Interactive Black-Hole Visualization

Annemieke Verbraeck and Elmar Eisemann

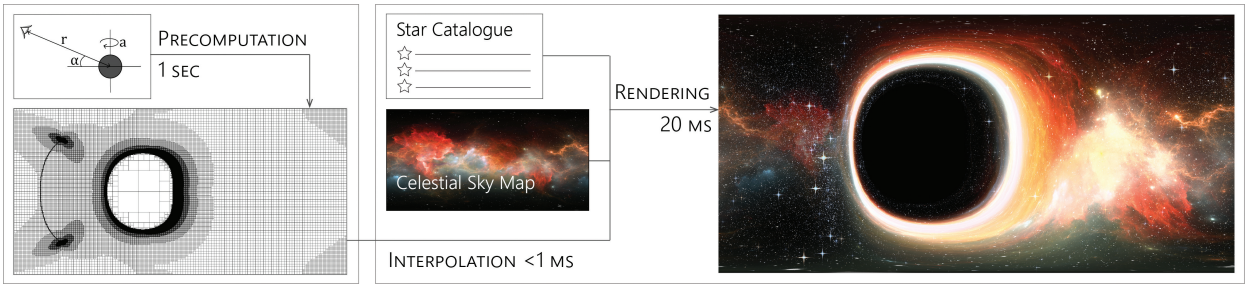

Fig. 1. We compute an adaptive grid (1 sec. per grid) for a given observer position (here in geodesic equatorial orbit with $r_{cam} = 5M$) and spin of the black hole (here $a/M = 0.999$). Grid values are interpolated to accommodate a given output resolution. The resulting frames (rendered at ~ 20 ms) capture the distortion of a celestial-sky map and star catalogue.

Abstract—We present an efficient algorithm for visualizing the effect of black holes on its distant surroundings as seen from an observer nearby in orbit. Our solution is GPU-based and builds upon a two-step approach, where we first derive an adaptive grid to map the 360-view around the observer to the distorted celestial sky, which can be directly reused for different camera orientations. Using a grid, we can rapidly trace rays back to the observer through the distorted spacetime, avoiding the heavy workload of standard tracing solutions at real-time rates. By using a novel interpolation technique we can also simulate an observer path by smoothly transitioning between multiple grids. Our approach accepts real star catalogues and environment maps of the celestial sky and generates the resulting black-hole deformations in real time.

Index Terms—Physical & Environmental Sciences, Engineering, Mathematics ; Computer Graphics Techniques; Algorithms.

1 Introduction

Astronomers have tried to visualize accurate representations of black holes for a long time, and recently an image was extracted from real observational data [28]. As we discover more about the universe, it is important to educate not only the professionals about these findings. Physically-plausible simulations of black holes make abstract principles tangible and enhance any educational experience, as in museums or planetariums, and could be included in games and movies.

Visualizing a black hole is challenging as light rays, traveling in its vicinity, diverge from straight paths. Consequently, a general-relativity ray tracer is needed to calculate the paths of light originating from stars and galaxies around a black hole towards the observer. Some of these rays loop around the black hole multiple times, resulting in very distorted visuals. To produce images of sufficient quality, billions of rays need to be traced, as for the 2014 movie Interstellar [13]. To visualize the view from a spaceship passing a black hole, the special effects studio DNEG worked together with astronomer Kip Thorne to create images that are visually appealing, high quality, and as scientifically accurate as possible.

Our method builds on the techniques developed for the Interstellar movie but targets higher performance. In this light, we present an efficient GPU mapping to accelerate the visualization. It produces physically-plausible simulations that can run on a commodity computer. We propose a two-step approach; we efficiently produce an adaptive grid that captures the black-hole distortion for varying observer positions using an order of magnitude fewer rays compared to tracing a ray for every pixel. Using an advanced interpolation method, we can transition between grids, which removes the need to update the grid in every frame. Images of higher resolution (e.g., FullHD) can then be generated at real-time rates using the computed grids. Our solution supports environment maps, representing nebula or dust, and star maps, the latter being a description of the location in spherical coordinates, emitted energy, and color of all visible stars from a given location. A separate map containing star locations delivers more accurate results than embedding them in the environment map, as a star projection is typically smaller than a pixel.

While a wealth of astronomy and physics work covers the underlying theory of black hole simulations, as well as related scientific insights, little attention has been given to the actual rendering procedures and most papers focus on the physics background [3, 5, 14, 23, 30, 31, 33]. We explain our rendering process in detail and discuss all optimizations and additional effects used to increase the realism and aesthetic of the imagery. Our method renders the black hole visuals at frame rates varying from real time to half a second per frame, depending on the star catalogue size. An overview of our method is shown in Fig. 1.

Specifically, our main contributions include a grid-based scheme that only involves a sparse ray tracing, an interpolation method to support camera movement, and an efficient integration process to avoid aliasing. Further, we present efficient implementations of effects to increase the realism of the visualization.

2 Black-Hole Physics

Here, we give an overview of black holes and their visual distortion effects. The text is based mainly on Chapter 16 of *An Introduction to Modern Astrophysics* [4] and *The Science of Interstellar* [29] to enable the interested reader to consult these for more information.

2.1 General Relativity

General relativity (GR) is a complex theory that cannot be fully detailed in this paper, the interested reader is referred to *Einstein Gravity in a Nutshell* [34] for an introduction to GR and to *Gravitation* as the authorative reference text on the field. To understand black holes,

• *Annemieke Verbraeck and Elmar Eisemann are with Delft University of Technology, The Netherlands. E-mail: {A.W.Verbraeck—E.Eisemann}@tudelft.nl.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

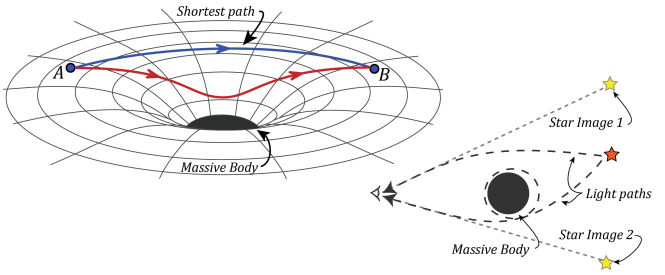

Fig. 2. **Left:** A planar 2D cut through spacetime deformed by a massive body. The shortest path (geodesic) from point A to point B is the blue curved line and not the 'straight' red line. **Right:** Bending of light by a massive body, with the actual postition and two of the observed positions of a star. Based on Figures 174 and 175 in [4].

curved spacetime is the most important concept explained by GR, together with the principle that light always travels along the shortest path. Mass, as in a black hole, causes spacetime to curve, which influences the way particles traverse spacetime. Even massless particles like photons are influenced because in curved spacetime the shortest route is no longer straight (Fig. 2, left). Shortest paths are called *geodesics* and can be found by solving a complex system of equations (detailed in supplementary materials) describing the local spacetime.

2.2 Black Holes

A black hole is the result of a very massive star that collapsed at the end of its lifetime. Any black hole can be completely described by three numbers: its mass, angular momentum, and electric charge. A Schwarzschild black hole is a type that does not have angular momentum nor electric charge, so the distortion of spacetime can be derived from mass M alone. A black hole is a spherical body, which implies that in this case, deformations are symmetric in all directions. Kerr black holes have additional angular momentum (spin). Their rapid rotation induces a phenomenon that warps the surrounding spacetime, reducing the spherical symmetry to axis symmetry about the rotation axis. This produces a region called the ergosphere in which the pull of the rotation is so high that all particles within have to co-rotate. The spin is expressed indirectly by its ratio to the mass with a constant that cannot exceed one (for the paper, all illustrations use $a/M = 0.999$, the video shows different parameters).

2.3 Distortion Effects

A black hole significantly distorts the surrounding: light rays arrive at the observer from different angles than from where they originated (Fig. 2, right). Further, the velocities necessary for a space ship to stay in stable orbit (avoiding descent into the black hole) are a significant fraction of the speed of light. This causes the observer in the ship to experience additional view distortions.

Gravitational Lensing

Even multiple images of the same star can be observed, as light rays in different directions from the same star can travel along different geodesics arriving at the observer. Inversely, two photons close to each other on the camera might actually originate from two stars with a large angular distance (relative to the observer) between them. An originally small distance or area can appear stretched and its brightness changes proportionally [18], which is referred to as gravitational lensing.

Redshift

Traveling at high speeds changes the perceived frequency of a wave (Doppler effect); opposing the camera movement, lightwaves become compact, shifting the color towards blue (blueshift), in the other direction they elongate and appear reddish (redshift). Both phenomena are typically grouped under the term $redshift$. When photons pass by a black hole, the effect of time slowing down also causes a gravitational redshift in their frequency. Close to the black hole's center, one observes the *shadow*. It is the purely black portion, where the gravitation effects are so severe that time slows to a stop and no light can escape.

