Computers & Graphics 115 (2023) 472–483

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on CAD/Graphics 2023

# Efficient tile-based rendering of lens flare ghosts

Andrea Bodonyi [a,b,*], Roland Kunkli [a]

[a] Faculty of Informatics, University of Debrecen, Debrecen, Hungary
[b] Doctoral School of Informatics, University of Debrecen, Debrecen, Hungary

ARTICLE INFO

*Article history:*
Received 12 May 2023
Received in revised form 4 July 2023
Accepted 9 July 2023
Available online 13 July 2023

*Keywords:*
Lens flare
Physically based rendering
Tiled rendering
Camera
Rasterization
Barycentric coordinates

ABSTRACT

The lens flare phenomenon is often an undesired artifact of the imaging process; however, it has become an important artistic tool in photography and cinematography as well as a highly impactful component for increasing the level of realism for computer-generated images. In this paper, we present a novel method for efficiently simulating the lens flares of optical camera systems in highly interactive environments. Recreating this effect in a physically correct way necessitates the use of ray tracing, which we made much more computationally efficient by using a tiled approach to rasterize the ghosts of the lens flare. One of the main drawbacks of the current state-of-the-art method is the huge pixel overdraw resulting from the large number of ghosts being rasterized individually onto the output image. The problem is made even worse when dense ray grids are utilized for improving the quality of the simulation. We overcome these limitations by collecting all the ray-traced ghost data into screen-aligned tiles and accumulating the per-pixel contributions in a single pass. We demonstrate that our tiled approach significantly outperforms the previous algorithm, scales much better with the number of flares rendered, and facilitates the efficient simulation of lens flares in real-time applications, while maintaining the physical correctness.



## 1. Introduction

Lens flares are phenomena created by the imaging system when light rays traverse it in undesired ways. When entering the optical system, normally, the ray should only undergo refractions by the lens surfaces before arriving at the sensor. However, there are cases when reflection also occurs during the traversal of the rays, which produces the lens flare effect [1]. From these undesired scenarios, the rays that suffer an even-number of reflections have the potential of reaching the sensor and creating visible flares, provided that they retain enough energy (mainly those suffering two interreflections).

These effects are often considered as artifacts in the field of photography, which led to the camera designers taking a variety of countermeasures to prevent the creation of lens flares, such as the use of optimized camera housing and the application of antireflective coatings to the lenses. However, video games and movies often use lens flares as an artistic tool for increasing the plausibility of the rendering [2,3]. Lens flares can also be used for simulating light conditions that would fall outside of the dynamic range of display devices [4] since the human visual system understands the lens flares as a consequence of intense luminance [5,6].

Flares are mostly visible in scenes with high intensity light sources and can be divided into two main components: starbursts and ghosts. Starbursts are generally high intensity, star-like patterns that appear at the position of the light source, decreasing the overall contrast of the image, while ghosts represent multi-colored, mainly translucent spots positioned farther away from the light source.

Throughout the history of computer graphics, the simulation of this phenomenon has taken two different paths. On one hand, there exist methods that favor computational performance over physical accuracy and reproduce lens flares using static texture sprites, which are manually placed based on the position of the light source. On the other hand, several methods exist in the scientific literature that utilize ray tracing for a physically based approach of rendering lens flares, which reproduce the phenomenon in a far more accurate manner, but at the cost of a highly increased length of the simulation.

With the rendering of starbursts being mostly inexpensive, the main cost of the simulation is the rendering of ghosts. The problem is further worsened as the number of light sources and complexity of the optical system grow, because the number of ghosts to be rendered increases proportionally, leading to enormous bandwidth requirements, and thus, significantly hindering the performance of ray-traced lens flare simulations. Our aim was

* Corresponding author at: Faculty of Informatics, University of Debrecen, Debrecen, Hungary.
*E-mail addresses:* bodonyi.andrea@inf.unideb.hu (A. Bodonyi), kunkli.roland@inf.unideb.hu (R. Kunkli).

https://doi.org/10.1016/j.cag.2023.07.019

**Fig. 1.** Examples of lens flares generated using our proposed tiled rendering method for an Itoh (left) and a Nikon (right) camera system. Rendering took, respectively, 4.04 ms and 12.79 ms on an NVIDIA RTX 3060 GPU and at a display resolution of 1920 × 1080.

to overcome these limitations for the physically based rendering of ghosts.

In this paper, we propose a tile-based method for efficiently rendering large number of ghosts using ray tracing, which can produce physically based outputs at real-time frame rates. Our method significantly reduces the length of the simulation when utilizing complex optical systems, simulating a large number of ghosts, or tracing dense ray grids. Our proposed solution doubles the rasterization performance even for the simplest setup and achieves a five-times speedup in the most demanding scenarios. Fig. 1 demonstrates two representative lens flare renderings generated using our proposed method for two different camera systems. Our main contributions are:

- A ghost rasterization method using screen-aligned tiles and barycentric coordinates, which facilitates the efficient rendering of large volumes of ghost information.
- A two-level tile-building strategy, which efficiently builds the tiles of the rasterization step and reduces the length of the process compared to the naive approach.
- A dynamic and adaptive primitive-merging strategy, which effectively filters and collapses the primitives of the ray grid, substantially reducing the amount of the data to be processed.

## 2. Previous work

Because of the growing desire for increasingly higher levels of realism in computer-generated images, a high emphasis of camera attributes can be observed in the related research areas [7–9]. Besides the effects such as motion blur and depth of field, the simulation of lens flares has also received significant attention, with related works materializing in two main and conceptually different approaches.

### 2.1. Approaches based on texture sprites

One way of simulating lens flares is using static textures composited with the input images. Although this approach can produce outputs in a computationally efficient manner, the lack of physical correctness significantly hinders the plausibility of the resulting simulations.

The method by Kilgard [10] used multiple flare and shine textures, which were simply placed on a line going through the center of the image, with the animation being realized manually as well. King [11] then altered the flares originating from the camera movements by adjusting the size and translucency of the elements depending on the distance between the image center and the image-space position of the light source. Maughan [12] realized a fine-tuning of the intensity of the flares by considering the partial visibility of extended light sources in the image. Later, Sekulic [13] proposed the use of occlusion queries for further advancements in the performance of the lens flare visibility checks. Oat [14] suggested the use of steerable postprocessing filters for simulating flares at the high-intensity points of the image. Finally, Alspach [15] implemented a vector-based approach to simulate lens flares in a user-configurable manner.

### 2.2. Approaches based on ray tracing

As the desire for higher real-world similarity grew, the demand for physically correct lens flare simulations also increased. Since this phenomenon is highly camera-specific, it was essential to include the optical properties of the camera in the process. Generally, ray tracing is a widely used component of this approach, and despite its increased computational complexity, it is often favored because of the significantly higher levels of realism it can achieve.

First, Chaumond [16] used basic light path-tracing for generating physically plausible results. Similarly, Keshmirian [17] applied photon-mapping for determining the flow of light in the optical system. One of the main drawbacks of these methods is the disregard of anti-reflection coatings, and thus, specific camera effects (such as spectral flares) cannot be simulated. While proposing a spectral camera lens, Steinert [9] included material coatings in the simulation. Although their method can generate multi-color flares by using the absorption characteristics of the lenses, its performance makes it highly unsuitable for real-time applications.

Hullin et al. [18] implemented a ray bundling method with coarse ray grids to significantly increase the lens flare simulation efficiency. The simulated ghosts were rendered in a triangulated form to facilitate the use of interpolation to fill in the gaps in the coarse ray grid. A large drawback of this approach is that the ghosts are rasterized one-by-one onto the output image leading to an unsuitably high number of memory transactions, and therefore significantly reducing the throughput of the algorithm. Furthermore, the preprocessing time can also substantially increase for optical systems of considerable complexity. Lee and Eisemann [19] improved the method by Hullin et al. through approximating the result of the ray-tracing process using transfer matrices and utilizing a sprite-based rendering method to gain its

benefits as well. The main limitation of their approach stands in the inability of handling complex lens flare shapes, such as non-linear deformations. Hennessy [20] determined the intensity of a ghost based on the effective aperture and computed the colors by tracing a single ray through the system with a random incidence angle for each channel of each ghost. Later, Hullin et al. [21] modeled the light transfer in the optical system using polynomial systems instead of brute-force ray tracing. Hanika et al. [22] achieved further advances in precision by using a Monte Carlo renderer.

To simulate physically plausible lens flares without building on the precise description of the optical system, Walch et al. [23] used real photo captures with visible flares, with the occurrence of the lens flare in the specific light setting predicted using Bézier curves.

The starburst component of the lens flare effect has a similar optical cause as the glare of a human eye. This phenomenon arises from the diffraction and light scattering thanks to the obstacles (such as the aperture and small floaters) inside the imaging system. In relevant works, the starbursts in the human eye are often simulated using the Fresnel's diffraction integral [4,24], which is efficiently computed using the fast Fourier transform with an image representing the 2D projection of the obstacles. Alternatively, Scandolo et al. [25] proposed a quad-based approach as a significantly faster, closed-form solution producing starburst textures.

