OpenLensFlare: an Open-Source, Lens Flare Designing and Rendering Framework

István Csoba
University of Debrecen
Faculty of Informatics
Debrecen, Hungary
csistvan94@gmail.com

ABSTRACT

Lens flare rendering in computer games has always been lacking. While physically motivated solutions based on real camera systems exist, their inherent complexity makes them inappropriate for widespread use. Developers are not trained to work with these complicated imaging systems, and even if they were, artists and designers prefer intuitive parameters over these complicated optical system descriptions. More support is needed to start adopting to these advanced models, and to show that with an appropriate tool the situation can be vastly improved. This paper describes OpenLensFlare, an open-source C++ framework for rendering convincing physically-based lens flare effects. Additionally, a supporting optical system editor is also provided, which is capable of producing and visualizing optical systems and showing an example usage of the run-time library. Together, these two systems can be used to replace the existing naive algorithms, with small effort and low integration complexity.

Keywords

Lens Flare Rendering, Optical Systems, Lens Editor, Open-Source Software

1 INTRODUCTION

Rendering realistic and plausible images has been a hot topic of research ever since the birth of computer graphics. A subset that is of special interest, which has been gaining popularity in recent years, is proper simulation of the aberrations in a camera system. Working with real optical systems is a challenging task, since it is hard to see what happens to a ray once it enters the lens housing. Supporting tools are needed to aid working with these systems.

This work focuses on one particular aberration, which is often referred to as lens flare. Lens flare is a phenomenon in photography that is due to the incoming light taking undesired paths while passing through the lens system of the objective. The two most prominent effects that lead to the birth of these flares are the internal reflections between the lens elements which causes ghost patches to appear, and the diffraction of the incoming light that is most visible on the starburst patterns at the locations of bright light sources on the image.

Although lens flare is a very subtle effect, its presence is just as important, since it plays a very important role in keeping up the illusion of a non-simulated world. Rendering these effects in real-time, at interactive frame rates is a problem that has existing solutions, but they tend to be very complex and do not enjoy widespread adoption. Softening their limitations is another open problem today, and one that demands more research work to be done.

In this paper an OpenGL-based C++ lens flare rendering library is introduced, named OpenLensFlare. Designed to be modular and extensible, it can serve as both a starting point into learning more about these recent advancements, but may also be used as the rendering solution of real-time applications. The library provides all the components needed to render plausible lens flare effects, while at the same time also granting the required customization and freedom.

A multipurpose supporting tool is also described, capable of not only managing the accompanying data of the run-time library, but also visualizing the various properties of the lens systems and their generated lens flare. The editor also serves as an integration example and can be used to further simplify the process of using lens flare for enhancing the visual quality of our renderings.

The main goal of this work is to help with the adoption of advanced lens flare rendering methods, while at the same time also providing an environment that is useful for designing and testing future, improved versions of the algorithms in question.

The paper is structured as follows: Section 2 gives an overview of similar tools and summarizes the results of the research done in the field of lens flare rendering. In Section 3 the rendering algorithms implemented in the framework are described in detail. Section 4 outlines the main features of the proposed framework. Section 5 describes the design choices of the library and explains some of the potential issues of implementing such a library. Section 6 discusses the various validations and testings performed on the implementations. Lastly, Section 7 sums up this work and talks about possible future improvements for the library.

2 RELATED WORK

2.1 Special Effect Frameworks

Code reusability has always been one of the core principles in computer programming. The advantages of using a well tested system are nearly endless. Real-time rendering is an area where this is especially true, thanks to the poor debugging support and vast amount of graphics APIs that should be supported. This section outlines a few existing frameworks, each similar in spirit to the proposed solutions.

The GameWorks [1] initiative by NVIDIA is one of the largest library collections, aimed at providing game developers with the best material needed for creating their products. The development kit has a whole section devoted to rendering, named VisualFX, covering a wide range of effect rendering needs for most games. These implementations have been used in many games throughout the years, with great success and positive feedback from the developers [6].

What the GameWorks libraries lack though is proper tooling support. The components themselves are well documented, but managing their settings can be unintuitive at times, making it hard to use them correctly. More focused frameworks come with easy-to-use editors, debuggers and visualizers, allowing for a better user experience. Examples of these systems include PopcornFX [11] and the Houdini tools [8]. Although the complexity of these specialized frameworks is much higher, developers still have an easier time creating the desired effects, all thanks to the superior editing capabilities of their accompanying toolsets.