2.4 Curved-Spacetime Ray tracing

To visualize the distorted space around a black hole, typically a ray tracing method follows the geodesics (Fig. 2, right) back in time. A system of differential equations describes these geodesics [13]. The equations define photon travel directions at any point in space. Integrating them from the observer back to $-\infty$, gives the direction that the photon originated from. We chose Runge-Kutta-Cash-Karp [24] for the numerical integration of the differential equations, because of its stability and adaptive step size. As all rays are independent, the ray tracing can be parallelized. Further, even before tracing one can also detect if a ray is caught in the shadow of the black hole. The differential equations we use do not provide solutions inside the ergosphere (Sect. 2.2), so in our simulation the camera cannot descend into the black hole. Details of the ray-tracing method can be found in the supplementary material.

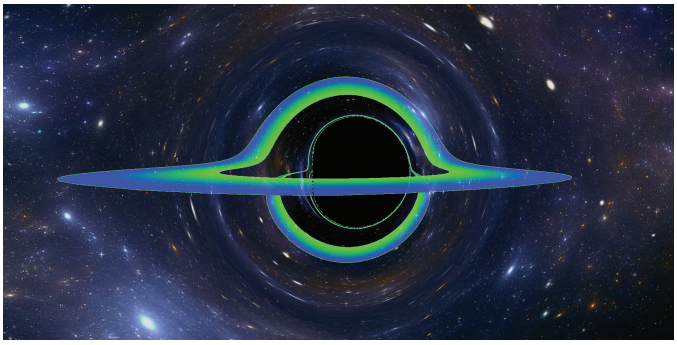

Fig. 3. Our rendering of a colored accretion disk between $r = 9M$ and $r = 18M$ around a Kerr black hole, seen through a pinhole camera ($r_{cam} = 30M$, inclination of $\theta_{cam} = \pi/2 - \pi/32$, view angle $\pi/2$).

3 Related Work

Black hole simulations started from an astronomical perspective to confirm theoretical work and telescope observations, which implied that a camera was typically far away from the black hole. In 1978 Luminet [17] computed an image of a rotating black hole illuminated by an accretion disk around it. Adding a disk is also used in more recent works [3, 20, 23], as it makes the image distortion caused by the black hole become very visible. While our method can be used to simulate an accretion disk located at finite distance (Fig. 3) by testing its intersection during ray tracing, our focus is on the background distortion. Additionally, such a disc would act as a light source that is much closer to the observer and its brightness would render stars and interstellar dust invisible.

GRay [5] and Odyssey [25] are parallelized GPU raytracers visualizing step-by-step photon travel near a black hole in real time but unlike us, they did not produce an actual observer image. They did compute the distortion of a grid image for benchmarking. It took them approximately 2 seconds for 256×256 rays on the GPU, while our solution on the CPU requires less than a second to trace this amount of rays, including adaptive grid construction. Our Runge-Kutta implementation uses an order of magnitude fewer total steps than Odyssey's, which can be contributed to our aggresive step size increase following small integration errors, and the difference in application, as their photon path visualization might require smaller steps and intermediate storage.

Schwarzschild black holes and their induced distortions are fully symmetric, which makes the involved equations simpler than for Kerr black holes. Weiskopf et al. [32] described the first interactive visualization of a Schwarzschild black hole distorting a background with objects assumed to be infinitely far away. They exploited the spherical symmetry that reduces the two-dimensional situation to a one-dimensional case by rotation. In a later paper, Müller [19] described an OpenGL

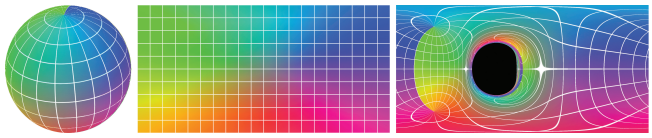

Fig. 4. **Left:** Celestial sky defined on a sphere and as an environment map (equirectangular projection). **Right:** The equirectangular projection of the distorted celestial sky caused by a Kerr black hole.

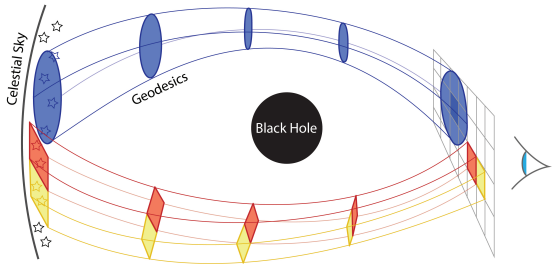

Fig. 5. In blue: Ray bundle tracing in *Interstellar* [13]. In red and yellow: Our method traces single rays from pixel corners to form ray polygons.

implementation that used the same strategy, allowing an observer to freely move around a non-spinning black hole, using a lookup table based on an analytic solution. Müller and Weiskopf [21] also presented a real-time visualization for a local observer viewing the stellar sky behind a Schwarzschild black hole. They did not compute the distortion of the whole image plane. Instead, they used the Schwarzschild metric to find the light paths connecting each star with the observer and computed the star's appearance change in terms of luminosity, size and light frequency. Riazuelo [26] used a similar (though not real-time) method that can simulate an observer falling into the Schwarzschild black hole. He provided an extensive explanation of how gravitational effects change the visuals and what methods were chosen to make the visualization look realistic and aesthetically pleasing.

Making realistic visualizations of distorted spacetime becomes even more interesting with the recent advancements in stereoscopy and virtual reality. Hamilton et al. [9] showed that visualizing a black hole in 3D is nontrivial. An accurate physical rendering in curved spacetime results in two images that are too different from each other to be correctly processed by our brain. They solved this problem by using an approximation where the two views are rendered as if the viewpoints are in flat spacetime. This is also the approach that Davelaar et al. [7] took for their research on visualizing an accretion flow into a black hole in VR. All solutions rely on offline rendering to produce the visuals. Our solutions could provide such imagery in real time, which is particularly interesting in VR to support interactivity and to be able to adapt the stereo baseline to accommodate varying eye distances.

Kobras et al. [15] present an inspiring method for general relativistic image-based rendering. The setup matches ours with a camera in the distorted spacetime and a 360-environment map at infinity. They also map pixels to regions but perform only axis-aligned filtering. Additionally, we handle adaptive computations, view interpolation, gravitational blueshift and lensing simulations, and actual star maps.

A very precise visualization was made for the movie Interstellar [13]. Their research on simulating a black hole with an accretion disk and the distorted stellar sky has an emphasis on quality of the produced frames rather than runtime. The work used the Kerr metric to describe spacetime and takes location and movement of the local observer into account. They derived a set of differential equations that describe light geodesics for single rays, and an improved version with ray bundles. Ray tracing a batch of rays per pixel resulted in a completely black image, as the chance of hitting a star (similar to a point light) is small and even for nebulae aliasing occurs. To this extent, their ray bundle tracing uses a volume instead; a circle centered on each pixel, which deforms during tracing and ends up as an elliptical region on the celestial sphere. All stars in this region were then integrated to compute the output pixel value. Unfortunately, the extra equations and variables needed to keep track of the deformations of the ray bundles is large, which makes the algorithm costly and difficult to reproduce. Our solution examines an alternative, using only the equations for single rays, which sacrifices some quality but reaches much higher performance.

4 Our Approach

Our solution consists of two main parts. The first is the grid computation on the CPU, which derives an adaptive grid that captures the distortion for a given viewpoint. The second is the processing of the celestial sky in real time on the GPU to simulate the distortion effects of a black hole. We first describe the variables and setting (Sect. 4.1) to provide the necessary background information. We continue with an overview of a basic approach (Sect. 4.2), before describing our complete method with optimizations (Sect. 4.3). Finally, we propose several extensions (Sect. 4.4). We show how to support a star map with precise sub-pixel star locations, as well as optical effects to increase the realism of the produced images. A schematic representation of the star map rendering with the advanced algorithm and all extensions is shown in Fig. 12.

4.1 Setting

Our approach supports arbitrary camera projections but, by default, we use an equirectangular projection of the whole 360 panorama with a pixel grid of equally-spaced spherical coordinates ($0 < \phi < 2\pi$, $0 < \theta < \pi$), with θ increasing downward and ϕ increasing counterclockwise. If wanted, this 360 view could be remapped onto different camera models, e.g., a pinhole camera with user defined parameters for resolution and field of view. As additional parameters, the user can define the spin of the black hole and the location, direction and speed of the camera. In our standard setup, the camera is always in orbit around the black hole.

