Simulating the Gravitational Lens Effect of a Schwarzschild Black Hole

Computer Programming: Student-Chosen Project

Final Report

Kim Yeonjun[1]

[1]Dept. of Computer Science and Engineering, Seoul National University.

kyjun0803@snu.ac.kr

2025-06-12

Abstract

My student-chosen project aims to simulate the graphical effects of the gravitational lens effect of a black hole, which is caused by a black hole curving the spacetime. I made a simulation software based on the theory of General Relativity, which calculates the image of background objects which are distorted by a Schwarzschild black hole.
I implemented the rendering algorithm based on ray tracing. I made multiple classes and utilized the OOP features of the C++ language to realize it. Also, I used the Qt framework to make a graphical user interface (GUI) where the user can easily configure the camera and background parameters. This report covers the overall code structure, the rendering algorithm, the development process, program manual and demonstrations.

Contents

1. Introduction

Massive objects make curvature in spacetime and bend the path of light that comes from another distant object. It results in a distorted and stretched image of the object. This phenomenon is known as 'gravitational lens' and is explained by the theory of General Relativity.

This project aims to simulate the graphical effects of a black hole's gravitational lens with the C++ language and its OOP (object-oriented programming) features. More specifically, it will simulate the image of the input background, which is blocked by a Schwarzschild black hole. A Schwarzschild black hole refers to a black hole which is not rotating and has no electrical charge.

2. Program Requirements

The goal program is a GUI (graphical user interface) program. I utilized the Qt framework to implement the GUI. The program takes the camera's distance from the black hole, field of view (FOV), and information about the background as input. The values of those inputs are given by the user through slider widgets. When the user clicks the 'render' button, The rendered simulation result is displayed at the window.

This project is largely inspired by a GitHub repo [1] which implemented the Schwarzschild black hole simulation algorithm in Python. My program provides a more intuitive control panel interface, using buttons and sliders in Qt, which becomes my original feature.

3. Development Plan

3.1 Project development plan in 3 stages

I have divided the entire process of project development into the following three stages.

[Stage 1] Implement the vector and ray classes, and the rendering algorithm in C++.

[Stage 2] Receive input via CUI. output rendering result (in ppm format).

[Stage 3] Implement input and output in GUI with Qt framework.

To avoid complications at early stages of development, I first developed the essential classes and algorithms in CUI (console user interface). Before Stage 3, I tested the rendering output as ASCII image and ppm format (which is a text-based image format, where the value of each pixel is represented as three integers in one row). Then, I started developing GUI with Qt framework and just implanted the complete rendering algorithm.

3.2 Light path computation & rendering algorithm

The rendering function is based on the ray tracing algorithm. Ray tracing is an image rendering method where the computer traces light rays emitted from an imaginary eye and determines each image's pixel value by computing the object hit by the light ray.

For simulation of the gravitational lens effect, my program computes the path for a bunch of light rays. (Specifically, nm light rays if the output is an $n * m$ image) Each light ray's path is computed by numerically solving Eq.1 (the geodesic equation) shown below [3]:

$$\frac{d^2x^\mu}{d\lambda^2} + \Gamma^\mu_{\rho\sigma}\frac{dx^\rho}{d\lambda}\frac{dx^\sigma}{d\lambda} = 0 \tag{1}$$

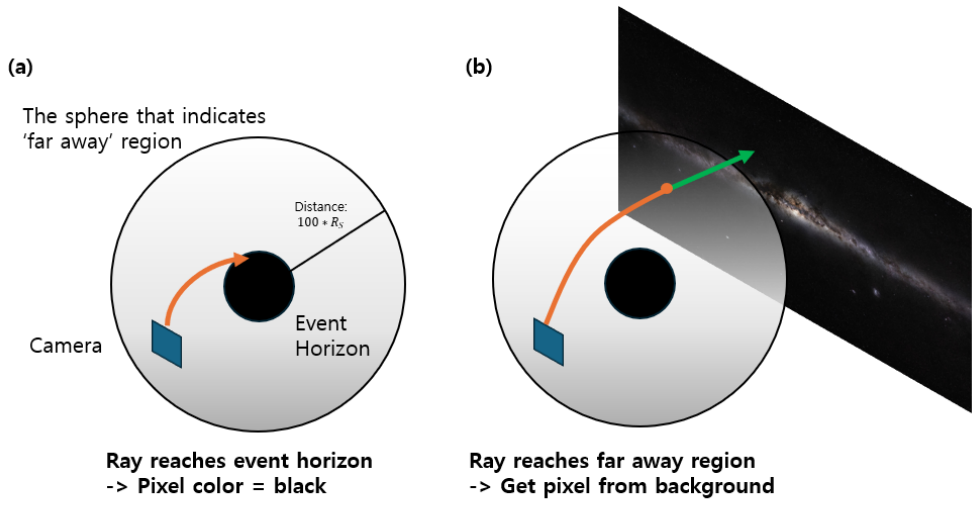
Where $\Gamma^\mu_{\rho\sigma}$ (the Christoffel symbol) is defined as

