School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Real-time Ray Tracing of a Schwarzschild Black-Hole

Dylan Curzon

Submitted in accordance with the requirements for the degree of
BSc Computer science

2021/22

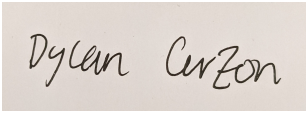
COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (27/04/2022)
Source Code (Zipped)	`bholetrace.zip`, `bholetrace.tar.gz`	Uploaded to Minerva (27/04/2022)
Source Code	Online Code Repository	https://github.com/curz46/bholetrace

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) 

Acknowledgements

- Luca Bertozzi (https://github.com/ekardnam) - helped via email exchange to bridge the gap in my knowledge of theoretical physics. He provided a formulation of the Binet equation for the path of a light ray in a black hole's gravitational field, which is used by the raytracer to integrate numerically when calculating the deflection of light. A few of the (referenced) resources that were ultimately crucial to the project's success were suggested by him too. Many thanks.
- David Head (project supervisor) - provided guidance and support, especially in the project's early stages to help clarify goals and objectives.
- My neighbour's cat Sean (https://imgur.com/a/LCUI7sw) - provided emotional support. Also exceptionally cool.
- For a couple additional acknowledgements (code repositories used as reference) - see the `README` file in the project source tree.

Honourable Mentions

- https://rantonels.github.io/starless/ - A fantastic resource on relativistic raytracing and the appearance of accretion disks. Helped with my understanding considerably.
- The Gravitation and Cosmology book by Weinberg (2008) - The diagrams provided are excellent and the use of the closest approach r_0 when finding the scattering angle $\Delta\varphi$ came directly from this book. A great reference for the mathematics of relativistic effects.

Contents

Abstract

The following paper details the background research, method and implementation details of a black-hole ray tracing application. It is heavily inspired by Verbraeck and Eisemann (2021) but focuses on performance and simplicity over a higher quality result. The researchers used a hybrid technique, computing and caching the deflection of light on the CPU, then leveraging the GPU to achieve very good rendering performance. This paper's approach uses OpenGL compute shaders to perform the vast majority of the computation with minimal constraints on the surrounding environment. The results are sufficiently accurate to be used as a teaching tool, while the implementation is kept simple and can be easily extended.

1 Background Research

Black-hole rendering is a fascinating field with a surprising amount of depth. It might not be immediately clear why a visual representation of a black-hole would be of value in academics, but the interstellar giants serve as classic example of the effect of space-time curvature. A tangible first-person perspective can captivate a viewer and do wonders for their understanding in a way numbers and textbook definitions can't. To this end, black-hole rendering is valuable in the way it can teach concepts that are hard for people to visualise, such as the bending of light.

1.1 Schwarzschild black-holes

This paper focuses on a particular solution to Einstein's gravitational field equations called the Schwarzschild metric, which assumes that the massive body in question has an electric charge and angular momentum equal to zero. This means that the black-hole is not rotating.

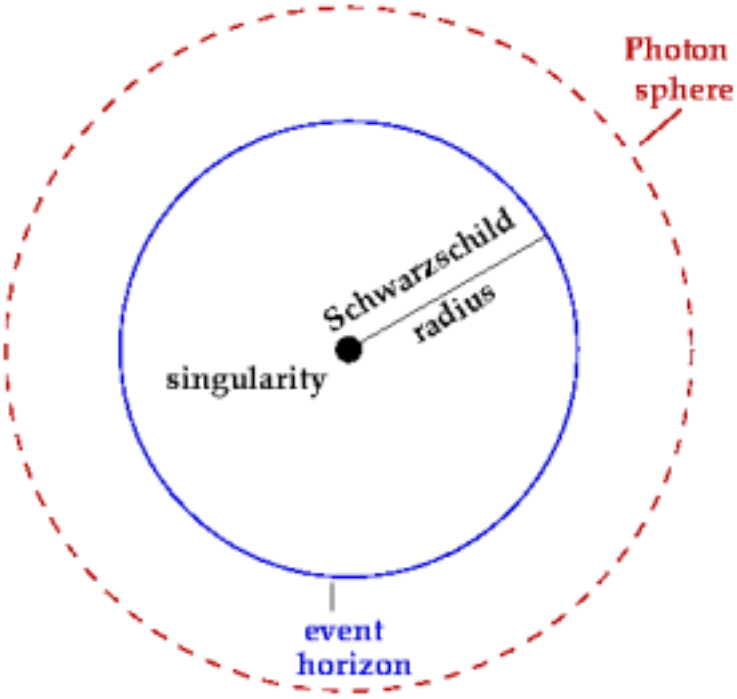


Figure 1: A drawing of a Schwarzschild black-hole (Francis, 2011).

It should be noted that a non-rotating black-hole is extremely unlikely to exist in reality. To understand why, consider how a figure skater spins faster as they pull their arms inwards. The dying star that (more than likely) gave rise to a black-hole will have a non-zero angular momentum. As the star collapses, it will rotate faster and faster. So any black-holes observed in reality will be spinning rapidly, and the Schwarzschild metric is merely a theoretical approximation.
The Schwarzschild black-hole has several notable properties we are interested in - its singularity, event horizon and photon sphere. The singularity is a point-like object where all of the mass of the black-hole is considered to be situated. The event horizon is the spherical theoretical boundary where light, once it crosses, could never escape. This is also the point where the escape velocity would be faster than the speed of light, which is physically impossible to achieve based on our understanding of the natural laws of our

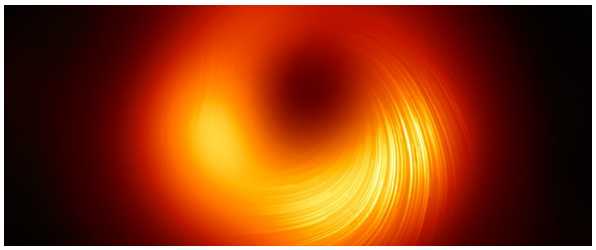


Figure 2: A real life image of a black-hole in polarized light (EHT-collaboration, 2019).

universe. So once an object crosses the event horizon, it never comes back. The photon sphere is the apparent size of the black-hole when observed in reality. The gravitational field at this point is so strong that photons (a theoretical particle of light) must orbit the black-hole or fall into it. The radius of the photon sphere is equal to $\frac{3}{2}r_s$, where r_s is the event horizon radius.