While the observer might be close to the black hole, stars and other celestial objects are assumed far away so that their distance can be considered infinite. This celestial-sky background is then defined as a virtual sphere around the observer with an infinite radius (Fig. 4 shows a celestial-sky environment map). Consequently, we make use of spherical coordinates throughout the algorithm:

1. The camera is located at a position $(r_{cam}, \theta_{cam}, \phi_{cam})$ relative to the black hole, where r_{cam} is expressed in black hole mass M.
2. The camera moves with speed β in direction $(r_{dir}, \theta_{dir}, \phi_{dir})$.
3. Light rays arrive at the **c**amera position from a direction (θ_c, ϕ_c).
4. Light rays originate from positions on the celestial **s**ky (θ_s, ϕ_s).

4.2 Basic Algorithm

The simplest version of a rendering algorithm shoots one ray from the center of each pixel and traces it back to a position (θ_s, ϕ_s) on the celestial sky. If we determine the pixel to be part of the black hole shadow, it is colored black, otherwise the pixel color is determined by a lookup in the environment map.

There are two problems with this simple approach. First, to simulate a change in the position or direction of the camera in real time, ray tracing needs to be performed for every frame. As the workload of tracing rays for pixels is very high, it becomes very costly. Second, tracing a single ray per pixel and performing a lookup in the celestial-sky map leads to visible aliasing due to stretch and compression by the black hole. In the next two sections, we introduce the concepts that form the a basic algorithm that overcomes these two hurdles.

4.2.1 Grid

As a first step towards an interactive simulation, we reduce the number of rays that need to be produced. To this end, we construct a grid on a sphere around the camera, where every vertex on the grid is a direction (θ_c, ϕ_c) of incoming light rays for $0 < \theta_c < \pi$ and $0 < \phi_c < 2\pi$. For each grid vertex, we shoot the corresponding ray and store the

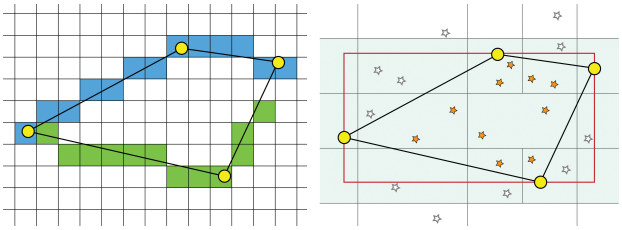

Fig. 6. **Env. Map, Left:** Polygon of a pixel projected onto the environment map. *Basic:* Average pixels within the polygon. *Advanced:* Precompute the sum from the top and store it. The difference of a blue pixel at the minimum row intersected by the polygon and the green pixel at the maximum yield the sum in the column. **Star Map, Right:** Projected pixel polygon in the stellar sky. *Basic:* Test all stars and if they are inside the polygon, sum their energy. *Advanced:* Compare the polygon's AABB (red) against a binary tree to select the stars in the shaded tree leaves. Only these are tested against the polygon, yielding the orange star set.

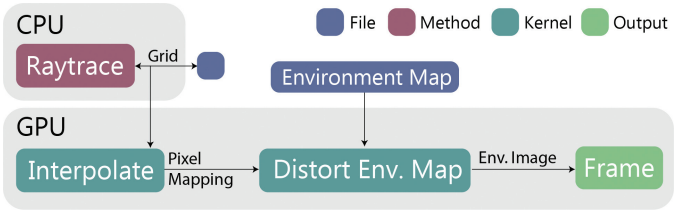

Fig. 7. Overview of the basic algorithm.

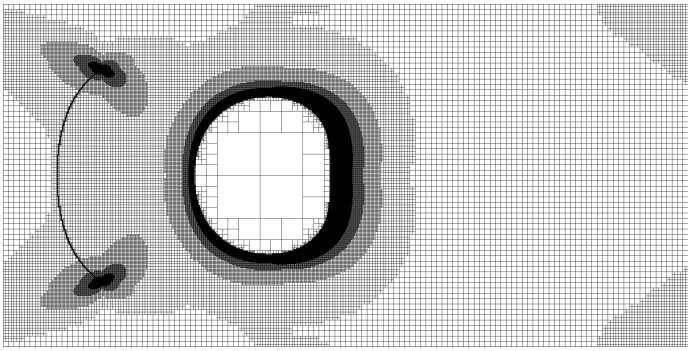

Fig. 8. Grid levels as computed by adaptive method up to level 10, y-axis has range $[0,\pi]$ and x-axis $[0,2\pi]$ (The camera is in a geodesic equatorial orbit with $r_{cam} = 5M$.)

celestial-sky coordinates (θ_s, ϕ_s). As the distortion of space around a black hole is smooth, it lends itself well for an interpolation approach. By interpolating the grid vertices, we can approximate corresponding (θ_s, ϕ_s)-coordinates for any ray direction (θ_c, ϕ_c). This means that for a given camera position, we only trace rays once to construct the grid. Then, for each pixel in every image frame, we can locate the grid cell containing the pixel center and perform a linear interpolation of the stored celestial-sky positions from the cell's corner vertices with respect to the camera-pixel center. This is independent of the camera model (e.g., pinhole), as the grid is omnidirectional and supports any camera orientation from the given position. Additionally, when the camera circles around the black hole spin axis at a fixed distance, the grid also does not require any updates due to the axisymmetric black-hole distortion. We only need to apply a uniform shift of the ϕ_s-coordinates. Other positional changes require an update of the sparse grid. Furthermore, when the camera is positioned in the equatorial plane, the distortion is symmetric with respect to this plane, and only the top half of the grid needs to be traced, halving the computation time.

4.2.2 Ray Polygons

To counter the aliasing caused by a single ray per pixel, Interstellar traced ray bundles around a pixel to determine an ellipsoid area on the celestial sky, which was integrated to find the pixel color [13]. Unfortunately, ray bundle tracing is much more complicated and costly than tracing single rays.

We can avoid the bundle-tracing costs, by realizing that a pixel is a square area defined by its four corners. Tracing single rays through those corners and connecting their celestial-sky coordinates, results in a four-sided polygon, which can be integrated just like the ellipsoid (Fig. 5). In this way, neighboring pixels map to adjoining polygonal regions on the celestial-sky and the number of traced rays remains identical. To determine the camera-pixel color, we average the contribution of each environment-map pixel that is contained in the polygonal region (Fig. 6) weighted by solid angle to counteract the spherical-parametrization stretching near the poles. If one pixel corner lies in the black hole shadow, we set the output to black. Care has to be taken when pixel corners project close to both sides of the $\phi = 0 = 2\pi$ border in the celestial-sky parametrization. We adjust for this problem by cutting the polgyon along the seam into two parts.

Combining this solution with the previously introduced grid is straightforward. Instead of pixel centers, we now compute the interpolated results for pixel corners. Changing from one data point per pixel (the center) to four (the corners) adds no workload to the interpolation, as every corner is shared by four pixels. Fig. 7 shows an overview of the basic implementation of the algorithm.

4.3 Advanced Algorithm

The previously-described algorithm eliminates the most expensive computations but still can be improved significantly. First, the use of a regular grid is not efficient; the spacetime deformation induced by the black hole is continuous outside the black-hole shadow and only some areas require much detail. Consequently, we propose the use of an adaptive grid (Sect. 4.3.1). Second, a linear interpolation of grid vertices does not follow the continuous distortion well and leads to artifacts. Hermite splines, defined based on the celestial-sky coordinates of four grid vertices, provide a much better approximation (Sect. 4.3.2). Further, we address the challenge of moving freely around the black hole by interpolating a set of grids (Sect. 4.3.3). Finally, due to the distortion, a pixel can lead to a large polygonal region on the celestial sky and integrating all contained pixels can be expensive. We provide an optimization that avoids looping over each individual pixel (Sect. 4.3.4).

4.3.1 Adaptive Grid

We start with a coarse grid, two cells each representing half of the sphere, that are initially subdivided like a quad tree to level L_{min}, unless the cell is entirely contained in the black hole shadow. Next, we will progressively refine these cells further. To this extent, we trace the ray for each current grid vertex and compute its location on the celestial sky. For each grid cell, we then approximate the size of its region on the celestial sphere by determining the length of the diagonals based on the locations of the opposing vertex pairs of the cell. If this length exceeds a threshold, we subdivide the cell by adding a grid vertex in the cell's center and at the midpoints of its edges. The subdivision process is then continued with the newly formed cells up to a level L_{max}. In practice, all examples in this paper and video use $L_{min} = 6$ and $L_{max} = 10$. We chose the polygon diagonals as subdivision criterion, as these are quickly calculated using only the grid cell itself and even a rough approximation of the level of deformation results in a highly functional adaptive grid. We experimentally determined the threshold of 0.015 radians, which results in the best balance between traced number of rays and interpolation error (see Sect. 5).