2.2 Other Related Software

Tooling support for working with real camera systems is lacking. The only existing software that is relevant to this field is OpticStudio [16], which is a professional optical system editor for optical engineers, developed by Zemax. Despite its good camera visualization capabilities, usage of an external tool that is so loosely coupled with our system is also a tedious process. Furthermore, since OpticStudio was designed with a different set of features in mind, it lacks any rendering support, making it an inappropriate choice for simulating lens flare.

2.3 Lens Flare Algorithms

Lens flare can be divided into two main components: a starburst pattern, that is due to the diffraction that occurs when the light passes through the aperture of the imaging system, and a varying number of ghosts, each of which corresponds to a unique sequence of inter-reflections between the optical elements. This section gives a broad overview of the most recent advancements in rendering these effects properly.

Texture Composition The most basic approach to the problem is additive composition with pre-rendered ghost and starburst textures. The starburst texture can be positioned at the center of the light source on the screen, while the ghosts are usually placed on a line that passes through it. Although not physically motivated, the algorithm is still often favored, all thanks to its low rendering and integration costs.

Starburst Texture The look of these starburst textures can be improved greatly by taking into consideration the diffraction that occurs in the optical system. Algorithms describing how to do this for the human eye exist [12], and an analogue technique can be implemented for camera systems as well. In fact, the most advanced starburst rendering method invented by Hullin et al. [4] is still based on texture rendering, but instead of artificially generating something that looks similar to a real-world effect, they attempt to mimic the looks of the phenomena by manual texture composition, as dictated by the underlying diffraction theorem.

Ghost Patches A more sophisticated ghost rendering solution can be given by involving a real camera system. Efficiently rendering with such a system has been a topic of hot research lately [5, 2, 13, 17], and has been the main idea of the work done by Hullin et al. [4] and Lee et al. [10] in the field of lens flare ghost image rendering. The premise of their invented algorithms is that today's graphics units are very well suited for tracing a vast amount of rays in parallel, under heavy time constraints. Therefore, images that closely match these ghost effects can be rendered by constructing an environment appropriate for imitating a real imaging system.

3 MATHEMATICAL BACKGROUND

In the last section we have already seen the core ideas behind the related lens flare rendering algorithms. This section gives the reader the required definitions and principles to understand design choices and inner workings of the proposed framework.

3.1 Rendering Starburst Patterns

One of the two parts that make up the lens flare effect is the starburst pattern. The main source for this interesting flare shape is from the diffraction that occurs when the incoming light passes through the iris aperture in the optical system, causing the light to bleed into otherwise shadowed areas.

To render a plausible starburst effect, a pregenerated texture is drawn at the location of each bright light source on the image, using the f-number of the camera to control the size and intensity. As shown by Ritschel et al. [12] and Hullin et al. [4], this texture can be generated by using the far field Fraunhofer diffraction formula [3]. For this first a mask image $T(x,y)$ needs to be provided, which represents the iris shape. From this mask the corresponding Fourier power spectrum $F(u,v)$ is computed by taking its squared Fourier transform. The starburst texture $S(x,y)$ is then generated by iterating through the visible color spectrum and blending together the corresponding scaled copies of the power spectrum as:

$$S(x,y) = \sum_{i=0}^{n-1} \lambda_i F(u_i, v_i)$$
$$\lambda_i = \lambda_s + i\frac{\lambda_e - \lambda_s}{n}$$
$$(u_i, v_i) = (u,v)\frac{\lambda_s}{\lambda_i}$$

where λ_s, λ_e and n are user defined parameters for the starting, ending wavelengths (in nanometers) and the number of wavelength steps, respectively. Note that the way (u_i, v_i) is computed slightly derives from what Ritschel et al. suggested, but that is because they used a mixture of Fraunhofer and Fresnel diffraction, however as they have also shown, the quality gains are marginal and thus very similar outputs can be rendered by only relying on the Fraunhofer formula, as presented above.

3.2 Rendering Ghosts

To render convincing ghost flares, a set of rays have to be traced through the optical system. As the time needed to perform this would be immensely long, some simplifications and optimizations are needed to make it a suitable method for real-time applications. Hullin et al. showed that this is indeed possible to do [4] with some carefully made considerations. Their method relies on the fact that rays originating from the front element reach the sensor in a continuous manner, so instead of tracing a dense set of rays through the system, a sparse set of rays suffices just as well, with some interpolations.