$$\Gamma^\mu_{\rho\sigma} = \frac{1}{2}g^{\mu v}\left(\partial_\rho g_{\sigma v} + \partial_\sigma g_{v\rho} - \partial_v g_{\rho\sigma}\right) \tag{2}$$

g here denotes the metric tensor, which is a mathematical object which contains the information about curved spacetime in General Relativity. In my program's case, g becomes the Schwarzschild metric, since it is simulating light rays near a Schwarzschild black hole.

For more descriptions of the physics background of this project, see Appendix 1.

The calculated light path is used to determine a single pixel to be displayed at the screen. If the light is calculated to reach the event horizon, the pixel color will be black. If the light is calculated to reach a far region from the black hole, the pixel color will be derived from the background image. [Fig. 1] illustrates how each pixel value is calculated through this ray tracing procedure.

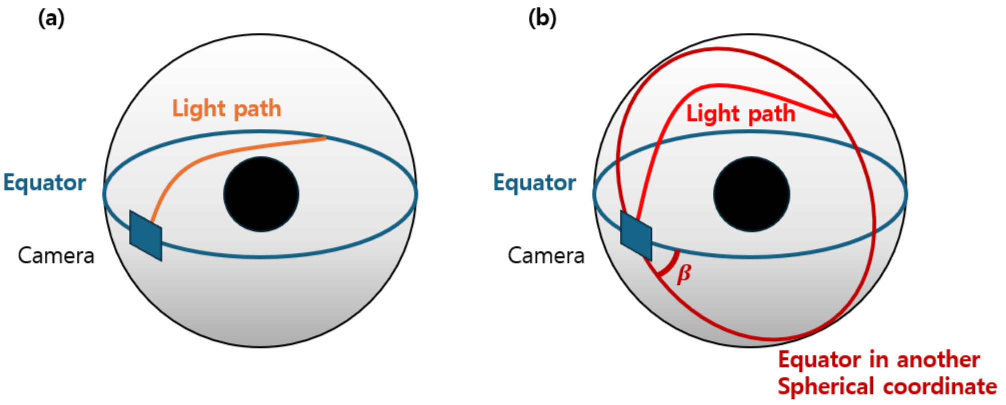

(a)
The sphere that indicates 'far away' region
Distance: $100 * R_S$
Camera
Event Horizon
Ray reaches event horizon -> Pixel color = black
(b)
Ray reaches far away region -> Get pixel from background

[Fig. 1] How each pixel of the image is calculated by ray tracing. The orange curve represents the calculated light path by numerically solving Eq. 1. Once the orange curve meets the 'far away' sphere, it is extended (the green arrow) to determine the corresponding pixel of the background image.

A bunch of pixel values calculated with many different initial conditions (initial directions) constitute the rendered image. However, naively computing with Eq.1 is tough and costly, since it is a 4-dimensional tensor differential equation. We can do better by exploiting the symmetry of the problem. The upcoming section describes a better algorithm.

3.3 Exploiting symmetry of the problem

The reference project [1] introduced a method to reduce the complexity of the equation to solve. This exploits symmetry of the situation. [Fig. 2] shows how this symmetry reduces the problem's dimensionality.



[Fig. 2] (a) Light path when the initial direction of the light is along the equatorial plane. The light path does not deviate from the equatorial plane. (b) Generally, any light path in this setting can be viewed as being on the equatorial plane in some spherical coordinate.