Due to the immense gravitational forces around a black-hole, there is a visible deflection of light. The effect has been minimally observed in reality due to the astronomical distances to black-holes close to us. The image shown in Figure 2 shows one of the few real life insights we have into a black-hole's appearance so far. Another significant attribute of black-holes is the accretion disk. Much like other massive solar bodies such as stars and planets, the gravitational pull of a black-hole will often cause matter (dust, rocks and gas) to orbit it. The immense tidal forces around a black-hole is unlikely to permit a nearby star or planet to remain in one piece, but there will more than likely be a substantial amount of matter in orbit around it. The compression will cause such matter to raise in temperature resulting in an emission of electromagnetic radiation. In other words, the accretion disk will glow. (Yiu, 2020)

1.2 Existing Work

One of the earliest attempts to render a black-hole using computer graphics was by Luminet (1979). The paper focuses on the appearance of an accretion disk when distorted by the black-hole, and touches on central concepts in the field such as use of the Binet equation as an approximation for computing the deflection of light. As can be expected given limited computational resources their result was coarse, but still provided a visual insight into the black-hole's deflection of objects close to the event horizon. More modern attempts (Müller and Weiskopf (2010), Riazuelo (2019)) can provide a significantly higher quality result. However, neither approach can be considered to render in real-time. Riazuelo (2019) achieved processing times of 1 CPU day for a 1 minute long clip, but noted that each frame could be rendered independently (and therefore in parallel.) They note that CPU time when rendering is mostly used for their (unoptimised) realistic star rendering, which is quite an involved process and out of scope for this paper. Overall, the paper goes into a lot of depth and is useful as a resource for the mathematics of light deflection, though the equations provided are not very well explained. They also suggest 4th-order Runge-Kutta numerical integration for finding solutions, which is similarly used by Verbraeck and Eisemann (2021). The approach by Müller and Weiskopf (2010) offers some level of real-time performance but requires precomputation of lookup tables. They do not mention if this imposes a significant limitation on the motion of the camera during runtime. Their approach also allows the observer (camera) to free-fall into the black-hole as it would happen in reality, which requires a specific formulation of equations that are well defined inside the event horizon. The method

presented by Verbraeck and Eisemann (2021) is much more relevant since they had an emphasis on creating a real-time solution that could be used to demonstrate relativistic effects. A large part of their paper is spent explaining how they maintain good quality without sacrificing performance. This is in part done by caching the deflection result at several predetermined distances then using an advanced interpolation process for the exact distance required. Their method is underpinned by an adaptive grid, which refines the accuracy of the cached result where distortion is predicted to be severe, and reduces it where the approximations can be coarse (e.g. directly toward the photon sphere.) Each of the mentioned works consider other relativistic effects which become significant when close to the black-hole, namely the frequency shift of light due to the continuous change in diffraction. They also introduce advanced raytracing or interpolation techniques, especially in the context of rendering stars, to achieve a higher quality result. Verbraeck and Eisemann (2021) considers the Kerr black-hole, which allows the massive body to rotate and is therefore more realistic. The raytracer this paper produces uses the Schwarzchild metric and allows aliasing and rendering artifacts as an acceptable limitation to keep the implementation simple and fast. The black-hole will not have an accretion disk, though this could be added with a little extra work. The project aims to provide an open and easily extended implementation that could act as an introduction to the field.

1.3 Relativistic Raytracing

The standard approach to render the perspective of an observer in front of a black-hole is a special form of raytracing. Raytracing is a rendering technique where a

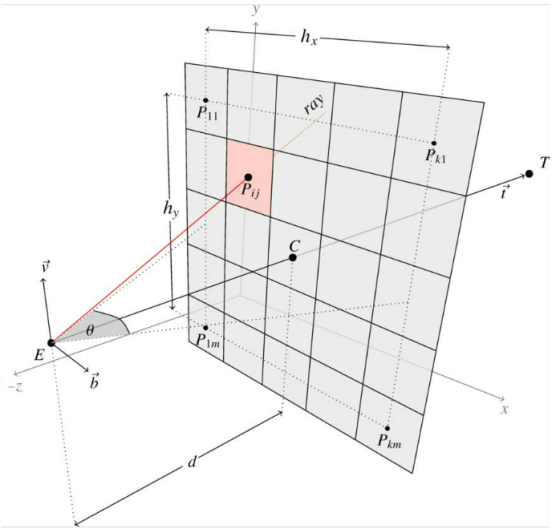


Figure 3: A visual representation of the maths behind raytracing (Wikimedia, 2019)

"ray" is traced through each pixel on the screen as if the camera was physically in the scene (see Figure 3.) The ray is then tested for intersections with the environment to determine the colour of that pixel. However, a black-hole distorts the path of light rays as they pass close to it, causing the image of a celestial sky surrounding a black-hole to appear very distorted. In order to render this distortion effectively, a raytracing method needs to follow the path a ray of light would take if it were in the simulated gravitational field. After computing the deflection, the resultant trajectory of the light ray can be used to make further intersection tests, eventually determining the colour of the pixel. A convenient way to do this is via the Binet equation, which determines orbital motion as a function of a polar angle (φ). The following is a formulation of that equation, making a few assumptions to simplify the mathematics.

$$\frac{d^2u}{d\varphi^2} = -u + \frac{3}{2}u^2$$

The change in u where $u = \frac{1}{r}$ (and r is the distance from the black-hole's singularity) is encoded in this equation as a function of the angle φ. The radius r here is expressed in

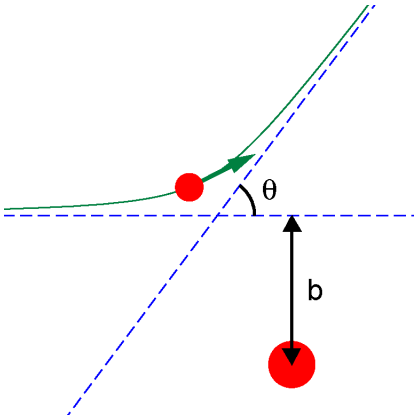


Figure 4: A sketch of the impact parameter and scattering angle in a field. (Wikimedia, 2007)

terms of the event horizon radius. Note that we let $2MG = 1$ and $c = 1$ to simplify the equation. (M is black-hole mass, G is the gravitational constant and c is the speed of light.) This means that the mass of the black-hole in our program is fixed and distances are not to scale, but since the purpose of the raytracer is to demonstrate deflection this is fine.