The algorithm starts by enumerating the various light paths that each correspond to a single ghost patch on the image. Knowing that the ray is propagating forward, this can be done by listing the optical system indices from which the light is reflected, and assuming that transmission on every other surface touched by the ray. With the unique paths computed, the ghosts are rendered independently, one after the other.

To perform the ray-tracing, the lenses are virtually treated as analytic surfaces (spheres and planes) in 3D. The interfaces are then iterated over one by one, taking the intersection of the traced ray and the optical element, computing the normal and angle of incidence, and continuing with the reflected and refracted ray, as needed. Once all the rays are traced through the system and their positions on the sensor is computed, the sparse grid is triangulated and filled by interpolating the various attributes that were generated while following the ray, such as the ratio of reflected energy.

To get the intensity of the ghost, the Fresnel reflectivity equation for unpolarized light can be used [3]:

$$R = \frac{1}{2}\left(\frac{n_1 cos\theta_1 - n_2 cos\theta_2}{n_1 cos\theta_1 + n_2 cos\theta_2}\right)^2 + \frac{1}{2}\left(\frac{n_1 cos\theta_2 - n_2 cos\theta_1}{n_1 cos\theta_2 + n_2 cos\theta_1}\right)^2$$
$$T = 1 - R$$

where n_1 and n_2 corresponds to the refractive indices of the two participating media, and θ_1 and θ_2 are the angles of light in the two media relative to the normal. As mentioned above, this equation is evaluated per ray at each interface where the light undergoes reflection, and multiplied together to compute the ratio of the reflected light amount.

To consider dispersion, a wavelength-dependent refractive index has to be computed and used during the ray-tracing process. The lens patents typically only contain the index of refraction at the Fraunhofer D-line (n_d) and the corresponding Abbe-number (V_d). Considering the definition of the Abbe-number:

$$V_d = \frac{n_d - 1}{n_F - n_C}$$

and knowing that λ_D, λ_F, and λ_C are fixed, substituting these into the two-term form of Cauchy's equation we get the following:

$$n(\lambda) = 1.50459 + \frac{\lambda}{0.004215}$$

which can be used to provide the wavelength-dependent refraction indices (note that the wavelengths are measured in micrometers). Using these indices, the final ghost images are rendered by performing the ray-tracing process for multiple wavelengths, as outlined above, and blending the results together.

4 PROPOSED SOLUTION

The main issue of integrating a physically-based lens flare rendering algorithm into our application is the requirement of a proper optical system prescription. Such prescriptions are hard to both acquire and edit, which is why even the largest companies have stayed away from adopting them. Artists and game designers cannot be expected to work with these complex systems, and so it is a reasonable move on the developers' side. But the situation is not hopeless, as with proper support even the largest systems become manageable.

This work focuses on a framework that is capable of rendering proper lens flare and managing all the data needed to achieve that goal. The framework is made up of two components: a run-time library implementing the rendering algorithms and holding the corresponding data, and an editor tool to design custom optical systems and lens flare effects. An example image of this editor in action can be seen on Figure 1. This section explains the various parts and features of the proposed framework and gives an overview of how it attempts to ease the process of using these large optical systems for creating custom lens flare.

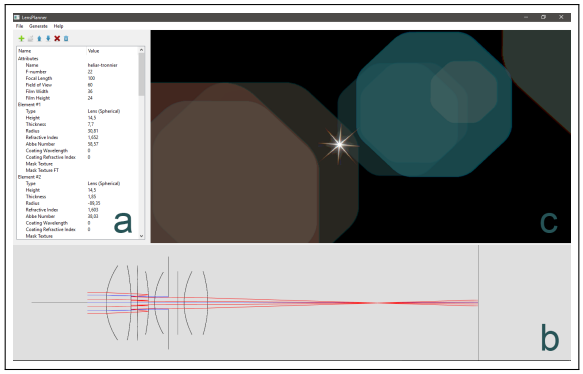

Figure 1: *Interface of the editor component. (a) optical system editor; (b) optical system visualizer; (c) preview lens flare rendering.*

4.1 Optical System Design

To render physically-based lens flare effects, first the description of a real imaging system has to be supplied. This description should include the various parameters of each of the lens interfaces present in the system, such as the material properties, physical height and radius of curvature of the interface, but it should also contain attributes global to the entire system, such as the focal length of the system, or the dimensions of its sensor.