The symmetry implies that the light rays are always moving on the equatorial plane in some spherical coordinate: the plane which forms the angle β with the equator of the ordinary spherical coordinate. Here β can be derived from the initial direction of the light ray. Now we know that this is practically a two-dimensional problem.

So, the improved ray tracing algorithm is as follows:

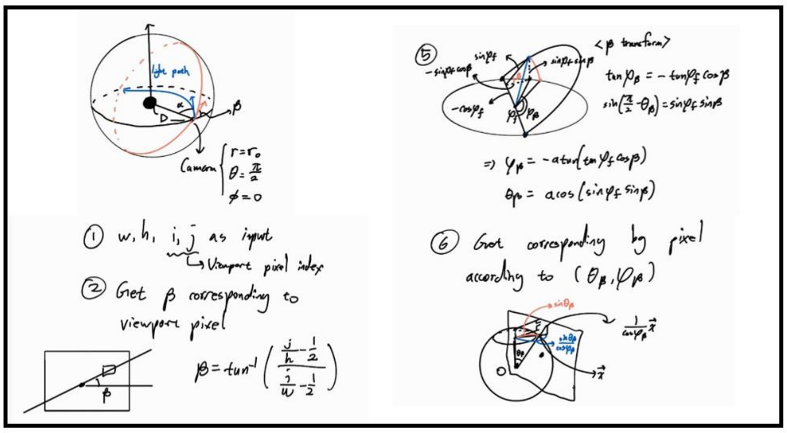
1. **Given the initial direction of a light ray, compute $\boldsymbol{\beta}$.**
2. **Compute the path of the ray on a 2-dimensional equatorial plane.**
3. **Rotate the plane by $\boldsymbol{\beta}$ to get the light's correct path.**
4. **Get the pixel value for the ray corresponding to its calculated path.**

The equation we need to numerically solve reduces to an ordinary differential equation, Eq.3. (The derivation is briefly explained in Appendix 2.)

$$\frac{d^2u}{d\phi^2} = \frac{3R_S}{2}u^2 - u \tag{3}$$

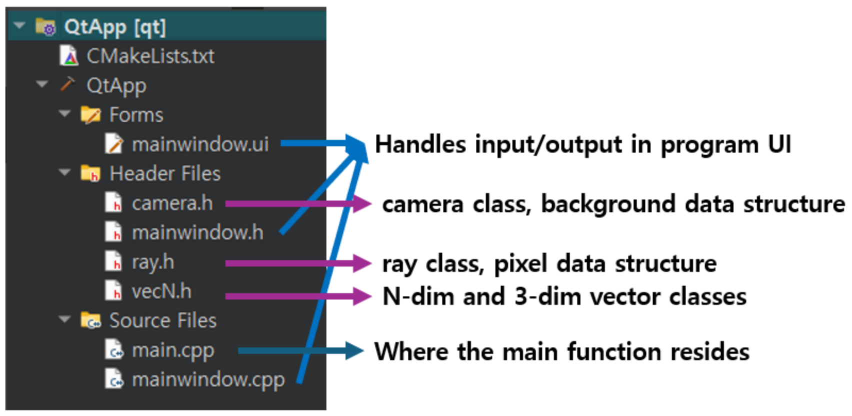
Where $u = \frac{1}{r}$ and R_S is the Schwarzschild radius of the black hole.

The formula for β are as shown in [Fig. 3].


light path
α
β
D
Camera $\begin{cases} r=r_0 \\ \theta=\frac{\pi}{2} \\ \phi=0 \end{cases}$
① w, h, i, j as input
Viewport pixel index
② Get β corresponding to viewport pixel
β
$\beta = \tan^{-1}\left(\frac{\frac{j}{h}-\frac{1}{2}}{\frac{i}{w}-\frac{1}{2}}\right)$
⑤
<β transform>
$\sin\varphi_f$
$-\sin\varphi_f\cos\beta$
$\sin\varphi_f\sin\beta$
$-\cos\varphi_f$
φ_f
φ_β
$\tan\varphi_\beta = -\tan\varphi_f\cos\beta$
$\sin(\frac{\pi}{2}-\theta_\beta) = \sin\varphi_f\sin\beta$
$\Rightarrow \varphi_\beta = -\mathrm{atan}(\tan\varphi_f\cos\beta)$
$\theta_\beta = \mathrm{acos}(\sin\varphi_f\sin\beta)$
⑥ Get corresponding bg pixel according to $(\theta_\beta, \varphi_\beta)$
$\sin\theta_\beta$
$\frac{1}{\cos\varphi_\beta}\vec{x}$
$\frac{\sin\theta_\beta}{\cos\varphi_\beta}$
θ_β
O
$\vec{x}$