To use the adaptive grid for interpolation, we need to find the cell containing a given ray direction (θ_c, ϕ_c). The construction of the grid is hierarchical, as every cell that is adaptively refined is divided into

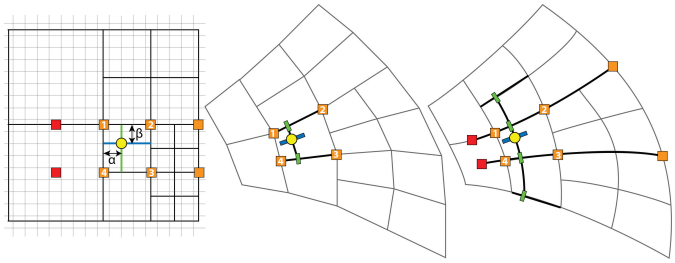

Fig. 9. Linear and spline interpolation methods. **Left:** Adaptive grid with pixel positions in grey overlay. The yellow dot is the pixel corner to be interpolated. The green and blue lines show the percentual horizontal (α) and vertical (β) displacement in the grid cell, respectively. The orange vertices are traced, the red ones are not. **Middle, Linear:** Interpolate the polygon edges $\{1,2\}$ and $\{3,4\}$ with α. Connect the resulting points and interpolate with β. **Right, Spline:** Create two splines, with inner points $\{1,2\}$ and $\{3,4\}$ and end points taken from equidistant vertices in the adaptive grid (values for red vertices are interpolated), and evaluate them at α. Use these as inner points to create a third spline, together with the linearly-α-interpolated, equidistant grid edges as end points and evaluate this spline at β.

four child cells. We can use the resulting tree structure to quickly descend to the required leaf cell. We start the process at the largest cell (either $0 < \phi < \pi$ or $\pi < \phi < 2\pi$) containing (θ_c, ϕ_c). We check the cell for children. If there are none, we found the leaf cell, and otherwise we know the cell has four child cells and we continue with the cell containing (θ_c, ϕ_c). This process is repeated until a leaf cell is found.

We wish to store the grid in a data structure that allows us to quickly find the cell to interpolate. At the same time, it should be suitable for use on the GPU and not take up large amounts of memory, to avoid making the transfer of the grid from CPU to GPU a bottleneck. To this end, we do not use an explicit tree structure, where the same grid vertex would need be stored for cells of different levels. Instead we store the representation in a perfect hash table [16], which enables fast lookups and stays compact, by only storing each grid vertex once. Hash tables use keys to retrieve values. In our case, the values are (θ_s, ϕ_s)-coordinates on the celestial-sky and the keys represent grid vertex positions with (θ_c, ϕ_c)-coordinates. For this specific structure, keys need to be pairs of integers, so we map grid-vertex positions to an (i, j)-index:

$$i = \frac{\theta_c}{\pi} N, \quad j = \frac{\phi_c}{2\pi} M, \tag{1}$$

where M and N are the horizontal resp. vertical number of grid vertices in a maximally refined grid. We use a stored value of (-1, -1) to indicate that a corresponding grid vertex is part of the black hole shadow. To descend in the implicit tree encoded by the hash map, we test for a given cell if the vertex at the center of this cell exists. If it does, the cell must have been subdivided and we can virtually proceed with the corresponding next cell.

A perfect hash structure consists of two tables and a lookup is a two-step process. First the key (i, j) is used for a lookup in an offset table, returning an offset (x, y) which is added to the key. This new key $(i+x, j+y)$ is used for the lookup in the table holding the (θ_s, ϕ_s)-values. The construction of these tables is such that a lookup using any index key will succeed and return a value in the table. This means that indices of grid vertices that have not been traced (like the ones in the middle of a leaf cell), will return a value that belongs to another grid vertex. To solve this problem, we also store the (i, j)-index of the grid vertex in the value table together with the (θ_s, ϕ_s)-value. Whenever we perform a lookup, we first check whether the key matches the stored index. Due to the extreme sparseness of the grid, this method is even more memory efficient than the use of a bitmask.

4.3.2 Spline Interpolation

Interpolating the celestial-sky coordinates of the grid vertices bilinearly does not follow the curves of the distortion accurately. Unlike spline interpolation, this method only considers the distortion of the current grid cell, when computing pixel to polygon mapping. This results in a discontinuity between the mapping of consecutive pixel corners in neighboring cells. Comparing the squared (θ_s, ϕ_s) distances of a fully traced level 10 grid (2048×1024) to a bilinearly-interpolated adaptive grid reveals a cumulative squared error of 0.369 rad^2. This error gets transferred to the projected pixel polygons overlapping with the cell borders and might lead to interpolation artifacts (Fig. 10).

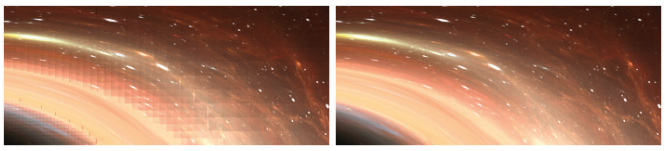

Fig. 10. Part of distorted image with lensing effect. **Left:** Linear interpolation causes artifacts that show the size of the adaptive grid. **Right:** Spline interpolation gives a smooth result.

Using a spline interpolation yields a much better result, as the smooth splines better capture the smooth distortions (Fig. 9). The error for the above example reduces to 0.008 rad^2, which gives an average squared error of less than 1E-8 rad^2 per grid vertex (not counting the vertices that exactly line up with the grid).

In practice, we employ cubic Hermite splines as these are smooth and efficient to calculate with four points (q_0, q_1, q_2, q_3):

$$\begin{aligned} p(t) &= h_{00}(t)q_1 + h_{10}(t)m_1 + h_{01}(t)q_2 + h_{11}(t)m_2 \\ m_1 &= (q_2 - q_0)/2; \quad m_2 = (q_3 - q_1)/2 \end{aligned} \tag{2}$$

Here, $h_{00}, h_{10}, h_{01}, h_{11}$ are the Hermite basis functions and the resulting spline is defined from $t = 0$ to $t = 1$ between inner points $p(0) = q_1$ and $p(1) = q_2$, with m_1, m_2 their respective tangents, calculated using the end points q_0 and q_3.

To explain spline interpolation for our grid cells, we first recap the simpler bilinear interpolation. We define the position of the ray direction in the grid cell c with two interpolation values, indicating the percentage of horizontal (α) and vertical (β) displacement within the cell. The four corners of the grid cell, in clockwise order starting from the top-left corner, map to four coordinates $\{p_1, ..., p_4\}$ forming a polygon on the celestial sky. With bilinear interpolation, first the two edges of the polygon (p_1, p_2) and (p_3, p_4) are interpolated using α, the resulting coordinates are then connected, and finally this line is interpolated using β:

$$\beta(\alpha p_1 + (1-\alpha)p_2) + (1-\beta)(\alpha p_3 + (1-\alpha)p_4) \tag{3}$$

The process for spline interpolation is similar. We also focus on the edges (p_1, p_2) and (p_3, p_4) first, and construct their splines, s_1 and s_2. We evaluate them at α, and use the resulting coordinates to construct the vertical spline s_3, which we evaluate at β for our final result. However, the coordinates $\{p_1, ..., p_4\}$ are not enough to construct s_1 and s_2, as they are only the inner spline points. The spline end points are retrieved from neighboring grid cells. As grid vertices have no knowledge of their nearest neighbours, we select the two end points such that all four spline points are equidistant in the grid. For s_1 and s_2, we select positions that lie at one cell width distance to the right and left of their inner points. Similarly, for the spline s_3, we linearly interpolate (at α) the horizontal grid edges that lie one cell width above and below the cell.

The chosen positions (even when shifted by one cell width) do not necessarily correspond to grid vertices. In this case, we obtain the result by interpolation as well. We use the same spline interpolation technique for this step, but if this process again encounters missing points, we adopt linear interpolation to avoid a long chain of recursive calls. When part of a spline is inside the shadow, linear interpolation is used as well.