These descriptions may be acquired from patent databases or special books [9, 14], however, designing a custom lens system is often desired, to improve the looks of the generated imagery. Since tweaking values in a text file or directly in the source code to achieve this would be a painful process, the proposed framework contains a component designated solely for editing these systems and exploring the characteristics of their generated flare effects by visualizing the optical system, the flare and various other aspects that contribute to the birth of the phenomenon. Figure 1 shows what this component looks like in action, using a real camera system to show the proposed editor.

To manage the aforementioned parameters, the run-time library contains a special object designed to hold the various optical system attributes. The lens editor component exposes these to the user in a tree structure, which contains both the global imaging parameters and all the lenses with their accompanying descriptions. Figure 2 shows a closeup image of the tree widget as it looks in the editor. The user is then able to assign custom values to these parameters, and immediately see the effects of the changes on the visualization widgets. The process of creating an optical system that has the desired imaging properties is vastly simplified this way. Once finished, the optical system can be exported and later reloaded by both the designer component and the rendering library.

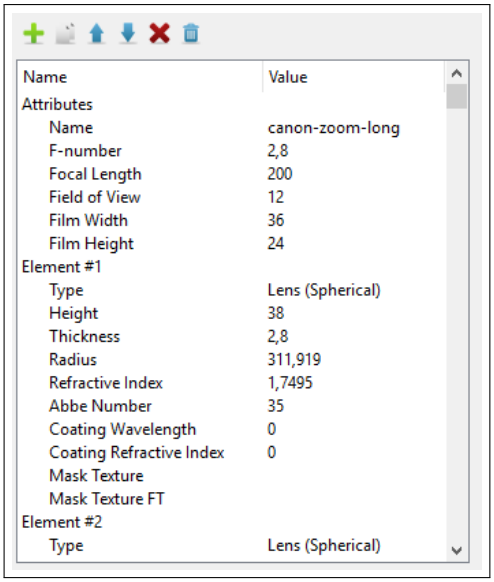

Figure 2: *The optical system editor widget of the lens designer tool. This is used to create and edit the lens system used for rendering lens flares.*

4.2 Data Compilation

To achieve acceptable rendering performance, some preliminary computations need to be performed. For most algorithms, these do not take a significant amount of time and can be carried out in the initialization step of the application. However, computing everything that is needed for rendering lens flare can easily take as long as an hour, even for a camera of average complexity.

Preparing all this data with an external tool is thus desired, so that they can be just loaded in at run-time, when they are needed.

OpenLensFlare was designed with pre-computation in mind. All the algorithm implementations support extracting the data they have computed and loading them back later. Thus, a full optical system coupled with ghost and starburst rendering parameters can be prepared and loaded in a matter of milliseconds. Everything is provided in native C++ and OpenGL objects, meaning the user is not locked to a single serialization strategy, but may choose the used methods as they will.

The editor contains a reference implementation for saving and loading the desired objects. The provided solution can be used to serialize the whole optical system and the rendering algorithm states - along with all of their generated data - into a set of XML and image files. This way the lens attributes can be calibrated once, and either used by us, in our product, or shared with others. Listing 1 shows the structure of one of these XML files - the one describing the optical system. Using the code provided with the framework, the user has out of the box support for generating run-time data and injecting it into their software.

```
<?xml version="1.0" encoding="UTF-8"?>
<opticalSystem>
  <name>ExampleOpticalSystem</name>
  <fnumber>11</fnumber>
  <!-- Other camera attributes... -->
  <elements>
    <element>
      <type>lensSpherical</type>
      <height>24.0</height>
      <thickness>9.6</thickness>
      <radius>65.220001</radius>
      <!-- Other lens attributes... -->
    </element>
    <!-- More elements...-->
  </elements>
</opticalSystem>
```

Listing 1: *Structure of an XML file that describes an optical system, as generated and understood by the provided reference serialization implementation.*

4.3 Rendering

Understanding an optical system is only half of the problem. We still need to find a way to simulate how the ghosts would be formulated on the captured image, and render one that looks similar to that of the original. As it turns out the problem is so complex that existing algorithms are very hard to grasp and providing a fully functional implementation is troublesome.

The proposed framework tries to ease the process of generating eye-catching lens flare imagery by giving the developers the necessary tools to render ghosts and starburst images. All the rendering algorithms reside in the lower, run-time layer. Currently the framework implements the algorithms presented by Hullin et al. [4], as described earlier in Section 2. The user is free to select, configure and combine these algorithms in any way that fits their requirements, allowing to adopt to the running environment.

The rendering aims to be fully customizable. It is possible to specify intensity scaling factors, list of ghosts to render, ray grid resolutions, anti-reflection coating parameters, ray culling factors and iris shape smoothing. Furthermore, the library expects the rendering context to be pre-configured, thus enabling the use of custom render targets and blend modes.

