# Relativistic Ray Tracing in Unity

Brandon Li
*Department of Physics, Cornell University*
(Dated: October 24, 2023)

A fully interactive real-time ray tracing simulation of spacetime manifolds is developed with the ability to render arbitrary solutions of Einstein's equations. The Unity engine is used to create an accessible user interface and combine it with high performance graphics code. The work is applied to produce images of several spacetimes, including the Schwarzschild and Kerr black holes, and the Ellis wormhole. The technical and performance considerations of this computer program are mentioned, including various techniques to improve performance and visual fidelity. Finally, the educational implications of relativistic ray tracers are discussed.

## I. INTRODUCTION

General relativity is one of the most successful physical theories to have ever existed in the history of science. First formulated by Albert Einstein 1916, it describes the gravitational interaction between massive objects and shows how the trajectories of freely moving bodies arise from the curvature of space and time. In this theory, the universe is described as a four-dimensional curved manifold referred to as a spacetime. The idea is that a spacetime is a geometric space consisting of a bunch of points, that when viewed close up, resembles a region of flat Minkowski space. In Minkowski space, the distance between two events at points $(t, x, y, z)$ and $(t + \Delta t, x + \Delta x, y + \Delta y, z + \Delta z)$ is given by $\Delta s^2 = -\Delta t^2 + \Delta x^2 + \Delta y^2 + \Delta z^2$ in natural units. This quantity, called the spacetime interval, is special because its value is identical in all reference frames. We may think of the spacetime interval as a function that takes a vector and returns a number, in which case we write it as

$$g = -dt^2 + dx^2 + dy^2 + dz^2 \tag{1}$$

where $g = g(u, v)$ is called the metric, and in fact takes two vectors $u$ and $v$ as input. It is analogous to the Euclidean dot product as it is symmetric and bilinear and gives geometrical information about the vectors. The spacetime interval of the vector $u$ is obtained by plugging $u$ into both arguments, meaning $\Delta s^2 = g(u, u)$.

In general relativity, we again have a metric (which acts like a yardstick and measures local lengths and angles), but now the metric may vary from point to point in spacetime. This is another way of saying that spacetime may be curved. What Einstein found was that the curvature at any point in spacetime is directly proportional to the amount of energy there. In fact, there are many different kinds of curvature, but in this case, it is the Einstein tensor $G_{\mu\nu}$ that couples to matter and energy. The Einstein equation reads

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} = 8\pi T_{\mu\nu} \tag{2}$$

where $R_{\mu\nu}$ and $R$ are curvature tensors that depend on $g$ whereas $T_{\mu\nu}$ encodes information about the energy density and flux. Solving Einstein's equation gives us a range of universes with unique shapes and distributions of energy. A very important example arises when there is no matter content and $T_{\mu\nu} = 0$. In this case, if we impose spherical symmetry and solve for the metric, we find a family of solutions parameterized by a single variable $M$:

$$g = -\left(1 - \frac{2M}{r}\right)dt^2 + \frac{dr^2}{\left(1 - \frac{2M}{r}\right)} + r^2 d\theta^2 + r^2 \sin^2\theta d\phi^2. \tag{3}$$

This is likely the most famous solution of Einstein's equation. It is a non-rotating, uncharged black hole with mass $M$. First written down by Karl Schwarzschild immediately after Einstein published his theory of gravitation, this spacetime contains many interesting features that demonstrate non-trivial effects of general relativity. Because of this, a great deal of attention is usually placed on the study of Schwarzschild black holes in relativity courses. We will come back to this solution later on.

Due to the abstract nature of differential geometry, general relativity can be a difficult subject to understand. Mastering it requires having both physical intuition and mathematical proficiency. One way to develop intuition for any kind of physical phenomenon is to visualize it, and this is what the subject of this report is about. The goal of the project was to develop an interactive simulation of a black hole (and other spacetimes) that the user can move around in. In total, there were four objectives that we tried to achieve. First, the simulation should be as physically accurate as possible. To determine what an observer sees, the code should compute trajectories of individual rays of light. Secondly, the user should be able to interact with the simulation and move around freely. There should also be an graphical user interface that is simple and easy to use. Next, the graphics calculations should run in real time - good performance is important for maximum user interactivity. Finally, the resulting simulation should demonstrate aspects of relativity that would be difficult to convey otherwise. In other words, the product should be informative or educative in some way.