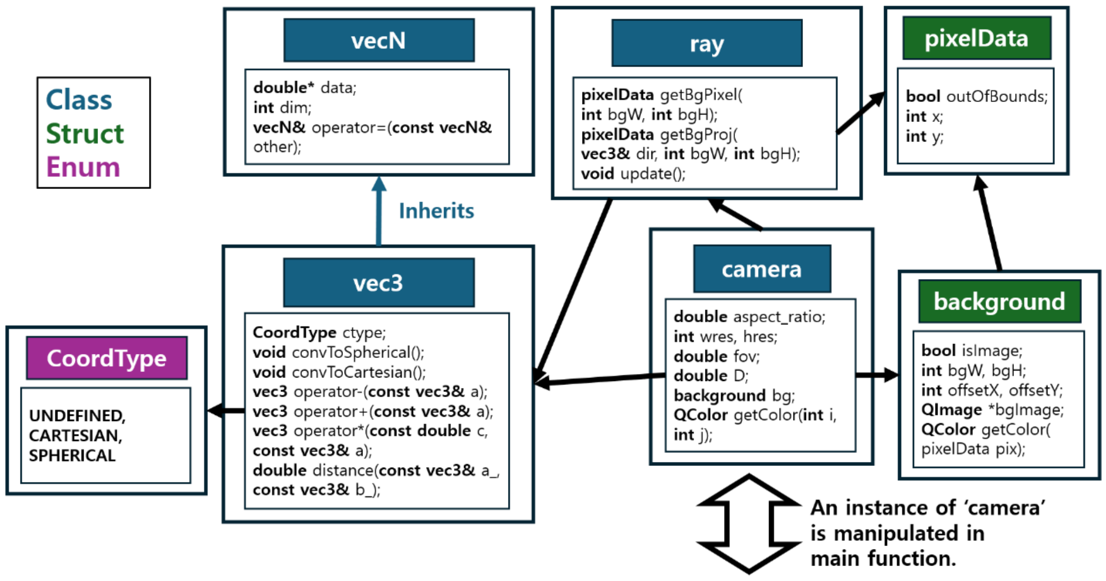
[Fig.3] The derivation of formulas for finding β(2), rotating the plane by β(5), and finding the corresponding background pixel for a traced light ray. (6)

3.4 Project code structure


QtApp [qt]
CMakeLists.txt
QtApp
Forms
mainwindow.ui
Header Files
camera.h
mainwindow.h
ray.h
vecN.h
Source Files
main.cpp
mainwindow.cpp
Handles input/output in program UI
camera class, background data structure
ray class, pixel data structure
N-dim and 3-dim vector classes
Where the main function resides

[Fig. 4] The file structure of my Qt project.

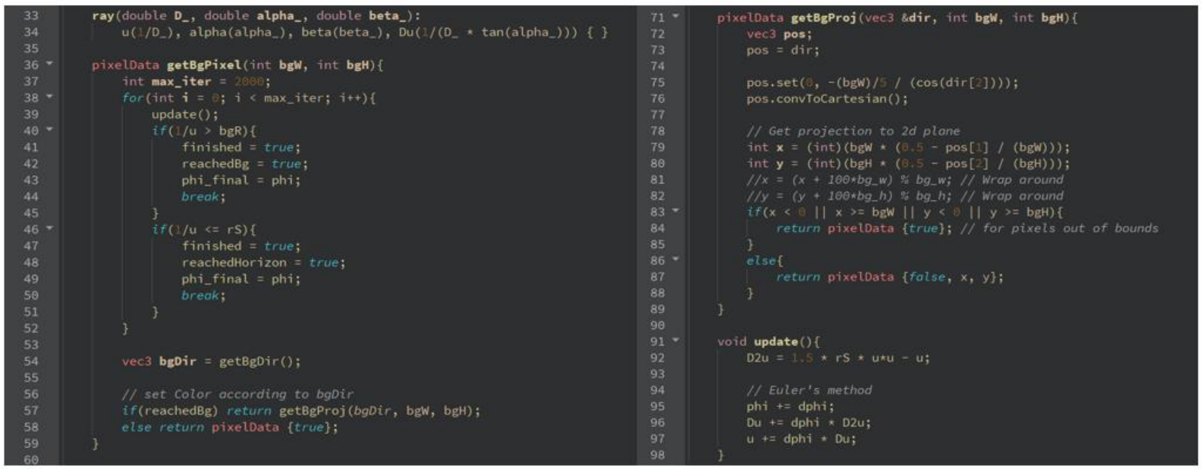
[Fig. 4] Shows the project's overall code structure. The project comprises files that handle input and output functions in the UI, and files that implement some classes needed for ray tracing computations.