To further improve the quality of interpolation, we post process the grid after construction. At locations where t-vertices appear (a larger

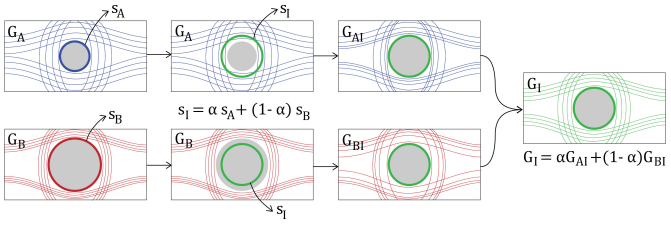

Fig. 11. **First**: Two grids G_A, G_B to interpolate. Grid G_A has a smaller black hole shadow s_A than G_B with shadow s_B. **Second**: We interpolate the new shadow size s_I to be at percentage α between s_A and s_B. **Third**: From the change in shadow size we calculate the offsets for lookup in the grids. These offsets have the effect of squeezing (G_{AI}) or stretching (G_{BI}) the grid around the new shadow s_I. **Last**: For every required lookup G_{AI} and G_{BI} are blended via α.

grid cell next to smaller cells), we interpolate the edge of the larger cell, and adjust the vertices of the smaller cells that lie on this edge accordingly. It does not give an analytic match for every point along the line between different sized grid cells, but, in practice, for larger grid-cell sizes, where we would expect larger deviations, we have by definition less distortion, which leads to a good balance. The result is a very close match and this adjustment only needs to be done once and takes little compute time.

While spline interpolation requires additional lookups in our adaptive grid compared to linear interpolation, it results in a much better approximation for the pixel corners, causing the filtering quality for the environment map to be significantly improved.

4.3.3 Grid Interpolation

So far, we have considered a single adaptive grid, but for camera movement other than rotation around the symmetry axis of the black hole, a new grid is needed. We allow the user to indicate a movement direction and determine an adaptive grid for a target location in this direction. Alternatively, the grids for a set of positions can be precomputed beforehand. To transition between different grids, we propose a special interpolation scheme, which we describe here.

To interpolate the lookups in two grids G_A and G_B at a percentage α, one might consider a linear interpolation $G_I = \alpha G_A + (1-\alpha) G_B$. However, when the two grids have black-hole shadows of varying size or shape, artifacts become clearly visible. For example, when the position of grid A is further from the black hole (larger r_{cam}) than grid B, grid A will have a smaller shadow than grid B. This means some lookups in G_A will return celestial-sky locations, while the same lookup location in G_B returns a (-1,-1) indicating the shadow. A weighted average would not be meaningful. We will solve this problem by virtually warping the images such that the black-hole shadows coincide.

Let G_X be a grid and c_X the center of the shadow in G_X. We parameterize the shadow boundary by an angle $0 < \gamma < 2\pi$ around c_X by $e_X(\gamma)$. We define $r_X(\gamma) := |e_X(\gamma) - c_X|$, the distance from the center to the shadow boundary.

For brevity, we will drop the parameter γ in the following. When interpolating between grids G_A and G_B with a value α to obtain a grid G_I, we would expect $r_I = \alpha r_A + (1-\alpha) r_B$ and $c_I = \alpha c_A + (1-\alpha) c_B$. Without loss of generality, we assume $r_A < r_B$. To limit the influence of the warp, we will only warp the textures in a region that is $1.5 r_I$ from c_I. For this, we impose that $1.5 r_A > r_B$, which is typically not a strong restriction for a reasonable camera movement. Hence, any lookup beyond $1.5 r_I$ can be done by linearly interpolating both grids. For a lookup position $p \in [r_I, 1.5 r_I]$, we introduce $w = (p - r_I)/(0.5 * r_I)$. w varies from zero to one for p from r_I to $1.5 r_I$. Now we define the warped lookup $p_A^w = w * 1.5 * r_I + (1-w) r_A$, which virtually moves the shadow boundary of G_A to r_I and falls back to the non-warped version beyond $1.5 r_I$. A similar definition can be provided for p_B^w. With this, we can define $G_I(p) = \alpha G_A(p_A^w) + (1-\alpha) G_B(p_B^w)$. These steps are illustrated in Fig. 11.

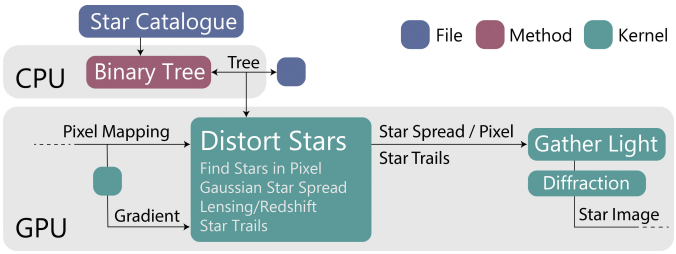

Fig. 12. Overview of the functions and kernels used to create the Star Image, with all optimizations and extensions added.

4.3.4 Celestial-Sky Integration

To determine the camera-pixel color, we trace the edges of the projected polygon on the celestial-sky environment map and average all pixels that fall within the region. In practice, we can first determine the columns that the polygon overlaps and for each column we check the vertical minimum and maximum position inside the polygon and sum everything in between. By using a technique similar to summed-area tables [10], we only need two lookups per pixel column. The idea is to preprocess the environment map image and store in each pixel the sum of its own color plus the color values of all pixels located above it in the same column. With this map, if we subtract an upper from a lower pixel in the same column, the difference yields the exact sum between both (Fig. 6).

The distortion becomes stronger the closer pixels are to the black hole shadow. Some pixels in this region map to a polygon containing almost the whole celestial sky. The summation process above would still take a disproportionate amount of time in this case, while, with an average of so many pixels, the selection does not need to be very precise. Therefore, when a polygon covers more than half the environment map width, we use an axis-aligned bounding quad of the polygon instead, which eliminates the need to find minimum and maxium positions for every column.

4.4 Extensions

In this section, we discuss additional measures to increase the realism of the produced imagery.

A. Star Catalogue

In the previous sections, we have seen how to retrieve coordinates on the celestial sky for each camera-pixel corner and how to use these to retrieve the pixel color via lookups in the celestial-sky environment map. While an environment map is a good representation for nebulae, interstellar dust, and galaxies, it is not for stars. A star in the environment map would always be at least one pixel large, which can lead to an unrealistic stretch, as distant stars rather act like point lights and appear under a much smaller solid angle. We thus propose a seperate rendering solution for stars, which we detail in the following.

We use a star map that contains the stars' position in radians (θ, ϕ) in the celestial sky, the apparent magnitude m_*, and the B-V color index. Instead of summing the environment map pixels, a simple adaptation is to test all stars for containment in the corresponding polygon (Fig. 6). For each contained star, we calculate the temperature from the B-V color index and magnitude (assuming black body radiation [2]), convert this to an RGB color (with the help of a lookup table [6]) and multiply it by the stars brightness b (calculated from the magnitude as $b = 10^{-0.4 m_*}$). The pixel color is then defined as the sum of all contained stars' brightness-adapted RGB colors.

Testing all stars against a polygon becomes quickly costly. To accelerate the computations, we build a hierarchy on the star positions in form of a uniform kD-tree, where the first division is at $\phi = \pi$, the second at $\theta = \pi/2$, and so forth. During rendering, we traverse this tree and collect all leaf nodes that overlap with the axis-aligned bounding quad of the polygon. We then test all leaf-node stars against the boundaries of the polygon (Fig. 6). In practice, we noticed that we

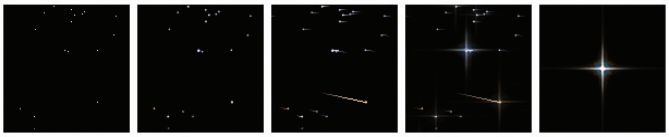

Fig. 13. Star effects: default stars, gaussian star spread, star trails, diffraction and the image used as convolution kernel [27].