We can numerically integrate this function to find the value for u (and therefore r) at any φ, setting some initial conditions for our particular ray of light. When $\varphi = 0$:

$$\frac{du}{d\varphi} = \frac{1}{b}$$
$$u = 0$$

where b is the impact parameter. Setting $u = 0$ here is equivalent to asserting that the ray comes from infinity. This makes the method presented an approximation inherently since this is never the case, though it may be possible to change this in order to get a more accurate result. See Figure 4 for a depiction of how the impact parameter relates to the path of the light ray. This is equivalent for the case of a gravitational field, but with a scattering angle that bends the ray towards the black-hole. We can take b to be the closest distance to the black-hole if the ray of the

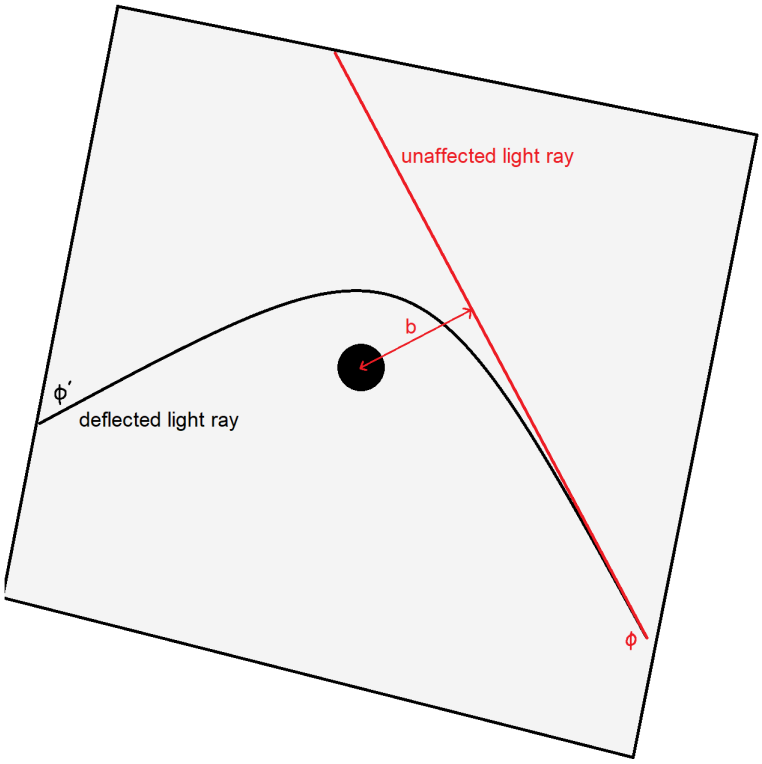


Figure 5: Light deflection is symmetric and occurs within a plane.

light were to travel completely straight, with no deflection. Note that the scattering angle θ is not the same as our variable φ, but they are related such that total change in $\varphi = \theta + \pi$.

So now our problem is reduced to finding the impact parameter b for any light ray, and then numerically integrating our equation to find the change in trajectory. The first problem is trivialised by the consideration that the deflection of a black-hole is symmetric and is bound to a consistent plane. See Figure 5 for a representation of this. The gray box denotes the plane of the deflection for the light ray. This plane is how we can take the Binet equation, which gives us the Schwarzschild radius for a particular polar coordinate, and apply it in a 3-dimensional simulated field. This is trivial to compute based on the position of the camera, the original direction of the ray and the position of the black-hole (which for simplicity is always at the origin.)

To numerically integrate our equation, the

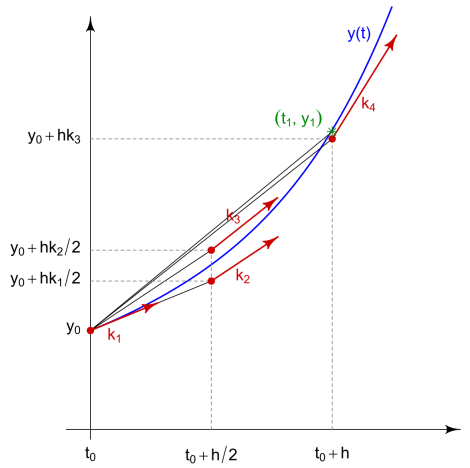


Figure 6: 4th-order Runge-Kutta numerical integration (Wikimedia, 2017)

raytracer uses an implementation of 4th-order Runge-Kutta, which provides a decent accuracy that is more than good enough for our purposes. The implementation takes inspiration from Press and Teukolsky (1992) in how they calculate an error for each iteration in order to update the step size h. However it is modified slightly due to the equation we wish to integrate. Runge-Kutta operates on a set of 1st-order differential equations, whereas we have a single 2nd-order differential equation. So the equation is translated to a pair of 1st-order differential equations so we can integrate it as follows.

Let $z = \frac{du}{d\varphi}$ such that $\frac{d^2u}{d\varphi^2} = \frac{dz}{d\varphi}$. Then:

$$\frac{dz}{d\varphi} = -u + \frac{3}{2}u^2$$

$$\frac{du}{d\varphi} = z$$

Then we can use these equations with 4th-order Runge-Kutta to find $u = \frac{1}{r}$ for a particular polar coordinate φ.

Let $f(\varphi, u, z) = \frac{du}{d\theta}$, $g(\varphi, u, z) = \frac{dz}{d\varphi}$ and h be some small number. Given some initial conditions $\varphi_0 = 0$, $u_0 = 0$, $z_0 = 1/b$, we can compute an iteration like so:

$$k_1 = hf(\varphi_i, u_i, z_i)$$

$$l_1 = hg(\varphi_i, u_i, z_i)$$

$$k_2 = hf(\varphi_i + \frac{1}{2}h, u_i + \frac{1}{2}k_1, z_i + \frac{1}{2}l_1)$$

$$l_2 = hg(\varphi_i + \frac{1}{2}h, u_i + \frac{1}{2}k_1, z_i + \frac{1}{2}l_1)$$

$$k_3 = hf(\varphi_i + \frac{1}{2}h, u_i + \frac{1}{2}k_2, z_i + \frac{1}{2}l_2)$$

$$l_3 = hg(\varphi_i + \frac{1}{2}h, u_i + \frac{1}{2}k_2, z_i + \frac{1}{2}l_2)$$

$$k_4 = hf(\varphi_i + h, u_i + k_3, z_i + l_3)$$

$$l_4 = hg(\varphi_i + h, u_i + k_3, z_i + l_3)$$

$$k = \frac{1}{6} \times (k_1 + 2k_2 + 2k_3 + k_4)$$

$$l = \frac{1}{6} \times (l_1 + 2l_2 + 2l_3 + l_4)$$

See Figure 6 for how these variables relate to the functions. Then we update φ_i, u_i and z_i as follows:

$$\varphi_{i+1} = \varphi_i + h$$

$$u_{i+1} = u_i + k$$

$$z_{i+1} = z_i + l$$

With this step, we have completed an iteration of Runge-Kutta. Note that we must incrementally decrease h as we approach the target φ so that our result for u becomes more accurate. This is the basis for how we numerically integrate the Binet equation to compute the deflection of light.