Large optical systems can imply high rendering times, which might be unacceptable for certain applications. The library has options for resolving this issue. Acceleration can be achieved by using the output of the precomputation process discussed in Section 4.2. The generated ghost data can be edited to reduce the number of rays used for rendering, decreasing the overall pressure on the graphical unit. Additionally, intensity and bounding information can be used to cull ghosts with a low impact on the render, saving additional time for other tasks.

4.4 Visualization

In computer science, visualization of the data is a key step of understanding a problem. Data that might take the developer days to fully learn, can easily be perceived in a matter of hours with the use of images. That being said, data visualization has been one of the most important motivating factors for making this framework.

As it has been outlined above, lens prescriptions are hard to understand. For the untrained eye, their parameters are totally meaningless and difficult to fully grasp. To help us better understand these raw camera parameters, the editor provides a view of the lenses and their relations to one another. First the optical axis is rendered in the center of the viewport. The elements are then drawn on top of it one by one, as either a simple section (flat surfaces and sensor), an arc (spherical surfaces) or a section with a hole in the middle (iris aperture). The elements themselves are not connected together, since the optical system descriptions do not contain the information needed to reassemble the physical lenses from the provided surface parameters.

The usefulness of this optical system sketch is further improved by showing how a set of example rays travel through the system, with or without involving interreflection paths between the lens elements. Each ray is traced through the whole optical system independently, computing and storing their intersections with each interface and updating the ray directions by following the

reflected or refracted ray, as needed. The intersection points that correspond to the same ray are then connected, forming a set of lines that either reach the sensor or show the points where their rays are fully blocked or miss an interface. This kind of exploration of the optical systems is a common practice among optical engineers as well, since it allows to better examine the performance and imaging characteristics of the system. Figure 3 shows an example output generated with the proposed lens editor, visualizing rays that correspond to normal, and reflected ray paths, respectively.

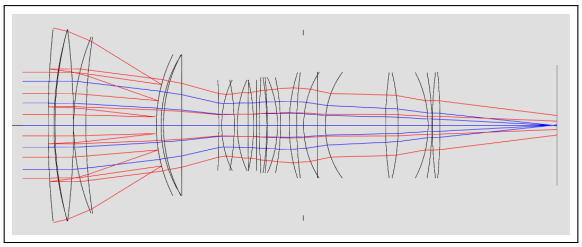

Figure 3: *Example visualization of rays taking the normal path, converging in a single point on the sensor (blue) and rays corresponding to ghost, spreading out on the camera sensor (red).*

Another important aspect of data visualization is previewing the output of rendering lens flare with our optical system. Immediate feedback of the changes done to the lenses is very important, as the whole purpose of the work done on designing these lens systems is to use them for lens flare rendering. To provide the user this wisdom, the editor component has an embedded flare previewer widget that is updated on the fly as the optical system is being modified, to display what the flares generated by the current configuration would look like. This preview rendering is performed by the run-time library component, to make sure the looks of the sample render fully matches the final image in the embedding application.

Finally, there are some other characteristics of both the camera system and the rendering algorithms that are worth exploring. Examples of these include the bounds on the front element that correspond to rays reaching the sensor, or the UV coordinates of the rays when passing through the aperture iris. An example output of such a simulation can be seen on Figure 4, which visualizes the aforementioned front element bounds. While these may not always have a visual impact on the rendering, they usually affect the average render times of the system. The run-time library can render this data directly on its own, by only setting a few flags on the rendering objects. This way these visualizations can not only be shown in the editor, but are also available in the embedding product. Such hidden information not only allows the designer improve the run-time performance of the rendering, but also helps developers to perform debugging tasks and better understand how a specific ghost image is rendered.

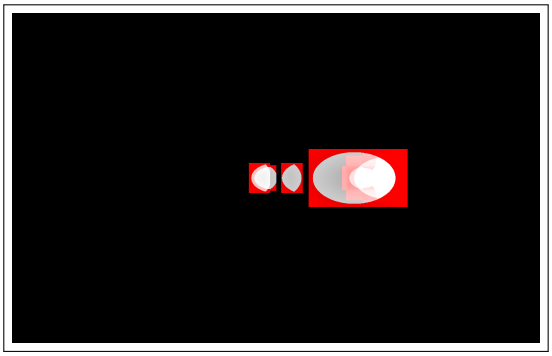

Figure 4: *Visualization of rectangles on the front element bounding the rays that can reach the sensor. The black section corresponds to the front element surface, on which each colored rectangle denotes a single ghost (note that they may overlap). It is clearly apparent that limiting the ray-tracing to these areas can vastly improve the quality and performance of the ghost renderings.*

5 IMPLEMENTATION

The main goals of the framework have been discussed in the previous section. The following subsections provide some insight into the design process that the library went through and some of the issues with alternative solutions that have been tested.