Class
Struct
Enum
vecN
double* data;
int dim;
vecN& operator=(const vecN& other);
Inherits
ray
pixelData getBgPixel(int bgW, int bgH);
pixelData getBgProj(vec3& dir, int bgW, int bgH);
void update();
pixelData
bool outOfBounds;
int x;
int y;
vec3
CoordType ctype;
void convToSpherical();
void convToCartesian();
vec3 operator-(const vec3& a);
vec3 operator+(const vec3& a);
vec3 operator*(const double c, const vec3& a);
double distance(const vec3& a_, const vec3& b_);
CoordType
UNDEFINED,
CARTESIAN,
SPHERICAL
camera
double aspect_ratio;
int wres, hres;
double fov;
double D;
background bg;
QColor getColor(int i, int j);
background
bool isImage;
int bgW, bgH;
int offsetX, offsetY;
QImage *bgImage;
QColor getColor(pixelData pix);
An instance of 'camera' is manipulated in main function.

[Fig. 5] The class diagram for my project.

[Fig. 5] Shows the class diagram for the classes, structures, and enumerator types defined at `vecN.h`, `ray.h`, and `camera.h`. First, I implemented the vec3 class which inherits from the vecN class, which represents a vector in 3-dimensional space in either Cartesian coordinates or spherical coordinates.

The ray tracing algorithm is implemented in the ray class, defined at `ray.h`. Given a light ray's initial position and direction, the getBgPixel() method of a ray instance computes the light's path and returns the corresponding background pixel position. The numerical solving of Eq. 2 is performed in the update() method. It uses Euler's method to calculate how the variable u changes as ϕ changes in a small update step ($d\phi = 0.003$). The code block corresponding to the algorithm is shown at [Fig. 6].