2 Implementation

The easy part, and simultaneously the bulk of the implementation, was to write a very basic raytracer. The program is written in C++ and uses OpenGL and GLSL shaders to render to the screen.

2.1 Source Tree

```
.
|-- README
|-- CMakeLists.txt
|-- include
    |-- bholetrace
        |-- keys.hpp
        |-- load_shaders.hpp
        |-- load_textures.hpp
|-- lib
    |-- CMakeLists.txt
    |-- ...
|-- src
    |-- load_shaders.cpp
    |-- load_textures.cpp
    |-- main.cpp
    |-- shaders
        |-- raycast.comp
        |-- render.frag
        |-- render.vert
|-- textures
    |-- nebula-0.png
    |-- nebula-1.png
    |-- nebula-2.png
    |-- nebula-3.png
    |-- nebula-4.png
    |-- nebula-5.png
    |-- grid-0.jpg
    |-- ...
|-- build
    |-- ...
```

2.2 Build Process

The program is built using CMake (find instructions to build in the `README` file) and builds two libraries from source, `glm` and `stb_image`, which are stored in the `lib` directory. These are used for vector mathematics and image loading respectively. Other than that the project depends on `glut`, `glew` and standard OpenGL bindings which must be available in the build environment. The program expects `textures` and `shaders` to be in the current directory when executing. The `build` directory is setup by CMake with symlinks to achieve this. These resources could, in future, be packed into the executable for a more portable application, but this has not been done due to time constraints.

2.3 Raytracing in OpenGL

The program loop and setup code is almost entirely written in `src/main.cpp` since it is fairly trivial. The setup code initialises attributes of the camera that observes the black-hole (position, rotation and FOV) then instructs OpenGL to render two triangles which span the entire screen. The raytracing is not done using RTX or any special technology (though a derivative work could use this for performance improvements when available), it simply uses an OpenGL compute shader (`src/shaders/raycast.comp`) and renders the result to a texture with dimensions which divide the window evenly, e.g. 1920/2 x 1080/2. Note that the application is hardcoded to render to a `1920x1080` window but this is easy to change in the `src/main.cpp` file. A single ray is traced for each pixel in the transfer texture. The transfer texture is responsible for storing the raytrace results (which can be rendered directly) so that OpenGL can render it to the screen. The fragment/vertex shaders (`src/shaders/render.verg`, `src/shaders/render.frag`) in the project do exactly this - they take the transfer texture, scale it up to the resolution of the window, then render it to the screen for the user to see. The result of this is that the number of rays can be reduced and the rendered texture can be scaled up to be the same apparent size on the user's screen. A ray traced by the program will always either fall into the black-hole or collide with the cube map. The cube map (sometimes referred to as the skymap, skybox or celestial sky equivalently throughout this

paper) is a box of textures that surround the scene, positioned at an infinite distance away. This, as well as a sphere intersection test, is what I used to initially test the basic raytracer. When the deflection is disabled, the program just renders a black sphere in front of the celestial sky. A derivative work could trivially extend the raytracing with further intersection tests to see what it would look like if an arbitrary object were close to the black-hole. While this would not be hard to do, due to time limitations the program only displays the distortion of the celestial sky.

2.4 User Interaction

The program loop handles camera movement, which by default orbits the black-hole at a fixed radial distance. In this mode, the user can use the up/down arrow keys to increase/decrease the orbital radius, seeing the distortion of light (observed by a distortion in the image of the celestial sky) change as they do so. They can also use WASD to offset rotation from looking directly at the black-hole, which is useful when the camera is positioned at small orbital radii (and the black-hole takes up most of the screen.) A change in the camera's attributes will immediately update the rendering of the black-hole since all rendering results are recomputed for every frame. For completeness, see a list of key bindings below.

- `WASD`: change rotation of the camera (offset from orbit centre.)
- `UP/DOWN` arrow keys: increase/decrease orbit radius.
- `Z`: enumerate over skyboxes (`nebula`, `starmap` and `grid`.) These are detailed further in Results.
- `R`: enumerate over orbit speeds (0x, 1x, 2x and 4x.)
- `L`: toggle on/off deflection of light. When deflection is disabled, the event horizon is rendered via sphere intersection.

2.5 Numerical Integration

Initially, I intended to numerically integrate the equation to find when r appears to be travelling straight. This approach was inconsistent due to the numerical instability of the formulated differential, and would require finding two pairs of values (φ, u) and (φ', u') in order to find the scattering angle $\Delta\varphi$. It seemed it would be much more effective to converge on a value instead, and somehow use that to determine the trajectory. Thankfully that is more than possible.

$$\Delta\varphi = 2|\varphi(r_0) - \varphi_\infty| - \pi$$

This equation comes straight from the Gravitation and Cosmology book by Weinberg (2008) and is much simpler than it looks. The deflection ($\Delta\varphi$) is twice the change from infinity (φ_∞) to the closest radial distance ($\varphi(r_0)$), minus π. Note that we subtract π here because if the ray were to travel straight the change in φ would simply be π. So the subtraction gives us the scattering angle $\Delta\varphi = \theta$ of the ray. Also, to avoid confusion, realise that the variables φ and $\Delta\phi$ that the book uses are distinct and refer to different things. Change in φ refers to the overall change in the polar angle as we follow the ray of light, whereas $\Delta\phi$ refers to the deflection of the ray from a straight line. Using this equation and by exploiting the symmetry of a black-hole's deflection, we only need to converge on a single pair of values $(\varphi, \frac{1}{r_0})$. OpenGL shaders being what they are, it is often impossible to debug errors in their code. So in order to be sure that the numerical integration was working properly I first implemented Runge-Kutta in Python. For