To compute what the observer sees, we will use a technique called ray tracing, in which photon trajectories are sent out from the observer's location and propagated backward in time until they hit something. The point on

the screen corresponding to the initial photon direction is then colored based on the color of the object hit 1. This works because photon trajectories are time-reversal symmetric, meaning the same trajectory will describe a photon that emitted by the object and travels forward in time until it hits the observer's eye.

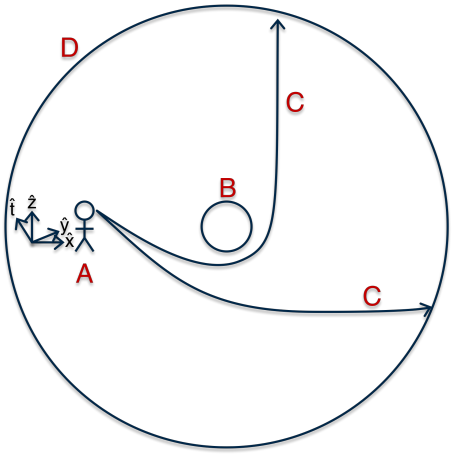


FIG. 1. Schematic depiction of ray tracing. A: Observer with observer-centric coordinate system. B: Massive object, eg. black hole. C: Photon trajectories emanating from observer. D: Background image. The background is spherical image with a large radius, meaning effectively it is infinitely far away.

To compute these trajectories, we use the geodesic equation, which describes the paths taken by any kind of small test object, including photons. Let $x(\lambda)$ be the position of a test object parameterized by the variable $\lambda$ and let $u(\lambda) = \frac{dx}{d\lambda}$ be the coordinate velocity. Then, the path taken satisfies

$$\frac{du^\alpha}{d\lambda} = -\Gamma^\alpha{}_{\beta\gamma} u^\beta u^\gamma. \tag{4}$$

Here $\Gamma^\alpha{}_{\beta\gamma}$ denote the Christoffel symbols, which encode how the coordinate system varies from place to place taking into account curvature, and they are computed from the metric. We see that the geodesic equation along with the equation $u = \frac{dx}{d\lambda}$ form a second order system of ODEs, which can be solved through Euler integration - the simplest method of solving ODE systems. This gives a simple way of computing the trajectories of rays of light as well as any other objects we like.

## II. IMPLEMENTATION

Before coding begins, it is necessary to determine the large scale details of how the project is structured. This involves choosing a development environment and programming language. Since performance was important, we chose to have the ray tracing code run on the graphics processing unit (GPU), leaving the rest of the code for the CPU. This arrangement is the most efficient since GPUs are optimized to perform many identical computations in parallel, exactly matching the demands of ray tracing. The CPU by contrast is much more flexible in the kinds of computations performed, but is is less parallelizable.

Integrating these two kinds of computing is tricky to do from scratch and requires a lot of work. Fortunately, a lot of this work has been done already and there are many off-the-shelf solutions for making interactive, user-interface (UI) based software that can run high performance code on both the CPU and GPU. For this project, we picked the Unity engine. It was chosen because of its ease of use, flexibility, and free cost. The engine is written in the C# language, and code for the GPU is written in HLSL shader language.

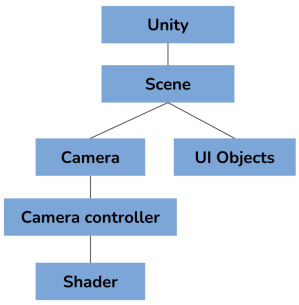


FIG. 2. Hierarchy structure of the Unity project.

The relativstic ray tracing project in Unity consists of a single scene in which the camera and all the UI elements are placed (Fig. 2). The UI contains all the buttons and the settings the user can adjust. The camera, on the other hand, is responsible for rendering a picture to the screen. Attached to the camera is a custom C# script, which we call `CameraController`, that contains a majority of the custom logic used to handle user input, move the camera around, send data to the GPU, and calculate the observer's trajectory. Finally, the graphics shader `GeodesicShader` is attached to the `CameraController` script and is used for ray tracing.