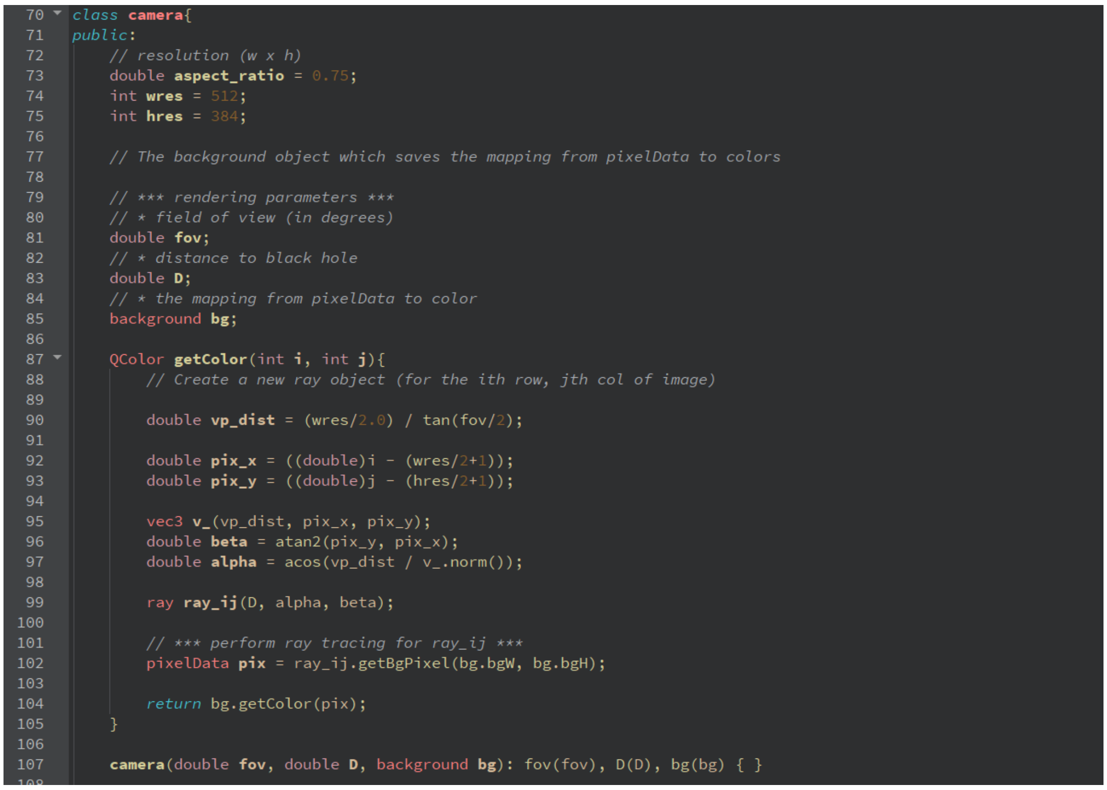
```cpp
    ray(double D_, double alpha_, double beta_):
        u(1/D_), alpha(alpha_), beta(beta_), Du(1/(D_ * tan(alpha_))) { }

    pixelData getBgPixel(int bgW, int bgH){
        int max_iter = 2000;
        for(int i = 0; i < max_iter; i++){
            update();
            if(1/u > bgR){
                finished = true;
                reachedBg = true;
                phi_final = phi;
                break;
            }
            if(1/u <= rS){
                finished = true;
                reachedHorizon = true;
                phi_final = phi;
                break;
            }
        }

        vec3 bgDir = getBgDir();

        // set Color according to bgDir
        if(reachedBg) return getBgProj(bgDir, bgW, bgH);
        else return pixelData {true};
    }
```

```cpp
    pixelData getBgProj(vec3 &dir, int bgW, int bgH){
        vec3 pos;
        pos = dir;

        pos.set(0, -(bgW)/5 / (cos(dir[2])));
        pos.convToCartesian();

        // Get projection to 2d plane
        int x = (int)(bgW * (0.5 - pos[1] / (bgW)));
        int y = (int)(bgH * (0.5 - pos[2] / (bgH)));
        //x = (x + 100*bg_w) % bg_w; // Wrap around
        //y = (y + 100*bg_h) % bg_h; // Wrap around
        if(x < 0 || x >= bgW || y < 0 || y >= bgH){
            return pixelData {true}; // for pixels out of bounds
        }
        else{
            return pixelData {false, x, y};
        }
    }

    void update(){
        D2u = 1.5 * rS * u*u - u;

        // Euler's method
        phi += dphi;
        Du += dphi * D2u;
        u += dphi * Du;
    }
```

[Fig. 6] the ray tracing algorithm implemented at `ray.h`. The getBgPixel() method returns the corresponding background pixel given a light ray's initial condition. The update() method numerically solves Eq.2.

An instance of the camera class, defined at `camera.h`, is directly manipulated from the main function. A camera instance stores not only the distance and FOV value (which is used to construct initial conditions for individual light rays) but also stores the background image data. The getColor() method creates a ray instance, computes its corresponding background pixel position, and returns the pixel color according to the background image data. So, the camera class directly provides the method needed to construct a full image of the background, distorted by the gravitational lens effect.

[Fig. 7] shows a code block from the camera class.