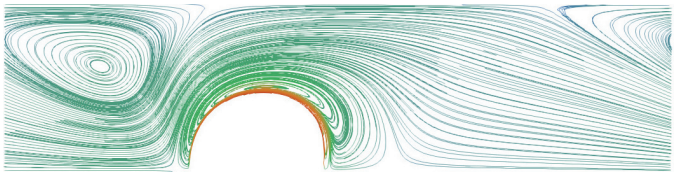

Fig. 14. Upper half of the symmetric image showing streamlines of the stars when rotating around the black hole. The color indicates velocity (blue is low, red is high), which is highest around the shadow. (Camera is in a geodesic equatorial orbit with $r_{cam} = 5M$)

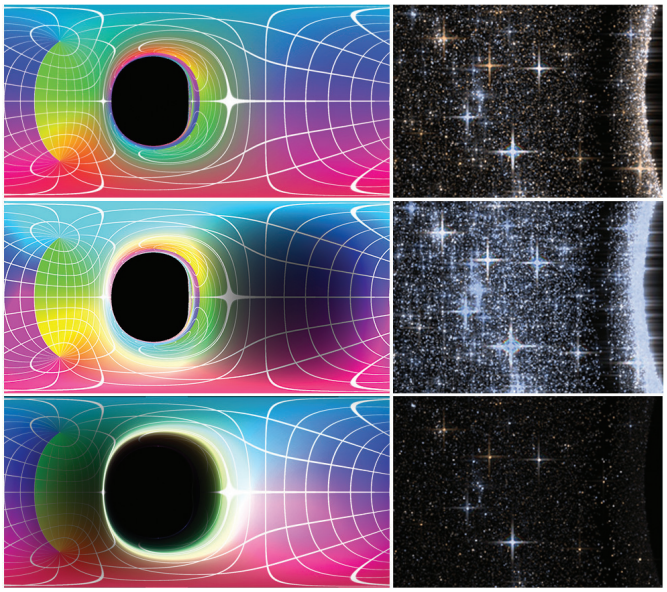

Fig. 15. Redshift and Lensing effect on environment map and stars. From top to bottom: reference image without effects, redshift effect, lensing effect. The redshift on the map is the extreme case to clearly show the effect. The star images are a close-up next to the shadow. The bleeding of light into the shadow in the first two images is caused by the diffraction of many stars. The lensing effect dims this area of the image. (Camera is in a geodesic equatorial orbit with $r_{cam} = 5M$)

can stop on a higher level in the tree; when the node overlaps more than eighty percent, we assume that all its leaf nodes overlap. This prevents spending too much time on descending into the tree and checking all leaf nodes, when a high percentage will be included.

B. Tone Mapping

Our input maps typically contain high-dynamic-range data. All values are gamma corrected before computations and gamma encoded again before storing. Additionally, we perform a simple gamma tone mapping to simulate an observer state. Hereby, we can indirectly control the number of visible stars by boosting their perceived brightness.

C. Gaussian Star Spread

While stars are point-light sources, having them contribute to a single pixel appears unnatural and makes different brightness levels hard to discern. For this reason, we evaluate the overlap of a Gaussian kernel centered on the star with the pixel centers of the surrounding pixels. With a standard deviation of 0.5 and cutoff of 1.5 pixels the Gaussian affects maximally eight surrounding pixels. The resulting weights are used to determine the contribution of the star to the corresponding pixels. We noticed that a normalization of these weights is not necessary and there was no visible change in brightness over consecutive frames in an animation. To integrate this operation efficiently into our pipeline, we do not actually render stars as a Gaussian kernel. Instead, we store the contribution to the eight neighbors inside of the single pixel containing the star. Once all stars have been processed, we collect in each pixel the contributions from the neighbors. This gathering strategy proved more efficient than a splatting solution.

D. Star Trails

When the camera orbits the black hole at high speed, some stars move so far between frames that they seem to jump from one screen position to another. To counter this problem, we add trails for these stars. For every frame, we store the pixel location of the stars, and draw a trail whenever a star in the next frame is more than one pixel further than its previous location. Nevertheless, every star appears multiple times in the rendered image and close to the black hole it is hard to discern which new image of a star corresponds to a previous location. To match the correct stars, we compute a vector field that approximates their movement, while the camera orbits the black hole. During orbit, the ϕ_s values per pixel corner change, but the θ_s values stay constant, meaning the stars will travel along these θ_s-isolines (Fig. 14). To find the correct previous pixel location of the star, we choose the one that follows the movement of the vector field the closest, which is the location that has the smallest angle between its associated vector in the vector field and its potential star trail. To avoid false matches, no trail is drawn when this angle is larger than $\pi/4$, or if the pixel distance of the trail is larger than $1/25$ of the image width.

E. Diffraction Spikes

Starbursts are artifacts that appear as a result of diffraction around the support vanes in the telescope [8]. As we are very used to seeing images with this effect, it is a natural addition. To simulate diffraction we created another kernel, that works on the image output after the star light gathering step. A texture with a diffraction pattern is used as a convolution filter on the matrix of pixel values, where only pixels that have a high enough brightness are affected. For our experiment we chose the 100×100 texture from Scandolo et al. [27] (Fig. 13).

F. Redshift and Lensing

As stated in Sect. 2.3, gravitational lensing causes light sources that are projected to a larger area to become proportionally brighter and vice versa. To calculate this brightness change, the ratio between the solid angle of the pixel and the solid angle of its projected polygon on the celestial sky is computed. Further, the redshift phenomenom changes the wavelength of incoming light. It is computed from the speed and direction of the observer and the location of the pixel on the camera sky [13]. We apply the following equations [26] to determine the observed temperature and magnitude of the stars:

$$m_*^{obs} = m_* + 4\log(1+z) - \log f, \quad T_*^{obs} = \frac{T_*}{1+z}, \tag{4}$$

where f is the fraction between the solid angles and m_* the magnitude (brightness), z the redshift and T_* the temperature of the star. The calculation of the redshift is based on previous work [13] but explained in the supplementary material for completeness.

It is impossible to adjust the colors of the pixels in the environment map for the redshift and lensing in a physically correct way, because unlike for stars, the spectral info of the objects in the map is not known. We follow Riazuelo's example [26] and only change the luminance of the output pixels, mapping the shift to a brightness function that goes asymptotically to 0 for red and 1 for blue. Using a similar function as for stars results in a large part of the image either very dark or very

Fig. 16. Heat map of the upper-half full level 10 grid, showing the number of Runge-Kutta integration steps performed per grid vertex. (camera is in a geodesic equatorial orbit with $r_{cam} = 5M$)

bright, while this function keeps the distortion visible. Fig. 15 shows the lensing and redshift effects on the map and stars.

Adding the lensing effect to a linearly interpolated grid causes severe artifacts. Using spline interpolation removes most, but some small artifacts remain, caused by discontinuities between the solid angle of the projected polygons of neighboring pixels. To smooth out these discontinuities, we compute a matrix of image size that holds the solid angle of its polygon, for every pixel. We filter this matrix with a simple averaging kernel and use it when lensing is applied.

5 Results

All experiments were conducted on an Intel Core i7-8700 processor with 16GB RAM and an Nvidia RTX 2080 Ti graphics card. Our method for the tracing is implemented on the CPU, the interpolation and rendering steps are performed in CUDA [1] where the result is mapped to an OpenGL buffer to display the frames on the screen. Parallel computations are employed per ray on the CPU, per pixel corner for interpolation and per pixel for rendering. When the camera is set to a pinhole view, the view direction can be changed interactively. Our standard setup is a Kerr black hole with a spin of 0.999 and an observer that is positioned at $r_{cam} = 5M$ in the equatorial plane. Putting the observer in this plane means that we are able to exactly calculate the speed for a geodesic orbit and make the simulation as accurate as possible. As discussed before, the distortion is symmetric with respect to the plane in this case, so only half of the grid needs to be computed, all other computations take the same time as for the non-symmetric case. Our standard output is a 360 view with an image size of 1920×960, because this matches full HD for most modern screens.

5.1 Grid Computation

Ray tracing a symmetric adaptive grid at level 10 takes less than $0.5s$ and produces a serialized 1MB file with the perfect hash table of 47848 rays at single precision. A non-adaptive grid of the same level takes around 5 seconds for 20 times more rays. On average rays take 40 Runge-Kutta integration steps in $3-5\mu s$ to reach the celestial sky with a maximum of 350 steps. Fig. 16 shows the number of integration steps per grid vertex for a full level 10 grid.

The adaptive grid file size could be reduced by switching the number type but in its current state, the transfer time to the GPU is less than a millisecond. The amount of traced rays depends the maximum level of the grid, the grid threshold, and the distance of the camera. At first sight it would seem that a larger distance requires more rays, as the size of the shadow gets smaller. However, a camera closer to the black hole observes a higher level of distortion, causing the adaptive grid to refine more. The paths to be traced are more complex in this case as well, so computation times for larger observer radii are shorter, as shown in Fig. 19. It also contains the graph illustrating the trade-off between the amount of traced rays and interpolation errors for different threshold settings. For our level 10 grid, we decided on a value of 0.015 rad, as a higher threshold (e.g. 0.02 rad) results in visible errors in the image, while a much lower (e.g. 0.001 rad) refines the grid so much that it negates its benefits.

When visualizing the full 360 view on an image not larger than 2048×1024, computing a higher level than 10 is unnecessary, as it would contain details more precise than the pixel size. For the pinhole camera view higher levels can be beneficial, especially to show enough detail close to the shadow (Fig. 17). We can easily decide the optimal grid level by ensuring that the solid angle of the (smallest) grid cells is comparable in size to that of the output image pixels.

