{ik\frac{(x-x')^2+(y-y')^2}{2z}} dx'dy' \tag{7.18}$$

We now have the Fresnel approximation equation.

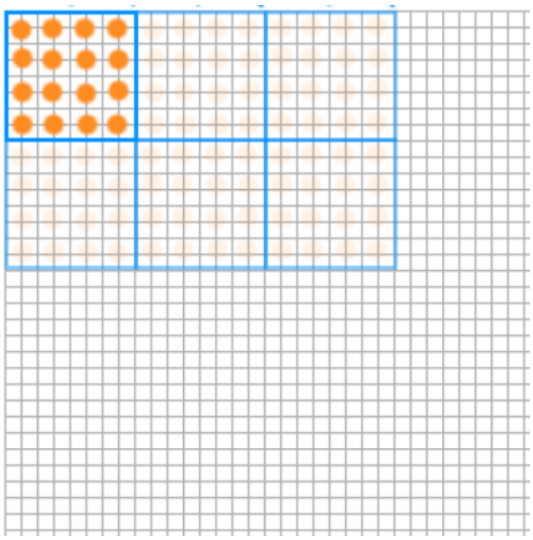


Figure 7.1: The grey grid is the array being worked on. The areas outlined in blue represent the coverage of each thread-group. Each orange dot represents one thread which treats 4 cells.

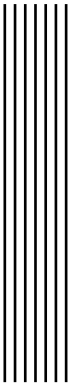
7.2 Compute shaders

A shader is a script that runs on a GPU: typically vertex and fragment shaders perform rendering operations. A compute shader however is a script that lets us run code that isn't necessarily rendering-related on the GPU. The point of using the GPU as a non-graphics processing unit is that it's very good at processing large amounts of similar data in parallel, so running on large arrays that can be split-up is the main use case for compute shaders.

We call functions inside compute shaders "kernels", and they're typically written to target array chunks instead of entire arrays. These chunks are defined by a group of cell indices generated when the kernels are "dispatched". Kernel functions aren't just run once per call, instead, many groups of threads get dispatched to perform several runs of our kernel, each thread-group working on a separate chunk of the data.

A typical example: if we're doing an operation on the pixels of a texture that is 512x512, our kernel might run on chunks of 16x16 texels, which cuts up the work load into 1024 chunks, which means 1024 thread-groups will be dispatched to run the kernel, one group for each of those chunks. It depends on how many threads our specific GPU can run in parallel, but if it can run 1024 groups of 256 threads each in parallel on those chunks of that size, we've just sped up the code 1024x256 times (in a simplified ideal scenario), compared to a non-parallel loop that iterates over every pixel individually.

The last main thing to explain is the fact the data the compute shaders run on has to be in GPU memory/VRAM, and not CPU memory/RAM. That has upsides and downsides: if we really need the data to go back and forth between RAM and VRAM, we're going to need to transfer it, which is generally very slow; but if we're in a scenario where we don't have to send data back and forth a lot, then generating data in VRAM directly with a compute shader for exclusive use in later rendering will save a lot of performance.



Bibliography

[1] T. Ritschel, M. Ihrke, J. R. Frisvad, J. Coppens, K. Myszkowski, and H.-P. Seidel. "Temporal Glare: Real-Time Dynamic Simulation of the Scattering in the Human Eye". en. In: *Computer Graphics Forum* 28.2 (Apr. 2009), pp. 183–192. ISSN: 01677055, 14678659. DOI: 10.1111/j.1467-8659.2009.01357.x. URL: https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01357.x (visited on 07/26/2021).

[2] Greg Spencer, Peter Shirley, Kurt Zimmerman, and Donald P. Greenberg. "Physically-based glare effects for digital images". en. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques - SIGGRAPH '95*. ACM Press (1995), pp. 325–334. ISBN: 978-0-89791-701-8. DOI: 10.1145/218380.218466. URL: http://portal.acm.org/citation.cfm?doid=218380.218466 (visited on 07/26/2021).

[3] Masanori Kakimoto, Kaoru Matsuoka, Tomoyuki Nishita, Takeshi Naemura, and Hiroshi Harashima. "Glare Generation Based on Wave Optics". en. In: *Computer Graphics Forum* 24.2 (June 2005), pp. 185–193. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/j.1467-8659.2005.00842.x. URL: https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2005.00842.x (visited on 07/26/2021).