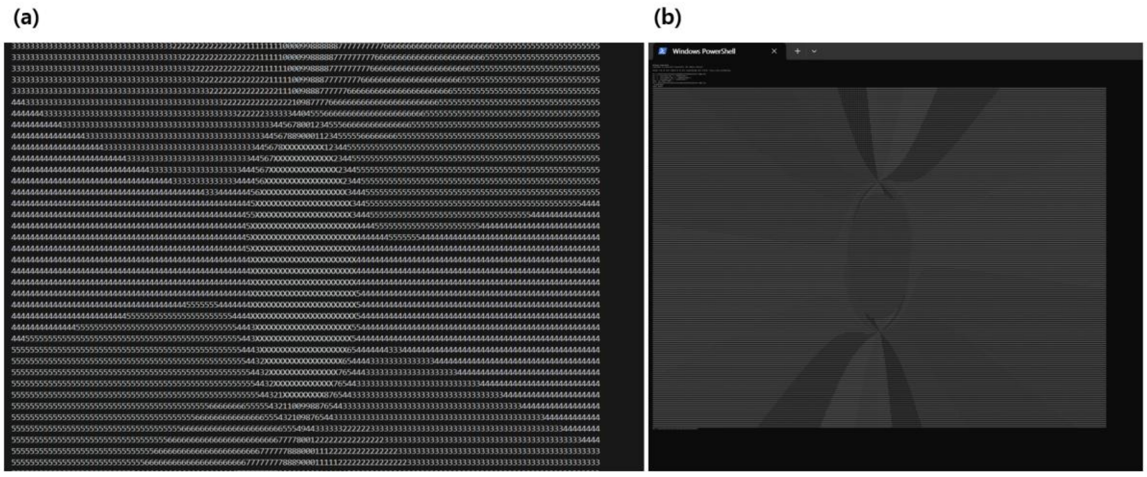
```cpp
class camera{
public:
    // resolution (w x h)
    double aspect_ratio = 0.75;
    int wres = 512;
    int hres = 384;

    // The background object which saves the mapping from pixelData to colors

    // *** rendering parameters ***
    // * field of view (in degrees)
    double fov;
    // * distance to black hole
    double D;
    // * the mapping from pixelData to color
    background bg;

    QColor getColor(int i, int j){
        // Create a new ray object (for the ith row, jth col of image)

        double vp_dist = (wres/2.0) / tan(fov/2);

        double pix_x = ((double)i - (wres/2+1));
        double pix_y = ((double)j - (hres/2+1));

        vec3 v_(vp_dist, pix_x, pix_y);
        double beta = atan2(pix_y, pix_x);
        double alpha = acos(vp_dist / v_.norm());

        ray ray_ij(D, alpha, beta);

        // *** perform ray tracing for ray_ij ***
        pixelData pix = ray_ij.getBgPixel(bg.bgW, bg.bgH);

        return bg.getColor(pix);
    }

    camera(double fov, double D, background bg): fov(fov), D(D), bg(bg) { }
```

[Fig. 7] the camera class and getColor() method implemented at `camera.h`.

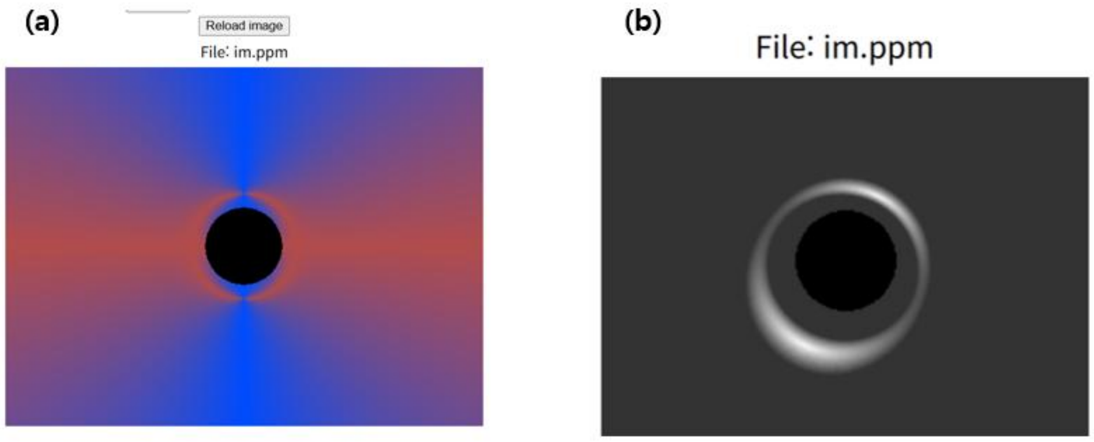
4. Development Progress

4.1. 3-Stage development progress

As described in the sections above, I first implemented the classes and the core algorithm first and tested them in a CUI program(Stage 1 and 2). [Fig. 8] and [Fig. 9] show the early-stage testing results.