5.1 API Design

The library was designed with extensibility in mind and aims to provide an environment suitable for implementing any physically-based lens flare algorithm. The fundamental terms are all fully represented with separate classes describing all of their relevant attributes. Examples of these core abstraction classes include optical systems with their elements, ghosts, ghost enumerators and abstract light sources.

The rendering implementations are all built on top of these foundation classes. They fall into two main categories, which correspond to the division of the phenomena described in Section 2.3. The user is free to choose any number of these as they will. It is possible to distribute the rendering of the same set of ghosts between multiple instances, by using a heuristic, for example.

The library also provides a tight abstract interface for the algorithms. These can be used where starburst or ghost rendering is required, without having to refer to concrete algorithm classes after initialization. This feature proved useful when implementing the visualizer components, and should also simplify the customization process for end-users as well. An usage example can be seen on Listing 2.

```
// Lens system for rendering, assumed
// to be loaded by the application
OpticalSystem* pSystem =
    loadOpticalSystem();
// Initialize a diffraction starburst
// algorithm, with a size of 0.1
// and intensity of 1.0
DiffractionStarburstAlgorithm* sb = new
    DiffractionStarburstAlgorithm(pSystem
    , 0.1f, 1.0f);
// Create a 1024x1024 starburst texture
// that merges scaled copies from 380
// to 780 nm, using a step size of 5 nm
sb->generateTexture(1024, 1024, 380.0f,
    780.0f, 5.0f);
// Create a ray traced ghost renderer
// that renders all ghosts and traces
// the parameter wavelengths
RayTraceGhostAlgorithm* ghost = new
    RayTraceGhostAlgorithm(pSystem,
    GhostList(pSystem), { 650.0f, 510.0f,
     475.0f }, 1.0f);
// Compute optimal rendering parameters
ghost->computeGhostAttributes();
```

Listing 2: *An example C++ code, showcasing the initialization process of the rendering objects.*

5.2 Graphics API

The choice of graphics API is mostly irrelevant for a tool, but the supported APIs play a crucial role when selecting a rendering library. Since OpenGL is so widely supported, it seemed natural to use it for both the flare algorithms and the editor interface. All objects – including the algorithms and the optical system descriptions – expect OpenGL handles when working with objects in their interfaces. The rendering implementations also make heavy use of OpenGL and require a valid context to be active when calling any of their rendering functions.

An alternative solution is the usage of an API abstraction layer. While it has the benefit that in theory it could work with every API in existence, it is not true in practice. There are holes in the feature matrix, and certain object types are not available everywhere. Another possibility is a third party library, of which the most commonly chosen one is bgfx [7]. While it could solve some the aforementioned issues, it has another set of deficiencies, such as the lacking capability of generating shaders at run-time, which is required to function correctly by some of the planned algorithms.

5.3 Shaders

Almost every external resource is configurable in the interface, but the use of shaders needed special attention. Since a large section of the rendering logic resides in shaders, they are long enough to require separation into their own files. Accessing them is not as trivial as it may first seem, since their presence should ideally be hidden in the interface. Additionally, since a rendering library should function properly with limited file system privileges, direct file system access is also ruled out, further complicating the situation.

A different approach has been taken to solve the problem. Instead of accessing files at run-time, the shader sources are compiled into the binary files upon compilation, as raw string literals. When using these shaders, the implementations can directly refer to their source codes in a special namespace, with the same name as they were on the disk before compilation. This way the application can run under limited file system access rights and the library remains functional, saving the user from the trouble of distributing and managing external file dependencies.

5.4 Rendering

Rendering of the starbursts and ghosts is done on-demand, when initiated by the user. The embedding application is expected to configure the rendering objects, as described in 5.1, and call their appropriate methods to perform the rendering. These objects rely on embedded shaders as described in the previous section, and assembles them under the hood into the required OpenGL shader programs.