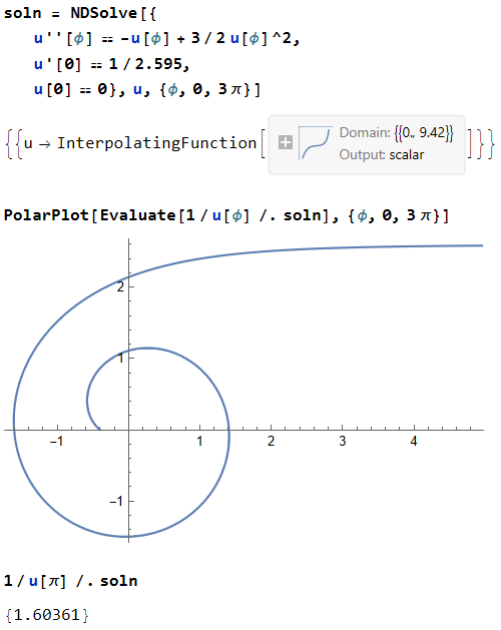


Figure 7: An example of the differential integrated in Mathematica.

reference, I used Mathematica to numerically solve the Binet equation and plot it on a graph. Then I could get an approximation for the value of u I should expect for a given φ. See Figure 7 for a case where the impact parameter (here set to $b = 2.595$) would cause the light ray to gradually fall into the black-hole. Recall that r is expressed in terms of the event horizon radius, so if the light ray falls below $r = 3/2$ (the photon sphere), we know it will never leave the black-hole. In this case we would render the relevant pixel black (since we know a light ray could never have originated from this direction.)
The numerical integrator for our program must handle two distinct cases:

1. When the light ray falls into the black-hole and we should render the pixel black; and,

2. When the light ray misses the black-hole and we must find the trajectory of the deflected ray.

Using the fact that it is sufficient to find r_0 (closest approach of the light ray), the first case is trivial. The closest approach of the ray has little meaning when it is falling into the black-hole, since the value of r will continue getting infinitely smaller. We can say that $r_0 = 0$ at the limit. But we have $u = \frac{1}{r}$, so similarly we would expect u to reach infinity at its limit. This means that if we can find some number r_{min} (or equivalently u_{max}) where the light ray will definitely fall into the black-hole, we can return early and report that the ray fell into the black-hole. This value is well defined: it is the radius of the photon sphere i.e. $r_{min} = 3/2$. So whenever u exceeds $2/3$ we know the light ray will never escape the black-hole. Due to the inherent imprecise nature of intermediate steps in numerical integration, it makes sense to be careful here about using this value exactly. The final numerical integrator for the raytracer uses an $r_{min} = 1$ to give a safe boundary for error.
For the other case, the integrator will take relatively large steps until it finds a point where $u_i > u_{i-1}$. When this happens, the step size is halved and negated (effectively changing the direction in which we move the polar coordinate ϕ.) This is done for a maximum number of steps or until $\Delta u = u_i - u_{i-1}$ is small enough that we have a sufficient accuracy (often much earlier than the maximum step count.) The source code for this Python program will be included in the project source tree for completeness, though the code is almost equivalent to the GLSL equivalent. Note that the output below denotes φ as a (for angle.)

```
$ python rk4_r0.py
Impact parameter (b) = 2.595
  0: 0.500   0.18581   0.34684
  1: 1.000   0.34029   0.26836
  2: 1.500   0.45423   0.18925
  3: 2.000   0.53229   0.12615
  4: 2.500   0.58346   0.08146
  5: 3.000   0.61626   0.05198
```

```
 6:  3.500   0.63725   0.03359
 7:  4.000   0.65111   0.02296
 8:  4.500   0.66112   0.01788
 9:  5.000   0.66972   0.01728
10:  5.500   0.67912   0.02112
11:  6.000   0.69176   0.03057
12:  6.500   0.71106   0.04846
13:  7.000   0.74252   0.08050
14:  7.500   0.79572   0.13821
15:  8.000   0.88883   0.24627
16:  8.500   1.05946   0.46415
17:  9.000   1.39651   0.96136
18:  9.500   2.15554   2.35740
19:  10.000   4.34424   8.05346
20:  10.500   15.16015   60.65896
done:
10.500   -1.000   60.659
```

The returned value for u is -1, so we know that this light ray falls into the black-hole.

```
$ python rk4_r0.py
Impact parameter (b) = 3
     ai       ui         zi
 0:  0.500   0.16059   0.29900
 1:  1.000   0.29242   0.22481
 2:  1.500   0.38411   0.14207
 3:  2.000   0.43530   0.06368
 4:  2.500   0.44857   -0.01027
 5:  3.000   0.42489   -0.08504
 6:  2.750   0.44139   -0.04718
 7:  2.500   0.44856   -0.01025
...
37:  2.430   0.44892   0.00001
38:  2.430   0.44892   -0.00001
39:  2.430   0.44892   -0.00000
done:
2.4301147461     0.4489167537
-0.000
```

So we can take the final $u = 0.44892$ and, if we wanted, compute $r_0 = \frac{1}{u}$. However all we really need is the refined value for $\varphi(r_0)$, which is computed to be 2.43011 for $b = 3$. Once it was confirmed that the integration worked to a sufficient standard, the GLSL equivalent was written and used within the basic raytracer to deflect each light ray. If the ray falls into the black-hole, the pixel is rendered black. Otherwise it takes on the colour of the intersection point in the skybox (celestial sky.) For the implementation of this logic, see `src/shaders/raycast.comp` in the source tree.

3 Results

One of the primary goals of the raytracer was for it to perform at real-time rates. A result which is not strictly physically accurate but suffices to teach the effect of curved space-time is acceptable. Further, if we must lower the resolution to achieve good performance (but it can be kept high enough to see the distortion clearly), this is also an acceptable limitation. All tests were evaluated on my laptop, but a dedicated viewing environment to display the program could potentially achieve a `10x` speed-up in rendering performance (given that my laptop is slow.) Therefore, if we are able to achieve good results at `480x270` but not higher resolutions (e.g. `1920x1080` for a factor increase in rays cast of `16x`), this would be sufficient performance for our purposes.