On each frame the `Update` method in `CameraController` is called. In this function, the keyboard is first checked to see if any keys are pressed. If so, the corresponding actions are performed. Next, the position of the observer is updated using the geodesic equation. Finally, the updated information is sent to the shader. Now, a tricky issue that needs to be dealt with is the manner in which the observer's orientation is encoded. Typically, in 3D flat space, the orientation would be stored as a triplet of numbers representing pitch, roll, and yaw. Those would then be turned into a rotation matrix in SO(3), and that gives the orientation. Here this is not possible since a points on a curved manifold cannot be arbitrary related to each other via a linear transformation. Instead, we will bring out the concept of a local lorentz frame, which is a set of vectors $\{\hat{x}, \hat{y}, \hat{z}, \hat{t}\}$ at $x$, the observer's location, such

that $g(\hat{x},\hat{x}) = g(\hat{y},\hat{y}) = g(\hat{z},\hat{z}) = -g(\hat{t},\hat{t}) = 1$ and $g(\hat{x},\hat{y}) = g(\hat{x},\hat{z}) = \cdots = 0$. Written in the coordinates formed from these vectors, the metric $g$ will look like the metric of flat Minkowksi space. We may then regard these vectors as the basis vectors for a lorentzian observer in their own stationary frame of reference. We will arbitrarily declare $\hat{x},\hat{y},\hat{z}$ correspond to the forward, left, and upward axes centered on the observer's body, and $\hat{t}$ is their timelike velocity vector. Then, when the observer moves around, these four vectors are parallel transported along the observer's trajectory. Finally, since parallel transport preserves the inner product, the lorentz frame will always remain orthonormal. Now, we pack these four vectors into the matrix $R = \begin{bmatrix}\hat{t} & \hat{x} & \hat{y} & \hat{z}\end{bmatrix}$. Rotations and boosts are performed by multiplying $R$ on the right by a lorentz transformation: $R \to R\Lambda$. Computing the initial velocity for rays of light is done by specifying a unit spatial direction $w = (w_x, w_y, w_z)$, turning it into the four vector $\vec{\mathbf{w}} = (1, w)$ that represents the photon 4-velocity in the observer frame, and then converting to the coordinate frame via the multiplication $R\vec{\mathbf{w}}$. With all this math, it is finally possible to allow the user to move and look around.

## III. RESULTS

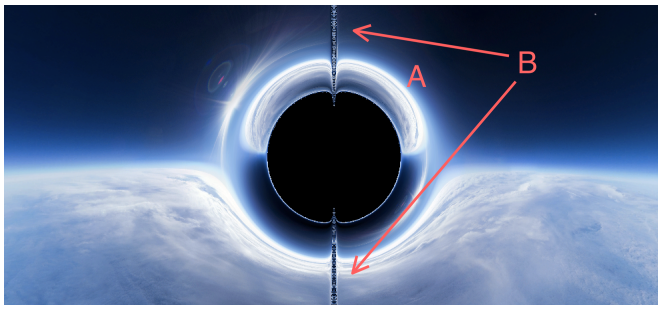


FIG. 3. Render of Schwarzschild black hole. A: Effect of gravitational lensing. B: Coordinate artifacts.

The first scenario implemented into the code was the Schwarzschild black hole in standard spherical coordinates. There are some disadvantages of this specific coordinate system, but the big upside is in its simplicity which allows for its rapid implementation and testing. After all the math and coding preparation, we are finally able to render a picture of the black hole (Fig. 3). Surprisingly, the graphics can run in real time without any performance issues. Movement and camera panning is completely smooth, at least when tested on the author's laptop. In the picture, we can clearly see the effect of gravitational lensing. One place this is evident is near the top of the black hole, where we can see the whites of the clouds as those photons were initially travelling upward but were bent forward by the spacetime curvature (Fig. 3, A). The black hole itself is black because photons that originate within the event horizon will never reach the observer. There is notably one issue that arises due to the presence of coordinate singularities. At $\theta = 0, \pi$, the metric is not invertible meaning there is a singularity there. This coincides with the visual artifacts in the image (Fig. 3, B). This can be explained by first observing that at values of $r$, the Schwarzschild metric approaches the Minkowkski metric in spherical coordinates, $-dt^2 + dr^2 + r^2 d\theta^2 + r^2 \sin^2\theta d\phi^2$. From this we know that the singularity at $\theta = 0, \pi$ is not real and is instead caused by the spherical nature of the coordinate system. From a calculational standpoint, the visual disturbances are caused by the $\Gamma^{\phi}_{\theta\phi}$ Christoffel symbol blowing up near the problematic values of $\theta$. Actually, there is another unphysical singularity at the event horizon radius $r = 2M$. This causes problems if the user wishes to go inside the event horizon.