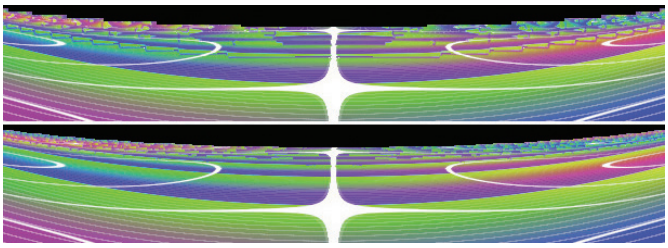

Fig. 17. Difference in quality between level 10 and level 12 adaptive grid. Closeup of the shadow edge (rotated 90 degrees).

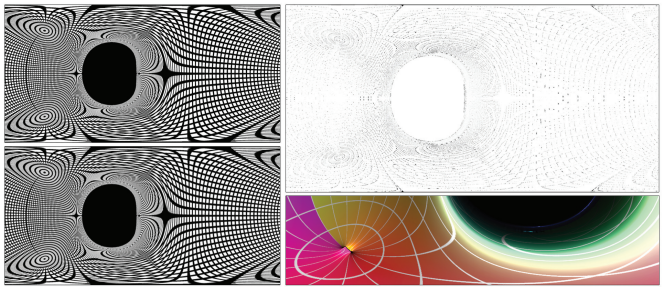

Fig. 18. **Left:** Image using a grid for $r_{cam} = 5.1M$ and the same image with an interpolated grid using $r_{cam} = 5.0M$ and $r_{cam} = 5.2M$. **Right, top:** Difference image multiplied by 4 and inverted to improve readability. **Right, bottom:** Lensing artifacts at poles and nearby the shadow when interpolating $\theta_{cam} = \frac{1}{2}\pi$ and $\theta_{cam} = \frac{17}{32}\pi$.

5.2 Interpolation

Interpolation is performed for every frame, to allow for changing the view direction when using a pinhole camera. For our standard setup it takes 0.68 ms to interpolate using splines and 0.55 ms for linear. The projected pixel area computation and filtering (used to avoid artifacts caused by lensing) adds only ~ 0.1 ms. As there is a large difference in quality, but not in speed, spline interpolation is the default.

Interpolating between Grids

The interpolation between grids runs on the GPU as well, and is performed as an extra step before spline interpolation, to calculate the warping of the lookup locations. To work smoothly, multiple grids need to be uploaded to the GPU. When changing the distance of the camera to the black hole (r_{cam}), for example, we trace a new grid for every increment of $0.2M$ along the trajectory. For two grids we calculate the location of their shadow edges once, at the first frame that is positioned in between these grids, which adds 0.3 ms to the computation. The interpolation kernel now needs to find the warped positions of two grids instead of one and alpha blend them, which adds 2.5 ms.

Our grid interpolation technique is quite effective. Changing the distance to the black hole particularly benefits from our special warping solution and the distortion is well captured. Even for relatively large displacements, the interpolated result approaches the reference rendering (Fig. 18), only some minor differences are visible to the naked eye. We tested with a black and white grid image as an the environment map to enable a better judgement of local distortions, and while these lines do not match up perfectly, the deviations remain within a few pixels. Comparing the (θ_s, ϕ_s) values obtained from a grid computed at camera radius $5.1M$ with the ones obtained from the interpolated grid halfway between a radius of $5M$ and $5.2M$, we find that the average squared error per pixel corner mapping is only 0.0045 rad^2. When changing the inclination of the camera (θ_{cam}) with respect to the spin axis of the black hole, the interpolation causes subtle brightness artifacts due to the lensing at the poles of the celestial sky (Fig. 18).

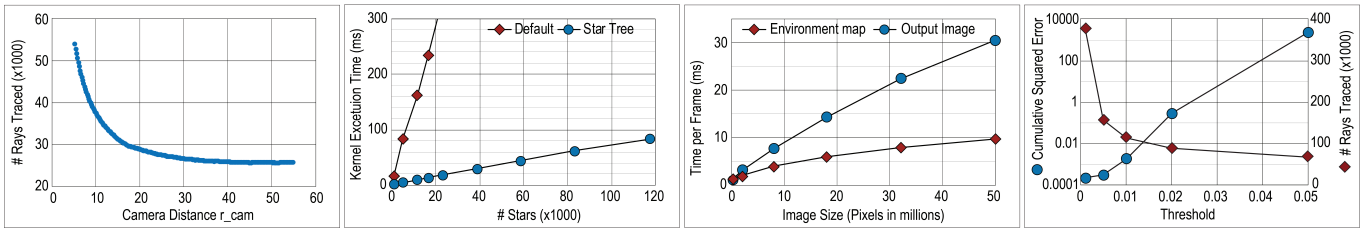

Fig. 19. All graphs use: black-hole spin $a/M = 0.999$, camera in geodesic equatorial orbit at $r_{cam} = 5M$, map size 4096 × 2048, image size 1920 × 960, level 10 grid, unless specified otherwise. From left to right **1:** Effect of camera distance on the number of traced rays. **2:** Effect of the star tree on the Star Distortion kernel's runtime for different catalogue sizes. **3:** Effect of the environment map resolution (with constant output size), as well as output resolution (with constant map size) on render time per frame (without stars). **4:** Effect of threshold parameter for the adaptive grid on the error (cumulative squared error compared to full grid) and amount of traced rays.

Table 1. Runtimes: Interpolation and Rendering Kernels (ms)
a) 4096 × 2048 pixels; b) brightest 5047 stars from HYG catalogue

Standard Interpolation	Grid-Grid Interpolation	Distort Env.Map[a)]	Distort Stars[b)]	Diffraction
< 1	2.5	3.5	9	4.5

5.3 Rendering

Rendering distorted environment maps runs in realtime, at around 3.5 ms for the GPU kernel to complete, with a map size of 4096 × 2048. Summing the map as a preprocessing step results in a crucial speedup, without it the kernel takes more than 200 ms. The bounding quad approximation for pixels that map to large polygons ensures that the GPU does not waste time—it reduced the computation by ~ 2 ms with no visible difference. The influence of output image size as well as map size on performance can be seen in Fig. 19. The effect of the summed map is visible here as well, as the time per frame increases approximately linearly with the map width instead of the total amount of pixels. The size of the output image has more influence on runtime; for every million pixels added there is an 1.2 ms/frame increase.

For the distortion of the stellar sky, we performed our tests with a file of approximately 100.000 stars from the HYG database [22]. With use of the binary tree structure, a full HD image depicting a stellar sky with around 5000 stars computes in realtime at ~ 9 ms per frame vs 90 ms per frame without (speedup of ~ 10). Distorting the whole catalogue takes 95 ms per frame. Fig. 19 shows that the runtime is linear in the amount of stars with and without binary star tree. Adding star trails to the image does not take a significant amount of extra time, and diffraction takes 4.5 ms with the current filter implementation. The amount of stars as well as the radius in which pixel light is distributed over neighbouring pixels govern performance. For full HD images spreading the star light to only the direct neighbours gives the best results, a larger spread makes the stars look blurred. When computing for 4k frames and up a larger spread of light can be preferable to make the stars properly visible on the screen.

6 Discussion

While previous work relied on either much higher computation times, or simpler configurations like the Schwarzschild metric, our approach works for Kerr black holes and can be executed on standard desktop devices. It shows that visually appealing and convincing simulations can be efficiently approximated. While we cannot claim the same accuracy as previous approaches, our evaluation of the adaptive grid shows that for similar quality, only a fraction of the rays need to be traced and large parts of the environment can be interpolated. Indeed, the computation of our precomputed grids is relatively fast (around the order of a second) and its loading time represents milliseconds. Consequently, it is possible to retrieve or compute new maps during navigation. If a trajectory is known, our solution can also compute intermediate grids based on a user defined budget to deliver a suitable rendering in a predefined amount of time. Nevertheless, an entirely free navigation with potential movement heuristics remains future work.

The ray-tracing equations in our implementation are not defined inside the ergosphere, making it impossible to descend into the black hole. There are visualizations that cope with this situation by changing the equations to the inner Kerr metric [9, 26]. In a similar way, the view through a wormhole could be visualized using our code with the appropriate differential equations [12]. Descending into the black hole and looking through a wormhole are future extensions that will make our solution more widely applicable for educational use.

Implementation-wise, CUDA proved more efficient for most operations but the final render is done in GLSL. The transfer between both is a bottleneck. With advances in GLSL, the standard pipeline could become more useful for some components (e.g., star trails rendered via line primitives). Diffraction also uses a disproportionate amount of time when a lot of bright pixels are present. Instead of a filtering process, we intend to experiment with sprite rendering. Furthermore, coarser grids during motion and general spatio-temporal upsampling strategies could be considered [11]. In the latter context, subpixel evaluations might prove useful, as we currently employ lensing on a per-pixel basis of the accumulated value.