3.1 Performance

The frames per second (FPS) of the raytracer is very dependent on the distance from the black-hole. When the photon sphere takes up much of the viewer's screen, the majority of the rays can return early once they reach a critical radius ($r < r_{min}$.) So a naive approach to setting the testing orbital radius has the potential to bias the results (as FPS changes a lot during actual usage.) In order to achieve more consistent results, the camera was set to oscillate from $r = 2.5$ to $r = 37.5$ with a period of a few seconds during testing. This oscillation was initially based on a "tick"

counter that incremented at most 60 times per second. But since this is limited by the frames per second of the program, the oscillation became slower at lower frame rates, which could have also introduced skewed frame rates. The oscillation instead uses the time elapsed as reported by the operating system, which gives a consistent oscillation period regardless of FPS.
The values in the FPS column denote average FPS versus minimum FPS. T_{ray} is the time in microseconds for each ray to trace, estimated as follows:

$$T_{frame} = T(Nframes)/N$$

$$T_{ray} = T_{frame}/(R_{width} \times R_{height})$$

where N is large (tests were evaluated with $N = 1000$.)

Resolution	FPS	T_{ray}
480x270	74 / 67	0.1027μs
960x540	24 / 21	0.0797μs
1920x1080	6 / 5	0.0771μs

Table 1: Average FPS with deflection enabled

Note that the program is always rendered at my laptop's native resolution `1920x1080`, but the raytracing can be scaled down to improve performance. The program feels most smooth to use when scaled down at a ratio of 4 : 1 (`480x270`) and this is observed in Table 1. We would expect the values for T_{ray} to be consistent across each resolution, since the difficulty in computing the colour for a single ray does not change. However T_{ray} seems to decrease as resolution increases, which is not surprising since it is only an estimate based on time elapsed across $N = 1000$ frames. The overhead of program loop, communication time with the GPU, etc. become less statistically relevant as the rays per frame increase. This means our estimate is too unreliable to use, but I have included it to give a general idea of the time spent tracing rays. Initial attempts to compute T_{ray} were based on OpenGL's timer query API (`GL_TIME_ELAPSED`), but these didn't seem to properly time the compute shader's execution (raytrace time did not change when deflection was toggled, despite a clear difference in frame rate.) My assumption is that timer queries do not wait for `glDispatchCompute` calls to complete. This behaviour may be different on other machines, so a follow up analysis of the program's performance could try using timer queries for raytracing benchmarks (and it might work.) Timer queries would be more effective to measure T_{ray} since its results would not be subject to rendering overhead.

Resolution	FPS	T_{ray}
480x270	224 / 224	0.0344μs
960x540	133 / 133	0.0144μs
1920x1080	50 / 50	0.0097μs

Table 2: Average FPS with deflection disabled

For completeness (and a sanity check) see Table 2 for the tests when deflection is disabled. The values for FPS average and FPS minimum are equal in each case, which we can assume is because orbital radius does not affect performance (and the FPS is therefore much more consistent.) Note that our T_{ray} estimate is substantially lower, too, which shows as expected that the deflection is the limiting factor in program performance.

3.2 Appearance

Another primary goal of the raytracer was to achieve good quality, consistent results that are physically plausible. There are three custom-made skyboxes (cube maps) included in the project: `nebula`, `starmap` and `grid`. The first two are procedurally generated starmaps which display the distortion of the black-hole against a celestial sky. The last is

an alternating grid texture which makes the effects of the distortion very obvious.
Each starmap was rendered for $r = 50$, $r = 10$, $r = 5$ and $r = 2$ to give an idea of the distortion at different orbital radii. The renders are additionally provided at the following URL for convenience: https://imgur.com/a/1nj0E8M. A video demonstration that showcases the program's features and render results can be found here: https://youtu.be/aV8gXieIkeQ.

3.2.1 $r = 50$ (far distance)

See Figure 8. At this fairly large distance, the black-hole's distortion is clearly visible but does not take up much of the observer's image. It's at this point that Einstein rings, as in real life, are most visible. See the `starmap` render for an example, though there are better examples in the demonstration video. A single star can be seen to be stretched all the way around the black-hole's outer distortion. Note that this is only loosely physically accurate since lights would almost always appear as point-like sources in real life. This is discussed more in Evaluation.

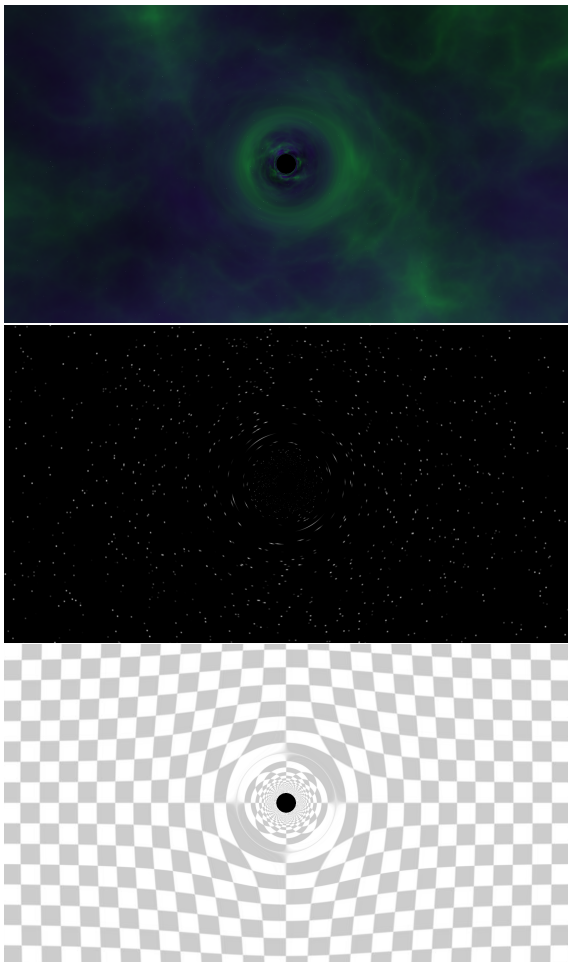


Figure 8: Render result for $r = 50$

3.2.2 $r = 10$ (medium distance)

See Figure 9. Distortion becomes significantly more prevalent as we approach the black-hole. There is a clear outer ring where the celestial sky is significantly stretched which seems to match existing black-hole simulations qualitatively.