Before rendering with the optical system, the lens parameters are locally transformed into an optimal format that is faster to access during rendering. These values are then uploaded to the GPU into a set of uniform arrays, consumed by the corresponding shader programs. The rendering is then carried out by issuing the appropriate draw calls for the selected ghosts and starbursts. The ray-tracing needed for ghost rendering is performed in the vertex shader and finalized in a geomery shader, outputting triangles to the hardware rasterizer for automatic interpolation.

5.5 Lens Editor

The lens editor tool is based on Qt 5 [15]. An alternative and natural solution would be using an immediate mode GUI library, but it proved to be not suitable for the task. The pre-computations discussed in Section 4.2 allow for important optimizations, but do not work very well with rapid changes to the optical system. To display a meaningful image takes a significant amount of time without examining the data, which needs to be repeated every frame with immediate mode interfaces. The interface thus becomes unusable, creating a horrible user experience.

6 RESULTS

Validation A test scene has been created to verify the library's applicability and its appropriate performance.

Many different camera systems and render parameters were experimented with, to check that the library is capable of working with imaging systems of varying complexity. Figure 5 shows an example output of a starburst and many ghosts rendered with the proposed library and composited onto the test scene, all in real-time, allowing for user interaction.

Figure 5: *Lens flare generated by a complex Nikon lens system, rendered with OpenLensFlare and composited onto the test scene.*

The rendering algorithms are all based on already existing methods, well tested by their own authors. Thus, only the correctness of the implementation had to be verified. This was done by reproducing the environments used to render the sample images provided in the original papers, and manually comparing my results to the reference renderings. The generated starburst texture, also the shape and intensity of the rendered ghosts seemed to closely match the reference images, proving that the implementations are correct. The only difference is in the colouring of the ghosts, which is due to the anti-reflective coating parameters used by the authors that cannot easily be reverse engineered. However, OpenLensFlare gives the user the option to assign these values to each lens, making it possible to produce images that also match the colors of the reference ghost rendering, and also create custom setups as desired.

Performance Performance of the library was tested with the lens system mentioned above. Table 1 shows the average render times for a Heliar (US2645156, 8 reflective interfaces, 13 ghosts rendered) and a Nikon (JPS53131852A, 21 reflective interfaces, 142 ghosts rendered) lens system (both if which can be found in Smith's book [14]), with and without involving a precomputation step. The tests concluded that both the algorithms and the implementations are suitable for real-time applications, and even a complicated optical system with many lenses can be simulated at interactive frame rates.

Finally, performance of the precomputation process has been also measured. Usage of pregenerated rendering parameters - such as optimal number of rays or ray bounding on the front element - is crucial for achieving both fast render times and convincing renderings. OpenLensFlare provides a custom algorithm for approximating them in real time, and also implements a slower, full-blown ray-traced parametrization process, which - once editing the imaging system is finished - can be used to compute the final values, and export them as outlined in Section 4.2. The parameters used during this process can also be customized, which directly affects not only the quality of the generated data, but more importantly, the run time of precomputation step. Table 2 summarizes the running times of both the approximate and precise methods with the lens systems used in the previous tests, with varying computation parameters. As it can be seen, a detailed ray-tracing solution is inappropriate for real-time lens modifications, however it is perfectly capable of preparing the final rendering parameters in a short amount of time, while the approximated values can be used throughout the process of editing the optical system.

	No Precomputation	Precomputation
Heliar	70,0 ms	4,8 ms
Nikon	458,4 ms	215,3 ms

Table 1: *Performance comparison of ghost rendering with a Heliar and a Nikon lens system, with and without pre-computated ghost bounding information.*

	Approximate	2 passes	3 passes
Heliar	0,006 s	8 s	9 s
Nikon	0,011 s	338 s	365 s

Table 2: *Run time comparison of the approximate and precise pre-computation processes, computing parameters for 181 different incident angles in total. The number of refinement passes denotes how many iterations were taken per angle per ghost when bounding the ghosts on the front element.*

Limitations As for limitations, the algorithms are currently implemented with the assumption that the optical system contains exactly one iris, which also has an accompanying mask texture. Additionally, while the shaders are well generalized for any number of reflections, currently they only consider the first two reflections of any ghost path. These limitations are easy to come around and are expected to be removed in the future, but to ease the implementation and management process, the decision was made to use these simplifications in the initial implementation, based to the observations of the authors of the rendering algorithms.

Furthermore, to keep the rendering times low, the algorithms currently treat each interface of the optical system as spherical. Also, only objectives with a field of view smaller than 90° were tested, since such a field

of view is used most commonly in computer games. While in theory the algorithms could handle all types of optical systems, the ray-tracing is performed with the assumption that rays reach each interface exactly once, in the order that they are given, which is often not true for other types of lens systems (e.g. fisheye lenses).