Finally, Joo et al. [26] proposed a ray-tracing method for handling aspheric lenses and their mechanical imperfections stemming from the manufacturing process. Their method can produce much more physically accurate flare textures for use with the sprite-based algorithms, further increasing the plausibility of the simulation.

## 3. Background

### 3.1. Optical system description

An optical system used for analytical ray tracing is built as a series of lens elements and an aperture. To reduce the complexity of the ray-tracing process, the lens elements are represented as planes and spherical surfaces, using analytical formulas. Real-world optical systems may contain more complex elements (like aspheric surfaces), which we chose to ignore in our method, because of the higher costs associated to the intersection tests with such surfaces. However, this is not a limitation for our approach, as extending our method with such surfaces is trivial if the corresponding analytical ray-intersection formulas are available.

The elements of the optical system are modeled as single surfaces even when two or more distinct surfaces comprise a single lens element. The properties belonging to a single surface are the height, refractive index, Abbe number, distance from the next element (thickness), radius of curvature, and refractive index of the antireflection coating. The shape of the aperture is modeled as a mask texture, with the height used for the ray tracing predefined. Fig. 2 shows a schematic of a representative optical system, with Table 1 summarizing the properties of its elements.

### 3.2. Ghost enumeration

As the rays reach an interface in the optical system, part of its energy is absorbed, another part is reflected, and the remaining energy passes through toward the next element. The lens flare ghosts correspond to ray paths that contain at least one reflection; more specifically, rays suffering an even number of reflections have the potential to reach the sensor.

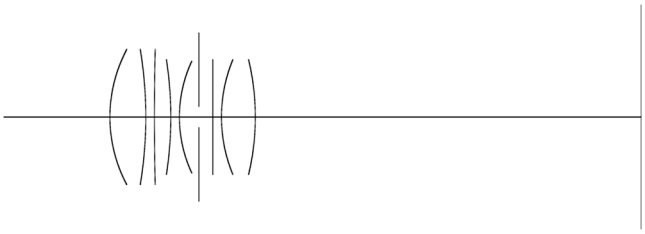

**Fig. 2.** Visualization of the Heliar Tronnier lens described in Table 1.

**Table 1**
Parameters of the surfaces for a Heliar Tronnier optical system (USP 2645156, f/2.8, 100 mm effective focal length, 60° field of view). The lens coating wavelength was set to 620 nm for all lenses.

| Height | Thickness | Radius of curvature | Refractive index | Abbe number |
|---|---|---|---|---|
| 14.5 | 7.7 | 30.81 | 1.65 | 58.57 |
| 14.5 | 1.85 | −89.35 | 1.60 | 38.03 |
| 14.5 | 3.52 | 580.38 | 1.00 | 89.30 |
| 12.3 | 1.82 | −80.63 | 1.64 | 47.74 |
| 12.0 | 4.18 | 28.34 | 1.00 | 89.30 |
| 11.6 | 3.00 | 0.0 | 1.0 | 89.3 |
| 12.3 | 1.85 | 0.0 | 1.58 | 40.98 |
| 12.3 | 7.27 | 32.19 | 1.69 | 53.20 |
| 12.3 | 82.86 | −52.99 | 1.0 | 89.30 |

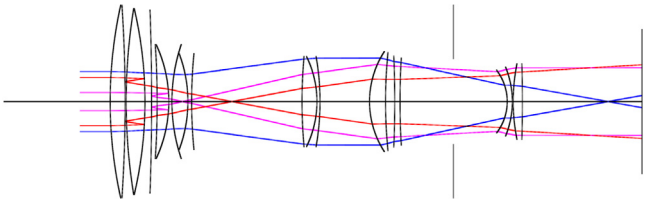

**Fig. 3.** Enumerated light paths with a Nikon zoom lens (S.53-131852, f/16, 140 mm focal length). The blue lines represent the light paths traversing the system without any reflections, creating the desired image. The red and purple lines represent paths suffering two interreflections, therefore resulting in ghost effects.

Following the concepts of Hullin et al. [18], we enumerate the potential light paths that may cause ghost effects. Fig. 3 demonstrates a few examples of such enumerated light paths (generated using the open-source OpenLensFlare tool [27]). When suffering reflection, the energy of the ray decreases. While, theoretically, considering all such light paths would be necessary for complete correctness, in most cases, only the rays suffering two reflections hold enough energy to create visible ghost effects. Therefore, we confine the simulation to these light paths. Furthermore, following the suggestions of [18], we also ignore ray paths that cross the aperture more than once (when the two reflections occur on different sides of the aperture), because a large portion of such rays is often blocked by the aperture, thereby making the contributions of such ghosts small and their simulations wasteful. The light paths shown in Fig. 3 were generated with these considerations.

### 3.3. Ray tracing ghosts

To obtain the projection of the ghosts on the sensor, we use ray tracing for each of the enumerated light paths. Following the

suggestions of Hullin et al. [18], a sparse grid of parallel rays is propagated through the optical system. The surfaces of the optical system along the ghost paths are taken in a consecutive order, and reflections and refractions are calculated at each surface until the ray finally reaches the sensor.

Because we model the lens surfaces as planes and spheres, we get a series of analytical surface descriptions, facilitating the simple calculation of the behavior of the light rays by efficiently performing the ray-sphere and ray-plane intersection tests. The result of this procedure is a ray grid that is projected onto the sensor of the optical system, representing a single ghost. The gaps between the grid elements are filled using rasterization and interpolation to obtain a continuous ghost image.

During the ray-tracing process, we also keep track of several per-ray properties that will be necessary for the interpolation process. Firstly, the amount of transmitted and reflected energy is calculated to obtain the final intensity of each ray when they hit the sensor, using the Fresnel equations, which can be formulated as follows [28]:

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

$$T = 1 - R, \tag{2}$$

where $R$ and $T$ denote, respectively, the amount of reflected and transmitted energy. Secondly, the heights of the optical elements are used to identify invalid rays (such as the rays that hit the lens housing or get blocked by the aperture), because these invalid elements facilitate the analytically continuous treatment of the interpolation process and thus play a significant role in enabling the sparse ray-tracing approach. Finally, because each ray is guaranteed to only hit the aperture once, we also mark the relative locations in the aperture plane of each ray and use them as texture coordinates, which will be utilized during rasterization with a mask texture for handling the exact iris shape of the optical system.

### 3.4. Barycentric coordinates

Barycentric coordinates have been widely used to perform interpolation tasks on a variety of 2D primitives [29,30]. Traditionally, the calculation of barycentric coordinates in two dimensions is performed using triangles, because only three reference points are required for obtaining the corresponding barycentric coordinates (page 46 in [31]). However, there may be cases when the barycentric coordinates need to be obtained for more than three vertices, such as in the case of a quad.

The computation of the barycentric coordinates using the vertices of a triangle can be realized based on the method suggested in [31]:

$$\mathbf{A} = \begin{pmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{pmatrix}, \quad \boldsymbol{\lambda} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} x_P - x_1 \\ y_P - y_1 \end{pmatrix}, \tag{3}$$

$$\boldsymbol{\lambda} = \mathbf{A}^{-1} \cdot \mathbf{b}, \tag{4}$$

with the notations shown in Fig. 4

For a more GPU-efficient calculation, Skala [32] proposed the use of homogeneous coordinates combined with cross product for obtaining the barycentric coordinates, formalized as follows:

$$\mathbf{b} = \mathbf{x} \times \mathbf{y} \times \mathbf{w}, \tag{5}$$

where $\mathbf{x} = (x_1, x_2, x_3, x_P)^T$, $\mathbf{y} = (y_1,\ y_2,\ y_3,\ y_P)^T$, $\mathbf{w} = (1,\ 1,\ 1,\ 1)^T$, and $\mathbf{b} = (b_1, b_2, b_3, b_4)^T$ is the homogeneous form of the barycentric coordinates. The final barycentric coordinates are then obtained as follows:

$$a_1 = -\frac{b_1}{b_4}, \quad a_2 = -\frac{b_2}{b_4}, \quad a_3 = -\frac{b_3}{b_4}. \tag{6}$$

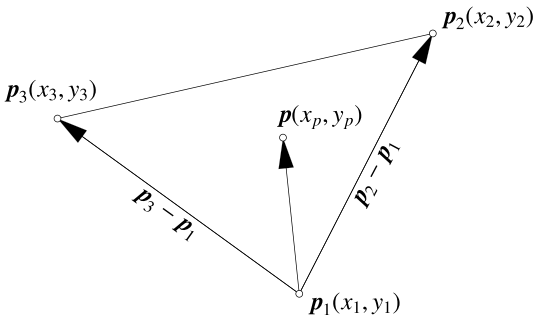


**Fig. 4.** Notations used to compute the barycentric coordinates of point **p** based on a triangle.

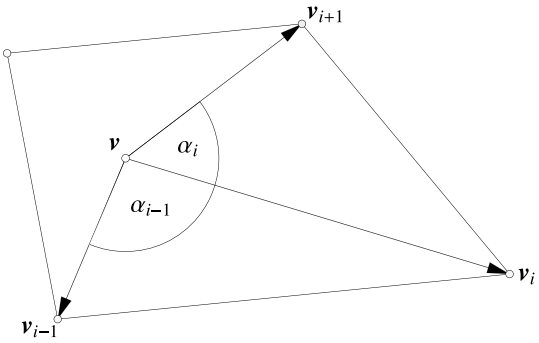


**Fig. 5.** Notations used to compute the barycentric coordinates of point **v** based on a quadrilateral.

To compute the barycentric coordinates for all four vertices of a quadrilateral, Hormann and Tarini [33] used mean value coordinates in their work for correctly handling the computation for all types of quads, including concave and self-intersecting ones. Their approach computes the barycentric coordinates in the following form:

$$\lambda_i(v) = \frac{\tan(\alpha_{i-1}(v)/2) + \tan(\alpha_i(v)/2)}{\|v - v_i\|}, \tag{7}$$

using the notations of Fig. 5 and $\alpha_i(v)$ denoting the signed angle between vectors $v_i$ and $v_{i+1}$.