3.2.3 $r = 5$ (close distance)

At a close distance the black-hole's distortion takes up the majority of the screen. In the `grid` render, the background seems to fold away from the viewer as if it were getting further away, when in reality it is just more significant distortion. The deflection seems to

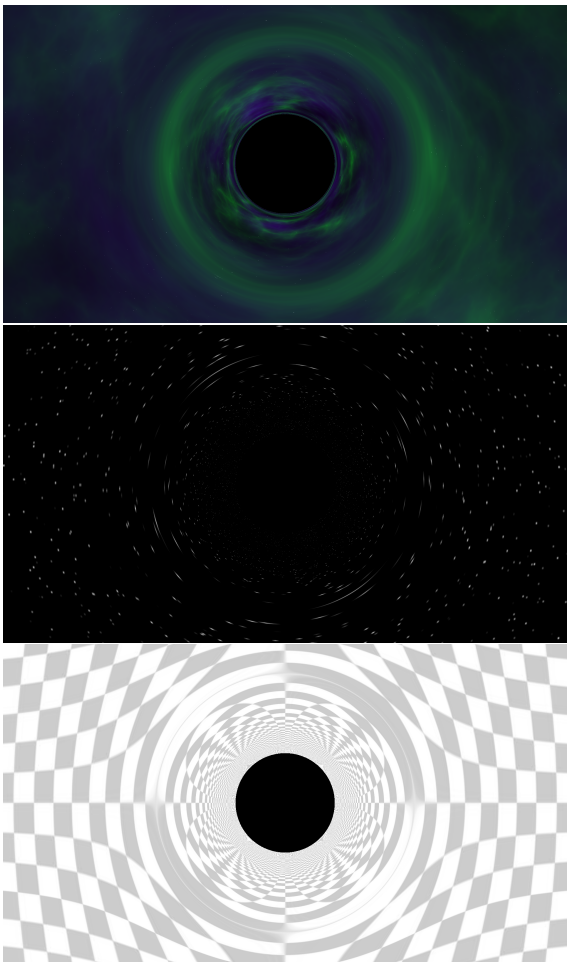


Figure 9: Render result for $r = 10$

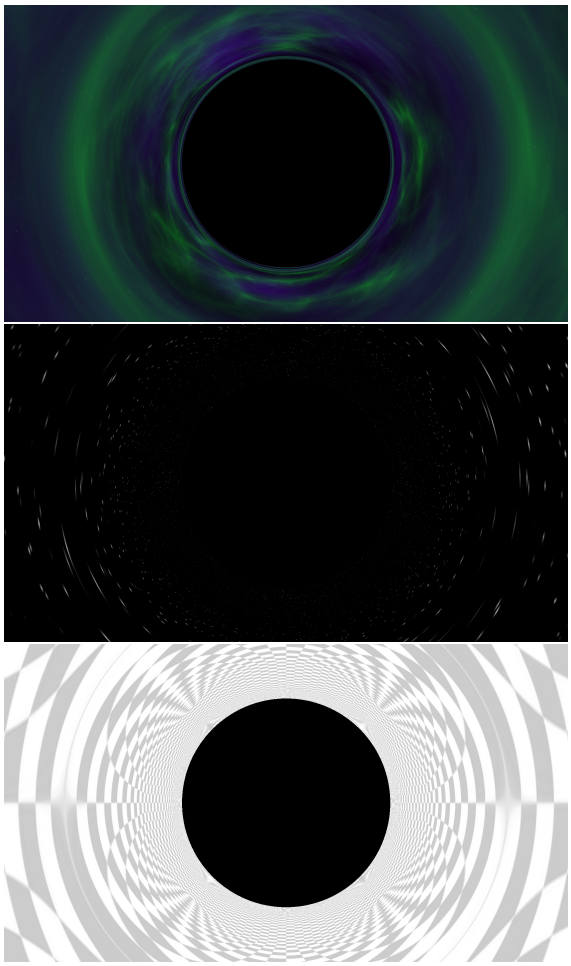


Figure 10: Render result for $r = 5$

reach a limit where the tiles are indistinguishable (the light ray is orbiting the black-hole many times over before reaching the grid.)

3.2.4 $r = 2$ (very close distance)

As we approach the photon sphere at $r = \frac{3}{2}$ (recall that the actual radius of this boundary is at $\frac{3}{2}r_s$, where r_s is the event horizon radius) the black-hole's apparent size takes up more than half of the screen. The renders are rotated to look at the distortion around its edge, which now appears as a folding "surface" in front of the viewer. The distortion at this distance is not physically accurate in any sense due to the assumptions of our equations, but the result appears mathematically consistent and gives an insight of what it would look like

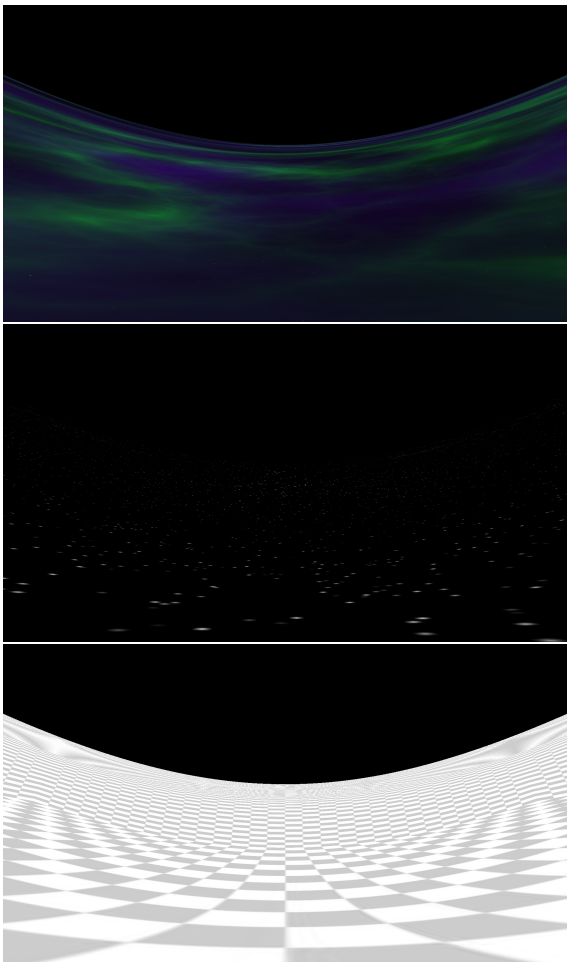


Figure 11: Render result for $r = 2$

to orbit at distances close to the event horizon.