[4] Max Born, Emil Wolf, and A. B. Bhatia. *Principles of optics: electromagnetic theory of propagation, interference, and diffraction of light*. Seventh (expanded) anniversary edition, 60th anniversary edition. Cambridge: Cambridge University Press (2019). ISBN: 978-1-108-47743-7.

[5] Joseph W. Goodman. *Introduction to Fourier optics*. 2nd ed. McGraw-Hill series in electrical and computer engineering. New York: McGraw-Hill (1996). ISBN: 978-0-07-024254-8.

[6] Eihachiro Nakamae, Kazufumi Kaneda, Takashi Okamoto, and Tomoyuki Nishita. "A lighting model aiming at drive simulators". en. In: *ACM SIGGRAPH Computer Graphics* 24.4 (Sept. 1990), pp. 395–404. ISSN: 0097-8930. DOI: 10.1145/97880.97922. URL: https://dl.acm.org/doi/10.1145/97880.97922 (visited on 07/26/2021).

[7] Mikio Shinya, Takafumi Saito, and Tokiichiro Takahashi. "Rendering techniques for transparent objects". In: *Proc. Graphics Interface*. Vol. 89 (1989), pp. 173–181.

[8] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-time Rendering*. 4th ed. K Peters/CRC Press (2018).

[9] T. Ritschel. “Using Simulated Visual Illusions and Perceptual Anomalies to Convey Dynamic Range”. en. In: *High Dynamic Range Video*. Elsevier (2016), pp. 209–235. ISBN: 978-0-08-100412-8. DOI: 10.1016/B978-0-08-100412-8.00008-5. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780081004128000085 (visited on 07/29/2021).

[10] Yoshiharu Gotanda, Masaki Kawase, and Masanori Kakimoto. “Real-time rendering of physically based optical effects in theory and practice”. en. In: *ACM SIGGRAPH 2015 Courses*. Los Angeles California: ACM (July 2015), pp. 1–14. ISBN: 978-1-4503-3634-5. DOI: 10.1145/2776880.2792715. URL: https://dl.acm.org/doi/10.1145/2776880.2792715 (visited on 07/26/2021).

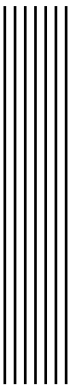


Image credits

[11] Figure 1.1. Battlefield 4: Official 17 Minutes "Fishing in Baku" Gameplay Reveal. URL: https://www.youtube.com/watch?v=U8HVQXkeU8U.

[12] Figure 1.2. Temporal Glare: Real-Time Dynamic Simulation of the Scattering in the Human Eye (Eurographics 2009 Supplemental Video). URL: https://www.youtube.com/watch?v=5ewKMOodT1Y.

[13] Figure 2.6. Diffraction pattern from wavelength-wide aperture. URL: https://en.wikipedia.org/wiki/File:Wavelength%3Dslitwidth.gif.

[14] Figure 2.7. Diffraction pattern from four-wavelengths-wide aperture. URL: https://en.wikipedia.org/wiki/File:Wave_Diffraction_4Lambda_Slit.png.

[15] Figure 2.8. Simplified optical system for diffraction modeling. URL: https://en.wikipedia.org/wiki/File:Diffraction_geometry.svg.

[16] Figure 2.9. Huygens-Fresnel principle. URL: https://commons.wikimedia.org/wiki/File:HuygensDiffraction.svg.

[17] Figure 2.10. Double-slit diffraction pattern. URL: https://en.wikipedia.org/wiki/File:Young_Diffraction.png.

[18] Figure 2.11. Color Matching Functions for wavelength to XYZ. URL: http://cvrl.ucl.ac.uk/cmfs.htm.

[19] Figure 2.24. CIExy 1931 sRGB Chromaticity diagram. URL: https://commons.wikimedia.org/wiki/File:CIExy1931_sRGB.svg.

[20] Figure 2.25. ACES2065-1 CIE 1931 2 Degree Standard Observer - CIE 1931 Chromaticity Diagram. URL: https://community.acescentral.com/t/understanding-gamut-with-aces-blender/1832/2.