However, in the case of convex and not self-intersecting polygons, a more efficient computation of barycentric coordinates can be realized using Wachspress coordinates [34]:

$$\lambda_i(v) = \frac{w_i(v)}{\sum_{j=1}^{n} w_j(v)}, \tag{8}$$

$$w_i(v) = A(v_{i-1},\ v_i,\ v_{j+1}) \prod_{j \neq i-1,\ i} A(v,\ v_j,\ v_{j+1}), \tag{9}$$

where $A(v,\ v_j,\ v_{j+1})$ denotes the signed area of the triangle $[v,\ v_j,\ v_{j+1}]$. Alternatively, Loop and DeRose [35] proposed the following alternative definition for $w_i$ in Eq. (9):

$$w_i(v) = \prod_{j \neq i-1,\ i} A(v,\ v_j,\ v_{j+1}). \tag{10}$$

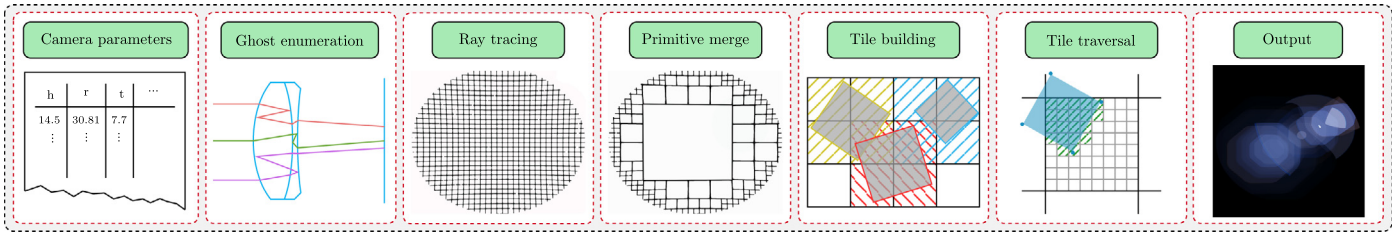


**Fig. 6.** Overview of the main steps of our proposed lens flare ghost rendering method. First, our algorithm takes the physical parameters of the simulated camera system as input. In the following step, we enumerate the most prominent, highest intensity ghost paths. We then trace a sparse ray grid through the optical system for each combination of light wavelength and ghost path. Following this, we build quadrilaterals out of the output ray grid of the lens flares, merge the similar neighboring quads, and store the results in one global buffer. In the next phase, we build a two-level (coarse and dense) screen-aligned tile-hierarchy from the indices of quads that intersect the individual tiles. Finally, we produce the output by traversing the corresponding dense tile buffers for each output image pixel and collecting the contributions of the relevant quads.

## 4. Our method

**Algorithm 1:** Our proposed algorithm for rendering lens flare ghosts.

**Input :** Optical system parameters $\mathcal{O}$,
list of light wavelengths $\Lambda$,
list of light sources $\mathcal{L}$,
camera state of the current frame $\mathcal{C}$,
number of primitive-merging steps $n_m$
**Output:** List of output image pixels $\mathcal{P}_o$ with all ghosts corresponding to the inputs simulated

$\mathcal{G} \leftarrow$ *EnumerateGhosts* $(\mathcal{O})$

**foreach** $(g, \lambda, l) \in \mathcal{G} \times \Lambda \times \mathcal{L}$ **do**
  $\sigma \leftarrow$ *DetermineIncomingLightDirection* $(\mathcal{C}, l)$
  $n_g \leftarrow$ *DetermineRayGridSize* $(g, \lambda, \sigma)$
  $\mathcal{R}_i \leftarrow$ *ConstructRayGrid* $\left(n_g,\ \sigma\right)$
  $\mathcal{R}_s \leftarrow$ *TraceRayGrid* $(\mathcal{R}_i, \mathcal{O},\ \lambda)$
  $\mathcal{P} \leftarrow$ *ConstructPrimitives* $(\mathcal{R}_s)$

**for** $i \in 1, \ldots, n_m$ **do**
  $\mathcal{P} \leftarrow$ *MergeNeighboringPrimitives* $(\mathcal{P},\ i)$

$\mathcal{T} \leftarrow$ *BuildTiles* $(\mathcal{P})$

**foreach** $(t, p_i) \in \mathcal{T} \times \mathcal{P}_o$ **do**
  $p_o \leftarrow$ *AccumulatePrimitiveContributions* $(t, p_o)$

As described in Section 2, algorithms built on static ghost textures lack any physical basis, while the methods built with physical correctness in mind often come with computation costs that make it impossible to use them in real-time applications. Our main goal was to develop a method that facilitates the rendering of the lens flare ghosts in such low-latency, interactive environments, while retaining the ability to simulate complex ghost shapes by using analytical ray tracing.

To this end, we utilize coarse ray tracing similar to the algorithm by Hullin et al. [18]. However, a critical drawback of their approach is the enormous memory bandwidth limitation arising from rasterizing the individual ghosts one-by-one. Ghosts are often significantly overlapping, and while barely visible, they still contribute to the final rendering. Therefore, rendering such ghosts is necessary to retain the physical accuracy, which causes heavy pixel overdraws and leads to high bandwidth requirements. The problem of overdraws was shown to cause issues in many other computer graphics problems. Tile-based approaches have been widely used to avoid this problem and decrease the cost of the rendering phase [36,37]. Therefore, we propose a tiled rendering approach, which significantly reduces the cost of ghost rasterization, and thus, substantially increases the overall throughput of the simulation.

An overview of the main steps of our algorithm is shown in Fig. 6 and an algorithmic description is provided in Algorithm 1. First, we perform a ray-tracing part with coarse ray grids, following the work of Hullin et al. [18]. We then build a set of quads out of the rays of each grid. Next, the primitives are collected into a coarse set of screen-aligned tiles, which is then refined into smaller, dense tile grid. Lastly, a tile traversal is performed for each output pixel, producing the final ghost simulation. In the remainder of this section, we provide a detailed description of the relevant new parts of our proposed algorithm.

### 4.1. Primitive construction

The first major step of rendering is the construction of the projected ghost primitives to be rasterized onto the output image. With the list of ghosts enumerated and readily available, we first perform ray tracing to obtain a set of projected ray grids that will be filled by interpolating the per-ray information. To this end, we trace a coarse grid of parallel rays through the optical system, which is performed for each channel of each enumerated ghost.

Because we would like to group parts of the ghosts modularly, we form primitives out of the output rays. To this end, we construct a set of quads from each neighboring $2 \times 2$ rays of the coarse grids. At this stage, we also discard irrelevant primitives. For a quad to be relevant, it must contain at least one valid ray, because such quads will contribute to the interpolation process and are necessary to retain analytical continuity. Therefore, we discard all quads comprising only invalid rays. We define a ray to be valid if it is in the inside or on the border of a ghost; it is invalid otherwise. The process is also visualized in Fig. 7.

Our filtering approach ensures that the list of primitives to be stored and processed is drastically decreased. All the primitives that survived the filtering process are then stored in a single primitive buffer, which will be used during all consecutive steps of the simulation.

### 4.2. Primitive merge

To render complex ghost shapes accurately and without artifacts, we often need to use a ray grid with a large number of elements. However, in the case of complex-shaped ghosts, the special curving that gives the complexity of the shape only happens close to the edges of the ray grid. Therefore, the need for higher ray density can be confined to these areas.

An ideal solution would use adaptive ray grids, whereby only the areas that correspond to high ghost-shape complexities would contain a higher density of rays. However, obtaining and managing such adaptive grids is memory intensive, making the task problematic and unsuitable for practical applications. Therefore, in our proposed method, we used uniform ray grids as suggested by Hullin et al. [18] and included a primitive-merging step,

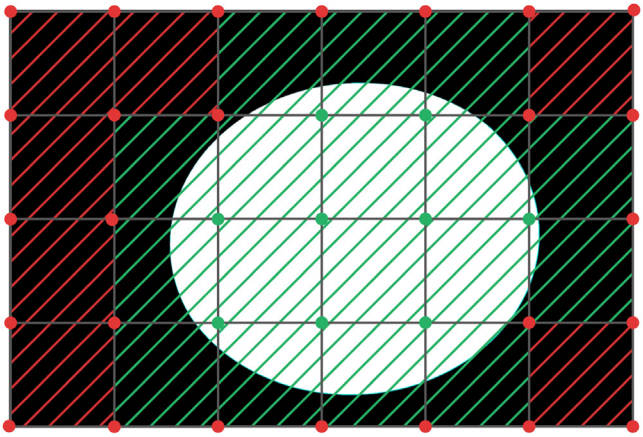

**Fig. 7.** Filtering of primitives in the ray grid. The green spots mark the valid rays, with the red spots representing the invalid rays. A primitive is stored (highlighted in green) if it contains at least one valid ray; it is discarded otherwise (red).