4 Evaluation

4.1 Approximate Results

The raytracer makes many assumptions and approximations in order to achieve a simple implementation. While this has made the project plausible given the time constraints (and can be considered a bonus as a teaching tool), it's not clear how accurate our approximation is. In the Gravitation and Cosmology book by Weinberg (2008), there is discussion about the error in the derived scattering angle $\Delta\varphi$ due to assumptions about distances. So even assuming the light-ray comes from infinity, the equation would give an approximate result (and we would have to multiply our resultant $\Delta\varphi$ by a scalar constant to correct this error.) However, the raytracer's rays never orginate from an infinite distance, and depending on the camera's orbit radius may be very close to the black-hole. So it's possible that the raytracer's results are inaccurate enough that they are even misleading. But it is hard to tell, given a lack of directly comparable results, how accurate the renders are in their displayed distortion. Further, the raytracer focuses solely on the Schwarzschild black-hole, which probably does not exist in reality. It would be more physically relevant to render a Kerr black-hole (or another metric), though the equation would need to be substantially changed.

4.2 Quantitative Analysis of Visual Results

The results of existing work that can render the distortion of a celestial sky would be highly valuable in validating this project's distortion effects. Source code for works like these are often unreleased or intricate to setup, and render times can be much longer when there is an implementation focus on higher quality, but at present it is hard to properly verify the distortion effects of the raytracer. When setting some values such as the `MAX_ITERATIONS` or `target_precision` constants for numerical integration, I have had to guess about what is an acceptable trade off in accuracy for performance. I could be much more sure I've made a good decision for these values if I could see the difference through direct comparison of a known high quality result.

4.3 Stars Are Problematic

Existing papers (Müller and Weiskopf (2010), Riazuelo (2019), Verbraeck and Eisemann (2021)) on black-hole rendering were very upfront about this complexity when rendering

a celestial sky. Taking a simplistic approach and rendering stars the same way galaxies and dust are rendered leads to poor quality results. This is clear to see in Figure 11 and Figure 9 where stars are drastically warped. In real life, stars post-distortion appear as mostly point-like sources and do not form Einstein rings or skew to the extent that the program displays. I considered this an acceptable limitation but a special method for rendering stars would likely not affect performance in a significant way, especially if it were simple. The most noticeable artifact of this behaviour is flickering - stars will "blink" in and out of visibility rapidly when the camera is moving. The reason why this happens is explained well in the paper by Verbraeck and Eisemann (2021) through how pixels are mapped post-deflection in a general relativity raytracer. Black-hole distortion is not linear so square pixels do not come out square after distortion. Pixels are instead mapped to larger, non-rectangular regions on the celestial sky. When this is reversed to find the colour of a pixel, a simplistic approach may or may not hit the star closely enough to render the pixel white. So stars can be "missed" even though they should be very visible to the observer. If I had more time, this is one of the first issues with the raytracer I would attempt to resolve.

4.4 The Problem With Symmetry

Due to the way the deflection of the light ray is done mathematically, there are technically two black-holes in the scene: one in front and one behind the camera. This is relatively easy to fix but highlights very well the problem with ignoring assumptions e.g. that the light ray comes from an infinite distance. The method of using r_0 to compute the deflection works well but it is symmetrical: shooting a ray "away" from the black-hole is equivalent to shooting it the opposite way. The effect of this is two black-holes. It is hard to notice at first, but can be observed by placing the camera at a far distance when the `grid` skybox is active, then counting the walls as the camera rotates. This shows how the equations used could never really hope to display "falling into" the black-hole like the approach by Müller and Weiskopf (2010) can, because there is no concept of shooting a light ray "away" from the black-hole (and this doesn't make much sense to see what light reaches the camera anyway.) I have kept the issue in the program since it would only be hiding the problem to remove it, and it doesn't really affect the standard use of the program.

4.5 Goals Achieved

While the raytracer as-is can't produce full-HD output at a good enough performance for smooth motion, it is fair to say it achieved real-time results with a relatively simplistic approach and implementation. A dedicated viewing environment could boost the performance enough to provide good quality. The program could serve well as an introduction to the field of curved space-time geometry and the visual aspect of black-holes. The code base is small and well structured, permitting a user to extend it with relative ease. There is certainly room for improvement with the raytracer and this project's approach overall. With that said, the raytracer's render results are more than acceptable for our purposes and demonstrate the bending of light appropriately.

5 Notes on legal, social, ethical and professional aspects of this project

As a research software project and tool, there are no legal, social or professional

considerations. Ethically, the program takes a lot of inspiration from existing work and resources that are published in the field, but these are all attributed appropriately, along with any images used to demonstrate concepts in this report.

References

EHT-collaboration. Real-image of a black-hole, 2019. URL https://www.eso.org/public/news/eso2105/. [Online; accessed April 26, 2022].

Matthew Francis. Galileo's pendulum: A science blog by matthew francis, 2011. URL https://galileospendulum.org/2011/09/02/some-further-notes-on-black-holes/. [Online; accessed April 26, 2022].

J.-P. Luminet. Image of a spherical black hole with thin accretion disk. *Astronomy and Astrophysics*, 75:228–235, 04 1979.

Thomas Müller and Daniel Weiskopf. Distortion of the stellar sky by a schwarzschild black hole. *American Journal of Physics*, 78(2):204–214, 2010. doi: 10.1119/1.3258282. URL https://doi.org/10.1119/1.3258282.

William H. Press and Saul A. Teukolsky. Adaptive stepsize runge-kutta integration. *Computers in Physics*, 6(2):188–191, 1992. doi: 10.1063/1.4823060. URL https://aip.scitation.org/doi/abs/10.1063/1.4823060.

Alain 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(02):1950042, jan 2019. doi: 10.1142/s0218271819500421. URL https://doi.org/10.1142%2Fs0218271819500421.

Annemieke Verbraeck and Elmar Eisemann. Interactive black-hole visualization. *IEEE Trans. Vis. Comput. Graph.*, 27(2):796–805, February 2021.

Steven Weinberg. *5 Deflection of Light by the Sun*. Wiley, 2008.

Commons Wikimedia. Impact parameter, 2007. URL https://en.wikipedia.org/wiki/Impact_parameter#/media/File:Impctprmtr.png. [Online; accessed April 26, 2022].

Commons Wikimedia. Runge-kutta methods, 2017. URL https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods#/media/File:Runge-Kutta_slopes.svg. [Online; accessed April 26, 2022].

Commons Wikimedia. Ray tracing (graphics), 2019. URL https://en.wikipedia.org/wiki/Ray_tracing_(graphics)#/media/File:RaysViewportSchema.png. [Online; accessed April 26, 2022].

Yuen Yiu. All black holes should sport light rings. 2020. URL https://www.insidescience.org/news/all-black-holes-should-sport-light-rings. [Online; accessed April 26, 2022].