There are two ways of resolving these issues. One is to make the integration steps finer near the poles, and more generally have the step size be smaller near any kind of singular region. As we will see, this is quite successful, but it comes with a performance cost (Table I), and only remedies the $\theta$ singularity. The other solution is to use a different, singularity-free coordinate system. In the case of the Schwarzschild black hole, people have invented many coordinate systems that all describe the same spacetime. Out of all these coordinate systems, there is one that eliminates the singularities at both the poles and the event horizon. Consider the following form of the Schwarzschild metric deduced by Gullstrand and Painlevé [1]:

$$g = -\left(1 - \frac{2M}{r}\right) dt^2 + 2\sqrt{\frac{2M}{r}}\, dt\, dr + dr^2 + r^2 d\theta^2 + r^2 \sin^2\theta d\phi^2. \tag{5}$$

If we compute the determinant of the metric, we find that it stays non-zero for all values of $r$, so there is no event horizon singularity. Furthermore, the second line of the equation is just flat 3D space and can be replaced with $dx^2 + dy^2 + dz^2$. We still have the $dt\, dr$ term to worry about, but we can use the relation $dr = \frac{x}{r}dx + \frac{y}{r}dy + \frac{z}{r}dz$. Using this, we can change to $xyz$ coordinates and after doing so we find all the coordinate singularities have disappeared.

After implementing Gullstrand-Painlevé (GP) coordinates, we now find a perfectly uniform sphere (Fig. 4) upon running the code. Note that performance of these coordinates is comparable to the standard coordinate system (Table I). We can also go inside the event horizon, unlike before. Upon crossing the event horizon, it seems as if nothing has changed. The world outside the black hole is still visible, and this is expected since information can still flow into the event horizon - it just cannot exit it. After a short period of time, we hit the singularity at $r = 0$, then the simulation stops. If the user tries to escape from the event horizon by boosting (accelerating) away, they will find that it is impossible to do so.

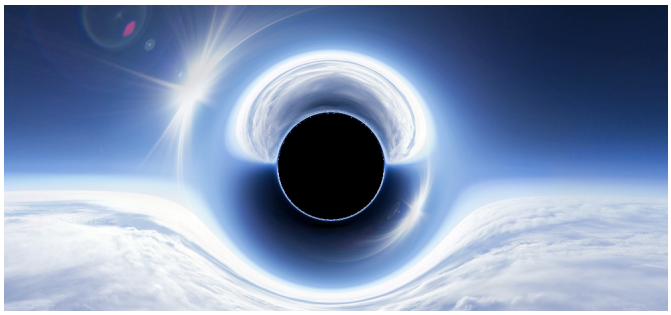

FIG. 4. Schwarzschild black hole in Gullstrand-Painlevé coordinates.

Whereas this may otherwise have been just a mathematical fact, with this tool, the user can clearly see why that is the case.

TABLE I. Performance of different coordinate systems and computational methods (lower is better).

| Coordinate system | Adaptive step size | Frame time (ms) |
|---|---|---|
| Schwarzschild | No | 8 |
| Schwarzschild | Yes | 13 |
| Gullstrand-Painlevé | No | 6 |
| Gullstrand-Painlevé | Yes | 22 |

After the Schwarzschild black hole was completed, two more spacetimes were added. The first was the Kerr metric, which describes a rotating black hole. It contains two parameters $M$ and $a$, with $M$ being the mass and $a$ denoting the angular momentum per unit mass. We plot the black hole for two different values of $a$, one physical and one unphysical (Fig. 5, 6). The unphysical solution features a naked singularity, one that is not hidden within an event horizon.