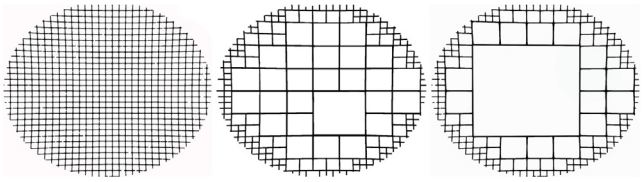

**Fig. 8.** Visualization of different levels of primitive merging. No primitive merging (left), moderate merging (middle), large merging (right).

where neighboring primitives corresponding to low-complexity areas are merged.

During this process, we attempt to merge $2 \times 2$ blocks of primitives that satisfy a set of conditions. We first check the validity of all primitives, ensuring that invalid primitives on the grid borders are ignored during this process. The orientations of the primitive edges must also be sufficiently close to each other; otherwise, a complex curving occurred in the region, and the primitives should not be merged. To evaluate this condition, we take the coordinate-wise summation of the corresponding edge pairs from the primitives to-be-merged. The edges are then said to be facing in the same direction if the result stays below a user-configurable threshold ($\gamma$), which can be formulated as:

$$\mathbf{e}_A = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \quad \mathbf{e}_B = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}, \tag{11}$$

$$(x_1 - x_2) + (y_1 - y_2) < \gamma, \tag{12}$$

where $\mathbf{e}_A$ and $\mathbf{e}_B$ is the edge-pair being tested. A $2 \times 2$ block of primitives is merged if the condition described above is satisfied by all four edge-pairs of all neighboring primitives.

Finally, we perform the primitive-merging step iteratively, during the primitive construction process; the outputs of a previous merge step can be further merged in consecutive steps, forming even larger blocks of primitives. If we consider the merging steps as a hierarchy, then the primitives must also be on the same level for merging, because merging cannot be performed between a smaller and larger primitive. Fig. 8 displays examples of our merging process using the ray-traced grid of a ghost generated by a Nikon lens system with no merging, moderate merging ($\gamma = 0.001$), and a larger amount of merging ($\gamma = 0.1$) applied.

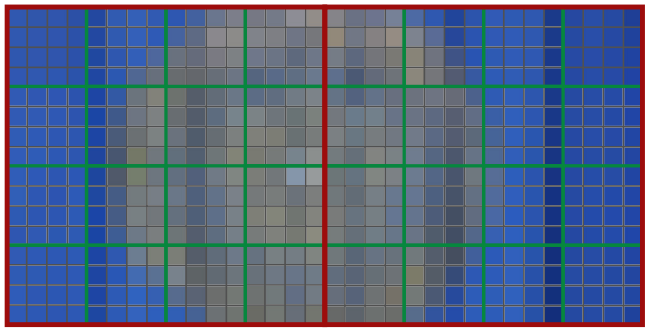

**Fig. 9.** Tile hierarchy. The cells outlined with red, green, and gray color mark the coarse tile grid, the dense tile grid, and the output image pixels, respectively.

### 4.3. Tile buffer building

The output of the ray-tracing algorithm is a set of ghost images built up by primitives. Our goal was to group the primitives based on their overlapping parts. To this end, the output image is divided into a uniformly sized grid, and the pixels of the output image are assigned to these tiles. Each cell then holds the sets of ghost primitives that are relevant to the pixels inside the tile. We create a list for each tile, containing all the primitives that contribute to at least one pixel of the output pixels of the corresponding tile.

To facilitate the efficient creation of the per-tile primitive buffers, we implemented a two-level tile hierarchy that improves the tile-building performance. On the first level, we group the primitives coarsely, which is realized during the primitive-building phase, after the primitives have been successfully merged. These coarse tiles represent a prefiltering step, which significantly reduces the number of tiles to be tested during the construction of the final, dense tile grid.

In the second step, we traverse the coarse tile list and assign each primitive to the relevant dense tiles covered by the given coarse tile. We ensure that each dense tile is fully contained by a coarse tile by choosing a coarse tile size as a multiple of the dense tile sizes. An example of such a tile hierarchy is shown in Fig. 9.

For both the coarse and dense tile hierarchy, the primitive grouping is based on tile-primitive intersections. For each primitive, we traverse the list of tiles individually and check for intersections. To make the intersection tests simpler and more efficient to perform, we use the screen-space coordinates of the tile edges and the axis-aligned bounding rectangles of the primitives. If an intersection is detected between a primitive and a tile, then the primitive is assigned to that tile. A schematic of this process is shown in Fig. 10 and an algorithmic description is provided in Algorithm 2.

**Algorithm 2:** Our two-level tile-building method.

**Input :** List of merged primitives $\mathcal{P}$,
list of coarse tiles $\mathcal{T}_c$ and dense tiles $\mathcal{T}_d$
**Output:** Per-tile buffers populated with all the overlapping primitives

```
foreach (t, p) ∈ T_c × P do
    if PrimitiveOverlapsTile(p, t) then
        AppendPrimitiveIndexToTile(p, t)

foreach t ∈ T_d do
    foreach p ∈ PrimitivesOfOwningCoarseTile(t) do
        if PrimitiveOverlapsTile(p, t) then
            AppendPrimitiveIndexToTile(p, t)
```

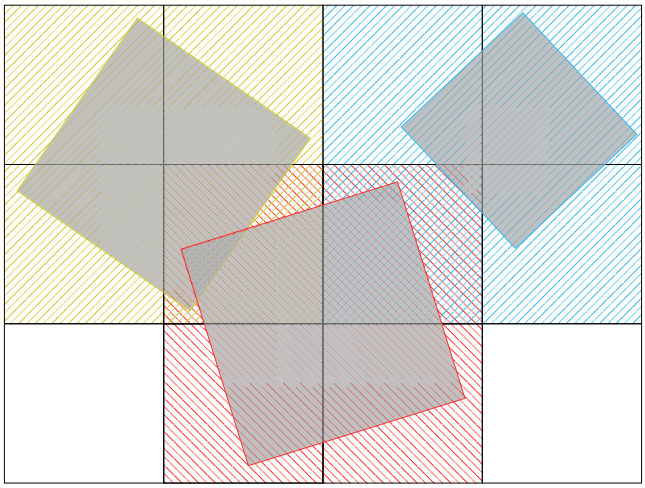

**Fig. 10.** Tile buffer building. Each cell in the grid represents one tile. The tiles marked by yellow, blue, and red indicate the ones containing at least one pixel of the primitive with the same outline color.

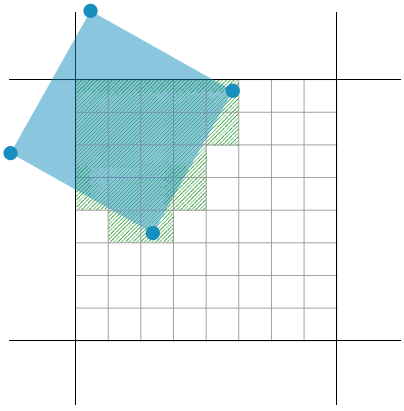

**Fig. 11.** Pixel-primitive intersection check. The cells outlined with gray mark the pixels of a given tile. The green cells are the ones actually intersecting the blue primitive. In those cases, there is a contribution to the final pixel color from the blue primitive.

The result of this process is a set of primitives potentially contributing to the pixels covered by a given tile. To reduce the memory consumption of the per-tile buffers, each set is realized using indices into the global primitive buffer that holds all the merged primitives resulting from the previous primitive construction and merge phases.

### 4.4. Per-pixel tile buffer traversal

The main goal of our proposed rendering approach is to avoid rasterizing each individual ghost pixel-by-pixel, because it is the main source of the huge number of overdraws. Rather, we aim to collect the contributions of all ghosts to any given pixel in a single step. We achieved this goal by accumulating the individual contributions using the per-tile buffers on a per-pixel basis, using the overlapping parts of corresponding primitives, as shown in Fig. 11.

The final rendering happens with a pixel-by-pixel traversal of the output image, whereby the contributions of the relevant primitives are determined and accumulated. To this end, we take all of the output pixels one-by-one and traverse the primitive list belonging to the tile that contains the given pixel. The output pixel is then obtained by taking the contributions of all the belonging primitives (which can be zero in case of no intersection) during the traversal and accumulating them. An algorithmic description of this process is also provided in Algorithm 3. The exact process of determining the per-pixel contributions is realized with barycentric coordinates, as described in Section 4.5.