Similarly, our solution involves CPU and GPU components but we tested a full GPU implementation. A full level 10 grid takes ~ 2 seconds ($1\mu s$ per ray) to trace on the GPU. Parallized CPU code takes 3-5μs per ray, a factor of three. Adaptive grids are much faster to compute and tracing takes only 0.2s on the CPU. Interestingly, the adaptive solution does not lead to a good GPU utilization, due to divergence (0.7s). Hence, we see our current compute distribution as a good tradeoff - the CPU executes the task faster and leaves the GPU idle for the actual rendering task. Different compute strategies remain future work.

7 Conclusion

We presented a GPU-based solution to visualize black holes. While black holes are a singularity, the distortions that they generate are largely smooth and can be well captured with an adaptive sampling strategy that evaluates the distortion only where needed. The underlying function can be well reconstructed using specialized interpolation techniques that make use of the black-hole characteristics. Special interpolation techniques are also very important when interpolating different camera positions. Other phenomena, such as redshift and starburst can be successfully applied in an image-based postprocess. Our solution thus combines various techniques to accelerate black-hole distortion images significantly, making a high-quality interactive visualization possible, even on standard hardware.

Acknowledgments

The authors wish to thank Markus Billeter and Alexander Verbraeck for their insights on interpolation and implementation, Baran Usta for feedback and support, and the reviewers for their detailed comments. This work was funded by the VIDI grant NextView of the NWO Vernieuwingsimpuls.

REFERENCES

[1] CUDA C programming guide. docs.nvidia.com/cuda/cuda-c-programming-guide, 2019.

[2] F. J. Ballesteros. New insights into black bodies. *Europhysics Letters*, 97(3):1–6, Feb. 2012. doi: 10.1209/0295-5075/97/34008

[3] K. Beckwith and C. Done. Extreme gravitational lensing near rotating black holes. *Monthly Notice of the Royal Astronomical Society*, 359(4):1217–1228, Jun. 2005. doi: 10.1111/j.1365-2966.2005.08980.x

[4] B. W. Carroll and D. A. Ostlie. *An Introduction to Modern Astrophysics*. Benjamin Cummings, 1996.

[5] C. Chan, D. Psaltis, and F. Özel. GRay: A massively parallel GPU-based code for ray tracing in relativistic spacetimes. *The Astrophysical Journal*, 777(1):1–9, Oct. 2013. doi: 10.1088/0004-637X/777/1/13

[6] M. Charity. Blackbody color datafile. www.vendian.org/mncharity/dir3/blackbody/, 2001.

[7] J. Davelaar, T. Bronzwaer, D. Kok, Z. Younsi, M. Mościbrodzka, and H. Falcke. Observing supermassive black holes in virtual reality. *Computational Astrophysics and Cosmology*, 5(1):1–17, Dec. 2018. doi: 10.1186/s40668-018-0023-7

[8] E. Everhart and J. W. Kantorski. Diffraction patterns produced by obstructions in reflecting telescopes of modest size. *Astronomical Journal*, 64(1275):455–462, Dec. 1959. doi: 10.1086/107973

[9] A. Hamilton and G. Polhemus. Stereoscopic visualization in curved spacetime: Seeing deep inside a black hole. *New Journal of Physics*, 12:1–25, Dec. 2010. doi: 10.1088/1367-2630/12/12/123027

[10] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555, Sep. 2005. doi: 10.1111/j.1467-8659.2005.00880.x

[11] R. Herzog, E. Eisemann, K. Myszkowski, and H.-P. Seidel. Spatio-temporal upsampling on the GPU. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, pp. 91–98, Feb. 2010. doi: 10.1145/1730804

[12] O. James, E. von Tunzelmann, and P. Franklin. Visualizing Interstellar's wormhole. *American Journal of Physics*, 83(6):486–499, Jun. 2015. doi: 10.1119/1.4916949

[13] O. James, E. von Tunzelmann, P. Franklin, and K. S. Thorne. Gravitational lensing by spinning black holes in astrophysics, and in the movie Interstellar. *Classical and Quantum Gravity*, 32(6):1–41, Feb. 2015. doi: 10.1088/0264-9381/32/6/065001

[14] M. D. Johnson, A. Lupsasca, A. Strominger, G. N. Wong, S. Hadar, D. Kapec, R. Narayan, A. Chael, C. F. Gammie, P. Galison, D. C. M. Palumbo, S. S. Doeleman, L. Blackburn, M. Wielgus, D. W. Pesce, J. R. Farah, and J. M. Moran. Universal interferometric signatures of a black hole's photon ring. *Science Advances*, 6(12):1–8, Mar. 2020. doi: 10.1126/sciadv.aaz1310

[15] D. Kobras, D. Weiskopf, and H. Ruder. Image based rendering and general relativity. *The Visual Computer*, 18:250–258, Mar. 2002. doi: 10.1007/s003710100145

[16] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, Jul. 2006. doi: 10.1145/1141911.1141926

[17] J. P. Luminet. Image of a spherical black hole with thin accretion disk. *Astronomical Astrophysics*, 75(1-2):228–235, May 1979.

[18] C. W. Misner, K. S. Thorne, and J. A. Wheeler. *Gravitation*. W. H. Freeman, 1973.

[19] T. Müller. Image-based general-relativistic visualization. *European Journal of Physics*, 36(6):1–11, Nov. 2015. doi: 10.1088/0143-0807/36/6/065019

[20] T. Müller and J. Frauendiener. Interactive visualization of a thin disc around a Schwarzschild black hole. *European Journal of Physics*, 33(4):955–963, Jul. 2012. doi: 10.1088/0143-0807/33/4/955

[21] T. Müller and D. Weiskopf. Distortion of the stellar sky by a Schwarzschild black hole. *American Journal of Physics*, 78(2):204–214, Feb. 2010. doi: 10.1119/1.3258282

[22] D. Nash. The HYG database. www.astronexus.com/hyg, 2006.

[23] O. Porth, H. Olivares, Y. Mizuno, Z. Younsi, L. Rezzolla, M. Moscibrodzka, H. Falcke, and M. Kramer. The black hole accretion code. *Computational Astrophysics and Cosmology*, 4(1):1–42, Dec. 2017. doi: 10.1186/s40668-017-0020-2

[24] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[25] H. Pu, K. Yun, Z. Younsi, and S. Yoon. Odyssey: A public GPU-based code for general-relativistic radiative transfer in Kerr spacetime. *The Astrophysical Journal*, 820(2):105–116, Mar. 2016. doi: 10.3847/0004-637X/820/2/105

[26] A. Riazuelo. Seeing relativity-I: Ray tracing in a Schwarzschild metric to explore the maximal analytic extension of the metric and making a proper rendering of the stars. *International Journal of Modern Physics D*, 28(2):60, Jan. 2019. doi: 10.1142/S0218271819500421

[27] L. Scandolo, S. Lee, and E. Eisemann. Quad-based Fourier transform for efficient diffraction synthesis. *Computer Graphics Forum*, 37(4):167–176, Jul. 2018. doi: 10.1111/cgf.13484

[28] The Event Horizon Telescope Collaboration et al. First M87 event horizon telescope results. I. The shadow of the supermassive black hole. *The Astrophysical Journal Letters*, 875(L1):1–31, Apr. 2019. doi: 10.3847/2041-8213/ab0ec7

[29] K. Thorne. *The Science of Interstellar*. W. W. Norton & Company, 2014.

[30] S. U. Viergutz. Image generation in Kerr geometry, I. Analytical investigations on the stationary emitter-observer problem. *Astronomy and Astrophysics*, 272(22):355–377, May 1993.

[31] F. H. Vincent, T. Paumard, E. Gourgoulhon, and G. Perrin. GYOTO: A new general relativistic ray-tracing code. *Classical and Quantum Gravity*, 28(22):1–18, Nov. 2011. doi: 10.1088/0264-9381/28/22/225011

[32] D. Weiskopf, M. Borchers, T. Ertl, M. Falk, O. Fechtig, R. Frank, F. Grave, A. King, U. Kraus, T. Müller, H. Nollert, I. R. Mendez, H. Ruder, T. Schafhitzel, S. Schär, C. Zahn, and M. Zatloukal. Explanatory and illustrative visualization of special and general relativity. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):522–534, Jul. 2006. doi: 10.1109/TVCG.2006.69

[33] X. Yang and J. Wang. YNOGK: A new public code for calculating null geodesics in the Kerr spacetime. *The Astrophysical Journal Supplement Series*, 207(1):1–32, Jun. 2013. doi: 10.1088/0067-0049/207/1/6

[34] A. Zee. *Einstein Gravity in a Nutshell*. Princeton University Press, 2013.