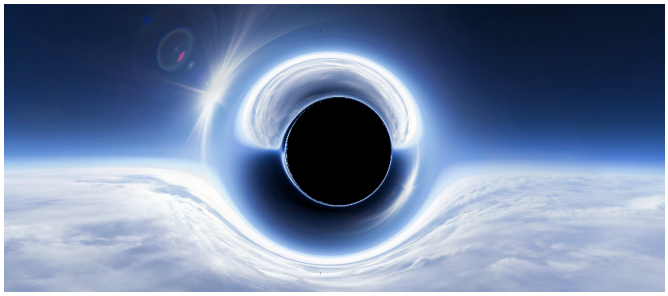

FIG. 5. Kerr (rotating) black hole for $a/M = 0.9$.

The other spacetime is the Ellis wormhole, which connects two distinct regions of spacetime that each resemble a full universe. It is described by the metric

$$g = dt^2 - dr^2 - (r^2 + l^2)(d\theta^2 + \sin^2\theta d\phi^2). \qquad (6)$$

Here the parameter $l$ controls the diameter of the wormhole. This wormhole is traversable, which means the user

FIG. 6. Kerr (rotating) black hole for $a/M = 10$. Solution features a naked singularity.

is able to freely move from one side to the other (Fig. 7). We use two different background images for the two sides, and looking into the wormhole we can get a glimpse into the other world.

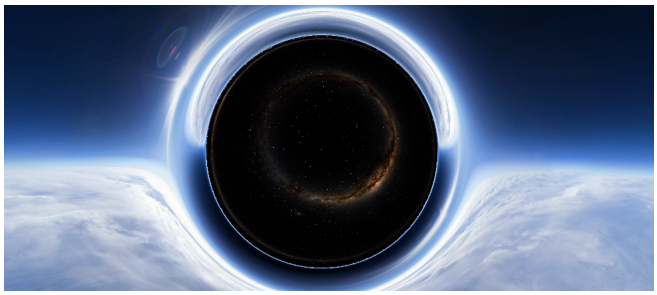

FIG. 7. Ellis wormhole connecting two universes.

## IV. DISCUSSION

Because of the decision to use Unity and to run ray tracing calculations on the GPU, we were able to build a fully interactive simulation that runs in real time and combine it with an intuitive user interface. Furthermore, the rendered image is physically accurate (for the most part) and corresponds to what a real observer would see if they went near a black hole. Complete physical accuracy actually requires one phenomenon that has not been implemented, namely relativistic doppler shift/brightening. This is where the the color and brightness of an object changes depending on its velocity. This is the only caveat to physical accuracy - otherwise everything is fine. Redshift for arbitrary moving objects is not easy to calculate, that is the reason it hasn't been implemented yet.

Another issue is that it is quite difficult to navigate around. The biggest reason for this is that there are no spatial landmarks to for a person to orient themselves around. Currently, the only point of reference is the background image, but as it is infinitely far away, its utility is limited. Ideally, we could place a few objects around the scene such as orbiting planets. The difficulty is we would need to modify the ray tracing algorithm to detect

collisions with objects. This requires storing the entire worldline of each face of the object and sending this information to the shader, which is very complicated, not to mention the additional performance impact. Development of a relativistic ray tracing algorithm may be the subject of a future work. Adding moving elements or clocks would also have the benefit of showing the effects of time dilation. Lastly, putting this simulation into virtual reality might make for an interesting experience.

Currently, due to some of the reasons mentioned above as well as the general lack of features, our ray tracer is not very educational. One idea to increase its educational value is to turn it into a game. In this game, the user would play as a scientist tasked with the mission of studying a black hole. They would have to complete several "quests" and accomplish different tasks related to the black hole such as getting into orbit, sending a probe into the event horizon, or taking time dilation measurements, for example. The user would gain points based on how well the tasks were completed, and in the process they would learn about various aspects of general relativity.

## V. CONCLUSION

We have found that it is possible to render a black hole on a computer in real time. The Unity engine is a good tool for applications that require both speed and user-friendliness. With the code we developed, we can generate beautiful pictures of black holes and other spacetimes. There is still a lot of work to be done to increase its pedagogical effectiveness, but we believe this software has the potential to be a valuable educational tool and help students understand the less intuitive aspects of relativity.

[1] E. Poisson, *A Relativist's Toolkit: The Mathematics of Black-Hole Mechanics* (Cambridge University Press, 2004).