**Algorithm 3:** Our approach for calculating and accumulating the per-pixel primitive contributions.

**Input :** List of dense-tile buffers $\mathcal{T}_d$
**Output:** Output image with all primitives rasterized

**foreach** $t \in \mathcal{T}_d$ **do**
  $\mathcal{P}_t \leftarrow ReadPrimitivesInTile\,(t)$

$\mathcal{P}_i \leftarrow PixelsInTile\,(t)$
**foreach** $(p_t, p_i) \in \mathcal{P}_t \times \mathcal{P}_i$ **do**
  $b \leftarrow ComputeBarycentricCoordinates\,(p_i,\ p_t)$
  **if** $b \in [0,\ 1]$ **then**
    $p_i \leftarrow AccumulatePrimitiveContribution\,(p_i,\ p_t,\ b)$

The pixels are traversed in parallel and grouped based on the dense-tile correspondences. We utilized hardware-based grouping (realized using compute shaders), because such an approach facilitates the division of primitive data reads between the group elements and the sharing of the read data using shared memory. Consequently, the memory bandwidth requirements of our approach are significantly reduced, leading to highly increased tile traversal performances.

### 4.5. Computation of per-pixel primitive contributions

To evaluate the pixel-primitive intersection and perform data interpolation, we utilized barycentric coordinates. Using the four vertices of a primitive as reference points, we can determine the corresponding barycentric coordinates of any given pixel. Due to the nature of barycentric coordinates, the position of a pixel is determined relative to the primitive, which can be used to check for the existence of a potential intersection.

As described earlier, the output of the ray-tracing process is created by dividing the projected ghost surfaces into quadrilaterals after the ray grid is propagated through the lens system. A potential solution for processing the primitives using barycentric coordinates may be to split up the quads into two triangles and process both triangles separately using Eq. (4) or Eq. (6). However, based on our practical experiments, working with quadrilaterals directly is beneficial to the computational performance of the procedure. In this case, we compute the barycentric coordinates of the pixel relative to the four vertices of the quad using Eq. (7) or Eq. (8).

With both approaches, if at least one of the barycentric coordinates is less than zero or greater than one, then the pixel is not part of the quad, and thus, its contribution is zero. Otherwise, we evaluate its contribution to the final pixel color, which is realized using its barycentric coordinates as well. Taking the vertices of the primitive as reference points, we obtained the final contribution by calculating a weighted sum of the properties (such as intensity and aperture intersection coordinates) of the primitive (triangle or quad) vertices (using the corresponding barycentric coordinates as weights), which is accumulated during the tile traversal.

Finally, with the input to ray tracing being a regular grid and the optical system elements being spherical surfaces, the grid projected onto the sensor contains only convex quadrilaterals. Therefore, we choose to apply the approach of Loop and DeRose [35] with Eq. (8) and Eq. (10). Otherwise, the generalized approach with mean value coordinates works well for computing barycentric coordinates for a quadrilateral. The impact of different calculation methods on the computation times is evaluated in Section 5.7.

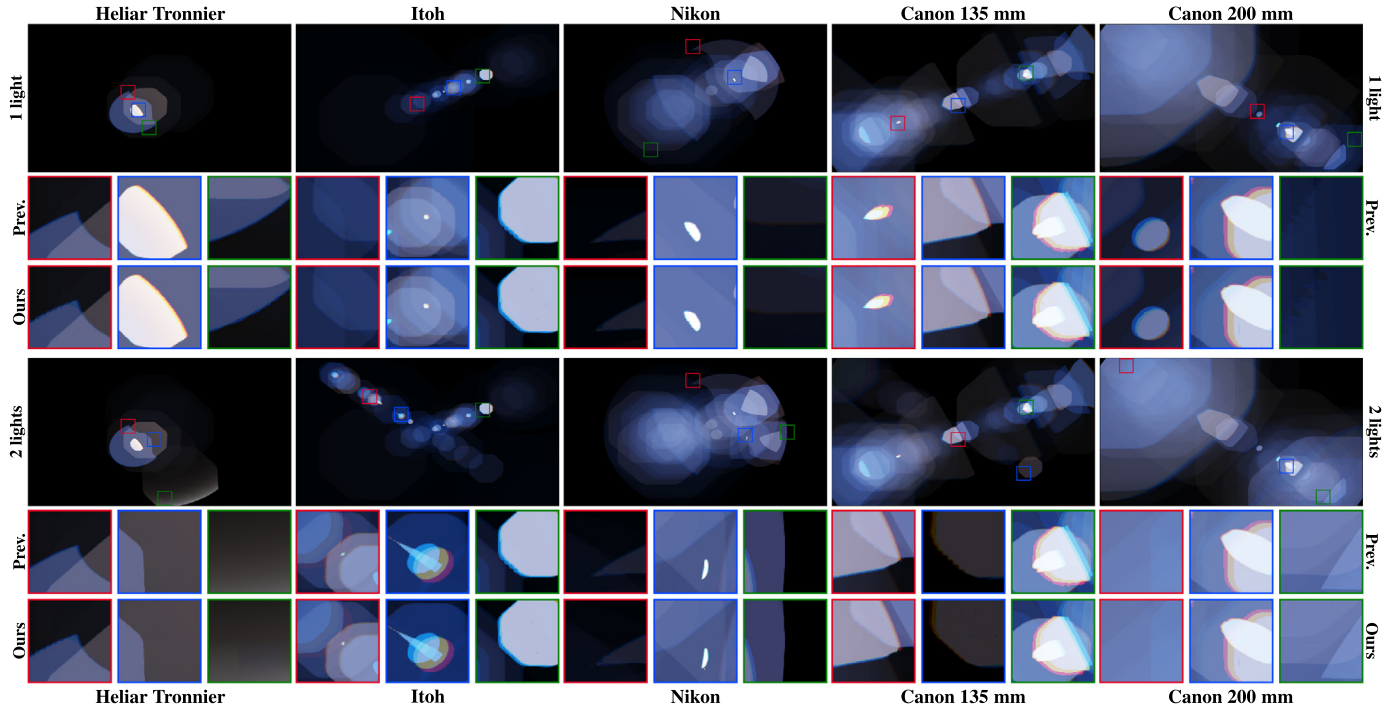


**Fig. 12.** Examples of lens flares generated using our proposed tiled rendering method with a single light source and two light sources. The insets below the output images provide a comparison between our proposed algorithm and the reference method by Hullin et al. [18].

## 5. Results

### 5.1. Test setup

We used the C++ programming language and OpenGL graphics API to create a reference implementation of our proposed method. To implement the computationally heavy phases of the algorithm, we utilized GLSL compute shaders. As outlined in Section 4.4, this implementation facilitated the use of group shared memory, significantly increasing the efficiency of data reads. In terms of hardware, we used an AMD Ryzen 5 1600 CPU and an NVIDIA GeForce RTX 3060 GPU for all our performance measurements.

To evaluate our results, we used five optical systems with varying complexities, all of which were obtained from the example lens systems of the OpenLensFlare framework:

- A Heliar Tronnier lens with a relatively small number of lens interfaces (USP 2645156, f/16, 100 mm focal length, 8 surfaces, 11 ghosts).
- An Itoh wide angle zoom lens (USP 4196968, f/32, 123 mm focal length, 18 surfaces, 64 ghosts) representing medium-complexity case.
- A Nikon tele zoom lens (S.53-131852, f/16, 140 mm focal length, 21 surfaces, 139 ghosts) to test a different medium-complexity lens system.
- A Canon tele zoom lens in two different zoom states (USP 5537259, f/32, 135 mm and 200 mm focal lengths, 33 surfaces, 304 ghosts) to model highly complex optical systems as well.

For all optical systems, we also tested two setups, using one and two light sources in total. The output images were created with a display resolution of 1920 × 1080 and are shown in Fig. 12.

To compare our results against, we also implemented the method by Hullin et al. [18] using the same tools. In the ray tracing phase, we propagated one distinct ray grid through the optical system for each unique combination of light source, ghost, and RGB channel.

During the comparison of the previous algorithm and our method, we used a base grid size of 256 × 256 with both approaches, which was dynamically reduced based on the projected size of the ghost on the sensor. To this end, we utilized the following formula to determine the amount of reduction:

$$\min\left(\max\left(\left(\frac{G_x \cdot G_y}{S_x \cdot S_y} \cdot \sigma\right)^{\varphi},\ 0.05\right),\ 1\right), \tag{13}$$

where $S$ and $G$ denote the sizes of the rectangles corresponding to the camera sensor and the ghost projected to the sensor, and $\sigma$ and $\varphi$ are user-configurable parameters. We used $\sigma = 1$ and $\varphi = 1/2$ in all our experiments, with the parameter values determined empirically, because such a setting provided a sufficient reduction of the ray grid size for small ghosts.

Our approach differs from the precomputation-based method outlined in [18], which determines the size of the ray grids on a per-ghost basis. Although our chosen base grid size is considerably large, our aim with this choice was to maximize the output quality and minimize the complexity of determining the grid sizes. Following the precomputation approach outlined in [18] would be a viable alternative for both of the compared methods; therefore, our different approach does not impact the results of our performance comparisons.