7 CONCLUSION

Lens flare rendering is often overlooked in rendering applications of the present. Developers tend to favor basic algorithms, due to their implementation and maintenance simplicity. However, the more advanced algorithms can be made to work in these time-constrained environments, and the efforts are well worth it. This paper described an open-source, physically-based lens flare rendering library and an accompanying editor capable of providing all the data needed for the runtime to function correctly. These together can be used to render physically-based lens flare in games and similar applications, but may also be useful for optical experts to explore the lens flares generated by their lens system designs. Hopefully, these efforts will make developers realize the potential lying in these algorithms and I am looking forward to seeing these effects incorporated into future applications.

As for areas for future work, the user interface of the editor tool needs more customization options for its visualization components. Namely, it is currently impossible to select in the software which ghosts should be previewed, and the looks of the sketch also cannot be altered. The implementation is capable of doing all of these tasks, and thus only the corresponding widgets need to be added, after a slight interface redesign.

Additionally, to make this framework a much more generic solution, implementations need to be added for other graphics APIs too. Furthermore, creating bindings to popular game engines could improve the library's accessibility. Lastly, implementing custom algorithms based on the solutions mentioned in Section 2.3 could further improve the visual quality of the renderings and give the user more options to choose from.

8 SUPPLEMENTARY MATERIAL

The source code of both the run-time library and the lens editor components is available for download, at https://github.com/luorax/OpenLensFlare.

9 REFERENCES

[1] COOMBES, DAVID. Cutting Edge Tools and Techniques for Real-Time Rendering with NVIDIA GameWorks. ACM SIGGRAPH, 2016.

[2] HANIKA, J., AND DACHSBACHER, C. Efficient monte carlo rendering with realistic lenses. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 323–332.

[3] HECHT, E. *Optics*, 4 ed. Addison-Wesley, 2002.

[4] HULLIN, M., EISEMANN, E., SEIDEL, H.-P., AND LEE, S. Physically-based real-time lens flare rendering. In *ACM Transactions on Graphics (TOG)* (2011), vol. 30, ACM, pp. 108:1–108:9.

[5] HULLIN, M. B., HANIKA, J., AND HEIDRICH, W. Polynomial optics: A construction kit for efficient ray-tracing of lens systems. In *Computer Graphics Forum* (2012), vol. 31, Wiley Online Library, pp. 1375–1383.

[6] JOHNSTON, D., MAHER, M., AND TOROK, B. The Witcher 3: Enabling Next-Gen Effects through NVIDIA GameWorks. Game Developers Conference, 2014.

[7] KARADZIC, B. bgfx [Programming library]. `https://bkaradzic.github.io/bgfx/`, 2017. accessed 13-03-2017.

[8] KEATING, SCOTT. Houdini 16 for Games. Game Developers Conference, 2017.

[9] LAIKIN, M. *Lens Design*, 2 ed. Marcel Dekker Inc, 1995.

[10] LEE, S., AND EISEMANN, E. Practical real-time lens-flare rendering. In *Computer Graphics Forum* (2013), vol. 32, Wiley Online Library, pp. 1–6.

[11] PERSISTANT STUDIOS. PopcornFX [Computer software]. `http://www.popcornfx.com/`, 2017. accessed 13-03-2017.

[12] RITSCHEL, T., IHRKE, M., FRISVAD, J. R., COPPENS, J., MYSZKOWSKI, K., AND SEIDEL, H.-P. Temporal glare: Real-time dynamic simulation of the scattering in the human eye. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 183–192.

[13] SCHRADE, E., HANIKA, J., AND DACHSBACHER, C. Sparse high-degree polynomials for wide-angle lenses. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 89–97.

[14] SMITH, W. *Modern Lens Design*, 2 ed. McGraw Hill Professional, 2004.

[15] THE QT COMPANY. Qt 5 [Programming library]. `https://www.qt.io/`, 2017. accessed 13-03-2017.

[16] ZEMAX LLC. OpticStudio [Computer software]. `http://www.zemax.com/os/opticstudio`, 2017. accessed 13-03-2017.

[17] ZHENG, Q., AND ZHENG, C. Neurolens: Data-driven camera lens simulation using neural networks. In *Computer Graphics Forum* (2017), Wiley Online Library. DOI: `10.1111/cgf.13087`.