### 5.2. Overall performance

In this section, we evaluate the performance profile of our proposed rendering method and compare it against the algorithm presented in [18] for all of the test cases described in the previous subsection. For this performance analysis, we measured the running times of the separate phases of our proposed algorithm and the total rendering times for both approaches. The results are summarized in Table 2.

During the performance measurement, we utilized a tile size of 8 × 8 with a coarse tile size of 128 × 128, which were

**Table 2**
Computation times (in milliseconds) of the various substeps of our proposed method as well as the total rendering times of the previous and our new algorithm, using five different optical systems with one and two light sources. For the measurements, we used 8 × 8 as tile size and 128 × 128 as coarse tile size.

| | | Our method | | | | Previous method | | |
|---|---|---|---|---|---|---|---|---|
| | | Ray tr. &coarse tiles | Dense tiles | Tile trav. | Total | Ray tr. | Raster. | Total |
| 1 light | Heliar Tronnier | 0.26 | 0.10 | 0.36 | **0.72** | 0.14 | 0.96 | **1.1** |
| | Itoh | 1.17 | 0.13 | 0.57 | **2.01** | 0.78 | 1.27 | **2.05** |
| | Nikon | 3.47 | 0.28 | 1.65 | **5.4** | 2.36 | 5.46 | **7.82** |
| | Canon 135 mm | 8.09 | 0.31 | 1.83 | **10.23** | 6.42 | 13.09 | **19.51** |
| | Canon 200 mm | 17.27 | 0.49 | 4.54 | **22.30** | 13.49 | 19.44 | **32.93** |
| 2 lights | Heliar Tronnier | 0.44 | 0.11 | 0.54 | **1.09** | 0.23 | 1.37 | **1.6** |
| | Itoh | 1.96 | 0.17 | 0.87 | **3.00** | 1.30 | 2.12 | **3.42** |
| | Nikon | 5.87 | 0.50 | 2.87 | **9.24** | 4.29 | 13.36 | **17.65** |
| | Canon 135 mm | 9.38 | 0.36 | 2.73 | **12.47** | 6.77 | 14.3 | **21.07** |
| | Canon 200 mm | 24.95 | 0.66 | 5.49 | **31.10** | 18.70 | 27.07 | **45.77** |

chosen empirically, because in our experiments, these parameters well balanced the running times and the quality of the outputs. However, the impact of the different parameters is also evaluated in later parts of this section.

Considering the performance of the separate phases, the ray-tracing part of the algorithm forms the largest portion of the overall computation cost. The rasterization is the second in line, which, as demonstrated by the results, effectively manages the accumulation of the large number of primitives in the output, even in the case of massive data amounts. Finally, the cost of the tile building is negligible, which is mostly a consequence of our two-level tile hierarchy.

Our improvements presented here were mainly achieved by the fast collection of the primitives using the two-level tile-building procedure, the tremendously reduced memory bandwidth requirements resulting from the tile-based accumulation, and the reduced number of primitives resulting from the primitive-merging approach having an impact on the rasterization and the tile building steps as well.

In general, as can be seen from Table 2, our proposed method substantially outperforms the previous method; it approximately doubles the rasterization performance (i.e., excluding the cost of ray tracing) even for the simplest case and it is five times as fast in the most demanding scenarios. Our measurements also clearly exemplify that the rasterization part of our algorithm scales significantly better with the number of light sources than the previous approach. Consequently, our method is far more suitable for practical applications, where a large number of light sources are present in the scene. This performance increase primarily stems from the effective filtering and data minimization realized by the primitive-merging and tile-building approaches, and the bandwidth minimization achieved by significantly reducing the pixel overdraws via the tile-based rasterization.

### 5.3. Overall quality

Hullin et al. [18] performed a comparison of their ray-tracing-based solution and photos taken with real cameras and demonstrated that their method is capable of generating plausible simulations that properly mimic the captured lens flares. Since our proposed method builds on the exact same physical basis, we consider the method by Hullin et al. suitable for creating ground-truth images. Therefore, our main goal was to reproduce the outputs of the previous method but with a drastical decrease of the rasterization time.

To validate the simulation quality of our proposed algorithm, we compared our outputs with the ground-truth reference images for the simulations presented in Fig. 12. The results, summarized in Table 3, make it clearly visible that our method faithfully reproduces the target ground-truth simulations and leads to no significant deviations with respect to the reference method.

**Table 3**
PSNR (in decibels) values for the outputs of our proposed method presented in Fig. 12, with respect to the reference method.

| | Heliar Tronnier | Itoh | Nikon | Canon 135 mm | Canon 200 mm |
|---|---|---|---|---|---|
| 1 light | 50.27 | 49.50 | 49.91 | 46.33 | 46.23 |
| 2 lights | 46.86 | 46.81 | 48.27 | 45.91 | 47.59 |

We also highlighted the regions that produced the highest differences in Fig. 12. From these regions, it becomes visible that the most prominent disparities are created at the edges of under-sampled ghosts. This result arises from the interpolation process of the sparse ray grids, which would require correction for both methods. Increasing the size of the ray grid would fix this undesired effect, in which case, our interpolation method building on barycentric coordinates would not cause such errors.

Finally, the primitive-merging step also leads to some visible artifacts. However, for a precise comparison of the outputs of the rasterization approaches, we did not perform the spectral filtering suggested by Hullin et al. on the outputs presented in this paper. Our practical experiments show that such spectral blurring would completely eliminate these artifacts, and other simple image-space filters (such as a median filter) can be used to efficiently overcome this limitation as well. Furthermore, choosing a more conservative setting for the number of primitive-merging steps or a more precise merge heuristic can further reduce these artifacts. Therefore, our primitive-merging step would not cause any significant drawbacks in real-world applications.

### 5.4. Impact of coarse tiles

In this section, we evaluate the impact of our two-level tile-building strategy and the different tile sizes on the total rendering times. To this end, we tested 8 × 8 and 16 × 16 tile sizes and analyzed the overall running times in both test cases, with and without our coarse tile strategy, simulating the lens flares of a single light source.

Our results, summarized in Table 4, clearly demonstrate that using coarse tiles significantly reduces the rendering cost in each of the cases, because it provides an effective and fast prefiltering during the tile building; thereby, as mentioned in the previous section, it ensures that the cost of the tile building is minimal.

Furthermore, the evaluation also revealed that the 8 × 8 tile size performed better in each scenario. This behavior can be explained by the substantially smaller per-tile primitive lists, keeping the amount of the primitives to be processed during rasterization low. Based on our practical experiments, this trend holds for larger tiles as well; therefore, we believe that 8 × 8 tiles are ideal for practical applications, which is the reason for our choice in the measurements described in Section 5.2.

**Table 4**
Runtime measurements (in milliseconds) for different tile sizes, with and without using coarse tiles (128 × 128).

| | **8 × 8** | | **16 × 16** | |
|---|---|---|---|---|
| | No coarse | With coarse | No coarse | With coarse |
| Heliar Tronnier | 3.86 | 0.73 | 1.77 | 0.99 |
| Itoh | 6.84 | 1.89 | 3.38 | 2.26 |
| Nikon | 19.78 | 5.38 | 8.04 | 6.69 |
| Canon 135 mm | 25.04 | 10.76 | 13.56 | 12.24 |
| Canon 200 mm | 51.65 | 21.64 | 31.68 | 25.33 |

**Table 5**
Primitive counts and the required memory sizes (in megabytes) for different camera systems, light source amounts, and primitive-merging strategies.

| | | No merge | | Light merge | | Heavy merge | |
|---|---|---|---|---|---|---|---|
| | | Prim. | Mem. | Prim. | Mem. | Prim. | Mem. |
| 1 light | Heliar Tronnier | 122,129 | 16.77 | 19,577 | 2.69 | 13,808 | 1.90 |
| | Itoh | 210,309 | 28.88 | 31,347 | 4.30 | 31,011 | 4.26 |
| | Nikon | 665,942 | 91.45 | 99,767 | 13.70 | 80,021 | 10.99 |
| | Canon 135 mm | 562,492 | 77.25 | 143,413 | 19.69 | 109,120 | 14.99 |
| | Canon 200 mm | 1,953,436 | 268.26 | 244,855 | 33.63 | 204,673 | 28.11 |
| 2 lights | Heliar Tronnier | 210,809 | 28.95 | 28,877 | 3.97 | 23,453 | 3.22 |
| | Itoh | 315,335 | 43.30 | 50,657 | 6.96 | 50,543 | 6.94 |
| | Nikon | 1,280,183 | 175.81 | 188,204 | 25.85 | 153,584 | 21.09 |
| | Canon 135 mm | 643,200 | 88.33 | 148,146 | 20.34 | 131,781 | 18.10 |
| | Canon 200 mm | 2,721,915 | 373.80 | 343,908 | 47.23 | 293,430 | 40.30 |

### 5.5. Memory consumption

In this section, we evaluate the memory consumption of our proposed tiled rendering method. First, we analyzed the required storage size for the primitive data. To this end, we utilized a statically allocated buffer for five million primitives, which required 610.35 MB. Based on our practical experiments, this amount far exceeds the memory demand for the real-world simulation of the lens flare ghosts, therefore, it is a safe but slightly wasteful choice. However, it is also clearly visible that even with such a pessimistic choice for the storage amount, the memory requirement of the primitive buffer is completely suitable for practical applications. Furthermore, we also examined the actual number of the primitives for the five camera systems presented in Section 5.1 with one and two light sources and different merge strategies. The results of the experiment, which are summarized in Table 5, also prove the suitability of our approach for real-world applications and demonstrates the actual memory requirements for realistic scenarios.

The second main part of the memory consumption of our algorithm is the coarse and dense tile buffers. Since our method builds these tiles using indices into the primitive buffer, the memory requirement for the tile buffers is insignificant and completely suitable for real-world applications. Building a buffer capable of storing 50,000 quads for each coarse tile requires 257.492 MB GPU memory in total, which is a comparatively low memory footprint for practical applications. In case of 8 × 8 sized dense tiles, allocating memory for buffers storing 2000 primitives per tile also results in a total memory consumption of 249.192 MB. Based on our empirical experiences, these choices also far exceed the real requirements for both tile buffers, which we used as a safe but highly pessimistic choice for preventing artifacts resulting from inappropriate buffer sizes. We analyzed the real memory requirement of the dense tiles for the scenarios presented in Table 5, and summarized the results in Table 6. These results also clearly outline our observation that our selected buffer sizes are pessimistic and prove that our method is suitable for operating on commodity hardware with significantly less memory usage as well.

**Table 6**
Number of primitive indices stored in the dense tiles and the total required memory sizes (in megabytes) for different camera systems, light source amounts, and primitive-merging strategies.

| | | No merge | | Light merge | | Heavy merge | |
|---|---|---|---|---|---|---|---|
| | | Prim. | Mem. | Prim. | Mem. | Prim. | Mem. |
| 1 light | Heliar Tronnier | 332,133 | 1.27 | 142,643 | 0.54 | 123,146 | 0.47 |
| | Itoh | 540,705 | 2.06 | 218,598 | 0.83 | 217,006 | 0.83 |
| | Nikon | 2,102,238 | 8.02 | 940,987 | 3.59 | 846,657 | 3.23 |
| | Canon 135 mm | 2,084,646 | 7.95 | 1,157,224 | 4.41 | 1,031,562 | 3.94 |
| | Canon 200 mm | 6,139,602 | 23.42 | 2,737,477 | 10.44 | 2,520,730 | 9.62 |
| 2 lights | Heliar Tronnier | 578,862 | 2.21 | 233,763 | 0.89 | 211,570 | 0.81 |
| | Itoh | 763,951 | 2.91 | 325,343 | 1.24 | 324,900 | 1.24 |
| | Nikon | 3,986,146 | 15.21 | 1,741,900 | 6.64 | 1,575,888 | 6.01 |
| | Canon 135 mm | 2,428,032 | 9.26 | 1,309,292 | 4.99 | 1,234,047 | 4.71 |
| | Canon 200 mm | 8,345,476 | 31.84 | 3,715,012 | 14.17 | 3,462,741 | 13.21 |

**Table 7**
Performance (in milliseconds) and PSNR (in decibels) measurements of our proposed method without primitive merge, with a moderate primitive merge, and a heavy primitive merge.

| | No merge | | Light merge | | Heavy merge | |
|---|---|---|---|---|---|---|
| | Runtime | PSNR | Runtime | PSNR | Runtime | PSNR |
| Heliar Tronnier | 1.12 | 50.38 | 0.74 | 50.27 | 0.72 | 46.87 |
| Itoh | 2.45 | 49.54 | 2.03 | 49.50 | 1.91 | 49.50 |
| Nikon | 7.47 | 50.18 | 5.42 | 49.91 | 5.07 | 46.88 |
| Canon 135mm | 11.88 | 46.48 | 10.25 | 46.33 | 10.04 | 45.20 |
| Canon 200mm | 26.96 | 46.23 | 21.84 | 46.23 | 20.75 | 45.12 |

Finally, it is important to mention that we determined the maximum buffer sizes empirically, ensuring that our method is capable of rendering all of the ghosts to be displayed even with the large ray grid sizes and the most complex optical systems. The buffer sizes can be drastically reduced when choosing optical systems with lower complexities (that generate a significantly lower number of ghosts) or using smaller ray grid sizes. Nonetheless, this worst-case scenario perfectly demonstrates that choosing an 8 × 8 sized tile grid is perfectly suitable for running on consumer-grade hardware, despite the use of these wasteful settings. The results of our experiments summarized in Tables 5 and 6 are also in line with this observation.

### 5.6. Evaluation of primitive merge

To evaluate the impact of the proposed primitive-merging strategy on the running times and output quality, we evaluated three test setups: no primitive merge, light merge (4 merge steps, $\gamma = 0.001$), and heavy merge (4 merge steps, $\gamma = 0.1$). We also only used a single light source in all cases. For each setup, we measured the total rendering times and the corresponding peak signal-to-noise ratio (PSNR) with the outputs generated using the previous method [18] as references. The results are summarized in Table 7.

As demonstrated by our results, our primitive-merging approach provides computational gains even with the smallest optical system, as the computational complexity is low enough that primitives successfully filtered out contribute more to the overall length of the simulation than the overhead of the merging procedure. However, as the optical system complexity grows, it is clearly visible that merging the primitives can provide even higher computational savings, and consequently, substantially reduce the total rendering time.

Additionally, it can also be seen from Table 7 that the light primitive-merging setup has a negligible impact on the output quality but considerably improves the speed even when a medium-complexity optical system is used. Although the process is less impactful for the small optical system, this setup can be

**Table 8**
Performance (in milliseconds) of the tile traversal phase of our proposed rendering method using several different approaches for computing the barycentric coordinates.

| | Triangle [31] | Quad [33] | Triangle [32] | Quad [34] | Quad [35] |
|---|---|---|---|---|---|
| Heliar Tronnier | 0.49 | 0.44 | 0.41 | 0.39 | 0.36 |
| Itoh | 0.76 | 0.69 | 0.65 | 0.61 | 0.57 |
| Nikon | 2.54 | 2.21 | 1.99 | 1.86 | 1.62 |
| Canon 135mm | 2.86 | 2.49 | 2.24 | 2.08 | 1.82 |
| Canon 200mm | 6.40 | 5.62 | 4.95 | 4.63 | 4.01 |

useful as well when several light sources are utilized, generating a significant number of primitives.

Finally, although the heavy primitive merge may reduce the PSNR and cause visible artifacts, such errors can be hidden by the spectral-filtering approach suggested in [18], making this strategy a valid choice as well, and thus, resulting in an even larger performance increase. Finally, application- and camera-specific parameters can be used to find a middle ground, which appropriately balances the impact of the primitive-merging process on the running times and the output quality.

### 5.7. Barycentric coordinates

Because the application of barycentric coordinates forms a significant part of the computation cost of our proposed rendering method, we also evaluated the performance impact of the different approaches (outlined in Section 3.4) for computing the necessary barycentric coordinates. We measured the length of the tile traversal phase using triangles with the basic [31] and homogeneous-coordinates-based [32] computation method and using quads with the generalized approach [33], the algorithm proposed by Wachspress [34], and the modification suggested by Loop and DeRose [35]. For all test cases, we only used a single light source as input. Our measurements are summarized in Table 8.

It can be clearly seen from our measurements that the best results are produced by the approach suggested by Loop and DeRose and the similar method proposed by Wachspress. Furthermore, it can also be noted that the computation cost of the homogeneous-coordinates-based calculation method suggested by Skala also performs well despite the division of the quads to triangles. This is because the 4D cross product can be efficiently calculated on GPU with a low number of operations required.

Finally, the generalized quad-based and the basic triangle-based barycentric calculation requires a large number of operations, which leads to an increased computation cost of the tile traversal phase. However, our results clearly demonstrate that compared to the performance of the previous rendering (summarized in Table 2), our method significantly decreases the cost of rasterization using any of the tested approaches.

## 6. Conclusions

In this paper, we presented an efficient tile-based rendering method for rendering physically correct lens flare ghost effects. Our proposed approach is capable of rendering ghosts at a significantly lower computational cost than the previous methods, making it possible to be used in real-time applications even with complex optical systems as well as high numbers of ghosts and light sources.

To achieve our goals, we proposed a tile-based rasterization approach that facilitates the efficient rasterization of large number of ghosts that result from ray tracing with sparse grids. To this end, we manually rasterized the quads resulting from ray tracing with the sparse ray grids, for which we proposed several different, barycentric-coordinates based solutions. We also developed a two-level tile building approach, which substantially reduced the amount of data that needs to be processed during rendering. Finally, we also created a dynamic primitive-merging approach that makes the final rendering process more efficient by significantly reducing the quantity of the data to be processed.

When a large number of ghosts are to be simulated (such as when using multiple light sources and complex optical systems), our method is able to efficiently render the lens flare ghosts by minimizing the memory bandwidth requirement via the batched per-pixel rasterization approach and reducing data that needs to be processed. We also demonstrated using several different optical systems and test setups that our proposed approach substantially outperforms the previous state-of-the-art algorithm in all cases and scales significantly better with the number of ghosts rendered.

As for future work, we would like to decrease the required time for the ray-tracing phase – which could be realized by, e.g., using polynomial optics or neural networks – further reducing the rendering time. We would also like to lower the rasterization time, which we believe is possible using an incremental reformulation of the barycentric coordinates calculation. Another promising future work would be the rendering of the lens flares fully using neural networks [38], avoiding the cost of ray tracing altogether. Finally, finding alternative primitive-merging strategies would further increase the output precision and reduce the computational costs.

## CRediT authorship contribution statement

**Andrea Bodonyi:** Conceptualization, Methodology, Software, Investigation, Writing – original draft, Editing, Visualization. **Roland Kunkli:** Methodology, Writing – review, Visualization, Supervision.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## References

[1] Kingslake R. Optics in photography, vol. 6. Roca Baton, USA: SPIE Press; 1992, http://dx.doi.org/10.1117/3.43160.
[2] Pixar.The imperfect lens: Creating the look of Wall-E. 2008, Wall-E Three-DVD Box.
[3] Pekkarinen E, Balzer M. Physically Based Lens Flare Rendering in "The Lego Movie 2". In: Proceedings of the 2019 Digital Production Symposium. New York, USA: ACM; 2019, p. 1:1–3. http://dx.doi.org/10.1145/3329715.3338881.
[4] Ritschel T, Ihrke M, Frisvad JR, Coppens J, Myszkowski K, Seidel H-P. Temporal Glare: Real-Time Dynamic Simulation of the Scattering in the Human Eye. Comput Graph Forum 2009;28(2):183–92. http://dx.doi.org/10.1111/j.1467-8659.2009.01357.x.
[5] Spencer G, Shirley P, Zimmerman K, Greenberg DP. Physically-Based Glare Effects for Digital Images. In: Mair SG, Cook R, editors. Proceedings of the 22nd International ACM Conference on Computer Graphics and Interactive Techniques. New York, USA: ACM; 1995, p. 325–34. http://dx.doi.org/10.1145/218380.218466.
[6] Yoshida A, Ihrke M, Mantiuk R, Seidel H-P. Brightness of the glare illusion. In: Spencer SN, editor. Proceedings of the 5th Symposium on Applied Perception in Graphics and Visualization. New York, USA: ACM; 2008, p. 83–90. http://dx.doi.org/10.1145/1394281.1394297.

[7] Kolb C, Mitchell D, Hanrahan P. A realistic camera model for computer graphics. In: Mair SG, Cook R, editors. Proceedings of the 22nd annual conference on computer graphics and interactive techniques. New York, USA: ACM; 1995, p. 317–24. http://dx.doi.org/10.1145/218380.218463.
[8] Lee S, Eisemann E, Seidel H-P. Real-Time Lens Blur Effects and Focus Control. ACM Trans Graph 2010;29(4):65:1–7. http://dx.doi.org/10.1145/1778765.1778802.
[9] Steinert B, Dammertz H, Hanika J, Lensch HPA. General Spectral Camera Lens Simulation. Comput Graph Forum 2011;30(6):1643–54. http://dx.doi.org/10.1111/j.1467-8659.2011.01851.x.
[10] Kilgard MJ. Fast OpenGL-rendering of Lens Flares. 2000, URL https://www.opengl.org/archives/resources/features/KilgardTechniques/LensFlare/. (Accessed 4 July 2023).
[11] King Y. 2D Lens Flare. In: DeLoura MA, editor. Game Programming Gems. Charles River Media; 2000, p. 515–8.
[12] Maughan C. Texture Masking for Faster Lens Flare. In: DeLoura MA, editor. Game Programming Gems 2. Charles River Media; 2001, p. 474–80.
[13] Sekulic D. Efficient Occlusion Culling. In: Fernando R, editor. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley; 2004, p. 487–503.
[14] Oat C. A Steerable Streak Filter. In: Engel WF, editor. ShaderX3, vol. 2. Charles River Media; 2004, p. 341–8.
[15] Alspach T. Vector-based representation of a lens flare. 2009, US Patent 7, 526, 417.
[16] Chaumond J. Realistic Camera - Lens Flares. 2007, URL http://graphics.stanford.edu/cs348b-07/JulienChaumond/FinalProject. [Accessed 4 July 2023].
[17] Keshmirian A. A physically-based approach for lens flare simulation. Master's Thesis. University of California, San Diego; 2008, URL https://escholarship.org/uc/item/5n07m4p6. [Accessed 4 July 2023].
[18] Hullin M, Eisemann E, Seidel H-P, Lee S. Physically-Based Real-Time Lens Flare Rendering. ACM Trans Graph 2011;30(4):108:1–9. http://dx.doi.org/10.1145/2010324.1965003.
[19] Lee S, Eisemann E. Practical Real-Time Lens-Flare Rendering. Comput Graph Forum 2013;32(4):1–6. http://dx.doi.org/10.1111/cgf.12145.
[20] Hennessy P. Implementation Notes: Physically Based Lens Flares. 2015, URL https://placeholderart.wordpress.com/2015/01/19/implementation-notes-physically-based-lens-flares. [Accessed 4 July 2023].
[21] Hullin MB, Hanika J, Heidrich W. Polynomial Optics: A Construction Kit for Efficient Ray-Tracing of Lens Systems. Comput Graph Forum 2012;31(4):1375–83. http://dx.doi.org/10.1111/j.1467-8659.2012.03132.x.
[22] Hanika J, Dachsbacher C. Efficient Monte Carlo rendering with realistic lenses. Comput Graph Forum 2014;33(2):323–32. http://dx.doi.org/10.1111/cgf.12301.
[23] Walch A, Luksch C, Szabo A, Steinlechner H, Haaser G, Schwärzler M, et al. Lens flare prediction based on measurements with real-time visualization. Vis Comput 2018;34(9):1155–64. http://dx.doi.org/10.1007/s00371-018-1552-4.
[24] Kakimoto M, Matsuoka K, Nishita T, Naemura T, Harashima H. Glare Generation Based on Wave Optics. Comput Graph Forum 2005;24(2):185–93. http://dx.doi.org/10.1111/j.1467-8659.2005.00842.x.
[25] Scandolo L, Lee S, Eisemann E. Quad-Based Fourier Transform for Efficient Diffraction Synthesis. Comput Graph Forum 2018;37(4):167–76. http://dx.doi.org/10.1111/cgf.13484.
[26] Joo H, Kwon S, Lee S, Eisemann E, Lee S. Efficient Ray Tracing Through Aspheric Lenses and Imperfect Bokeh Synthesis. Comput Graph Forum 2016;35(4):99–105. http://dx.doi.org/10.1111/cgf.12953.
[27] Csoba I. OpenLensFlare: an Open-Source, Lens Flare Designing and Rendering Framework. In: Skala V, editor. WSCG 2017: Short papers proceedings. Computer Science Research Notes, Plzen, Czech Republic: Vaclav Skala–UNION Agency; 2017, p. 195–203.
[28] Hecht E. Optics, Global Edition. 5th ed. Harlow, UK: Pearson Education; 2016.
[29] Hillesland KE, Yang JC. Texel Shading. In: Bashford-Rogers T, Santos LP, editors. Eurographics 2016 – Short papers. Goslar, DEU: The Eurographics Association; 2016, p. 73–6. http://dx.doi.org/10.2312/egsh.20161018.
[30] Clarberg P, Toth R, Hasselgren J, Nilsson J, Akenine-Möller T. AMFS: adaptive multi-frequency shading for future graphics processors. ACM Trans Graph 2014;33(4). http://dx.doi.org/10.1145/2601097.2601214, 141:1–12.
[31] Marschner S. Fundamentals of Computer Graphics. 4th ed. Boca Raton, USA: A K Peters/CRC Press; 2015.
[32] Skala V. Barycentric coordinates computation in homogeneous coordinates. Comput Graph 2008;32:120–7. http://dx.doi.org/10.1016/j.cag.2007.09.007.
[33] Hormann K, Tarini M. A Quadrilateral Rendering Primitive. In: Akenine-Möller T, McCool M, editors. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. New York, NY, USA: Association for Computing Machinery; 2004, p. 7–14. http://dx.doi.org/10.1145/1058129.1058131,.
[34] Wachspress EL. A rational finite element basis. New York, USA: Academic Press; 1975.
[35] Loop CT, DeRose TD. A multisided generalization of Bézier surfaces. ACM Trans Graph 1989;8(3):204–34. http://dx.doi.org/10.1145/77055.77059.
[36] Csoba I, Kunkli R. Efficient Rendering of Ocular Wavefront Aberrations using Tiled Point-Spread Function Splatting. Comput Graph Forum 2021;40(6):182–99. http://dx.doi.org/10.1111/cgf.14267.
[37] Olsson O, Assarsson U. Tiled Shading. J Graph GPU Game Tools 2011;15(4):235–51. http://dx.doi.org/10.1080/2151237X.2011.621761.
[38] Zheng Q, Zheng C. NeuroLens: Data-Driven Camera Lens Simulation Using Neural Networks. Comput Graph Forum 2017;36(8):390–401. http://dx.doi.org/10.1111/cgf.13087.