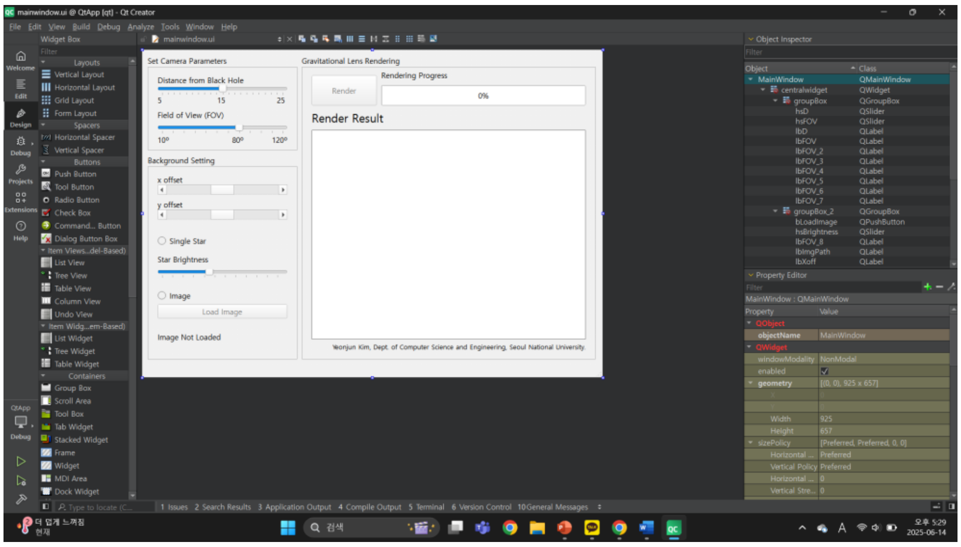


[Fig. 8] Ray tracing algorithm results in CUI. (Results are printed in ASCII image, the numbers are assigned according to where the light rays landed.) You can see some distortion around the black hole in the middle of the image. (a) output of width=128. (b) output of width=512.



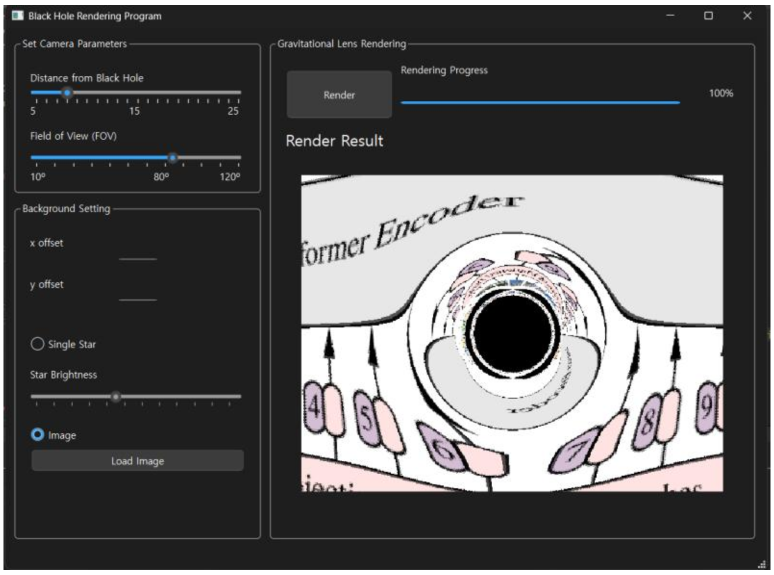
[Fig. 9] Testing results in a 512*384 PPM image format, viewed with an online PPM image viewer(https://www.cs.rhodes.edu/welshc/COMP141_F16/ppmReader.html). (a) The background pixels are colored red if it is close to the equator, and colored blue if it is close to the poles. (b) A rendering of an Einstein ring. (An image produced by a single star right behind the black hole.)

Once I confirmed that the ray tracing algorithm was implemented correctly, I moved on to Stage 3 and started developing the GUI. [Fig. 10] shows the GUI design interface of the Qt Creator IDE.



[Fig. 10] The UI design interface of Qt Creator IDE.

[Fig. 11] shows the complete program UI.

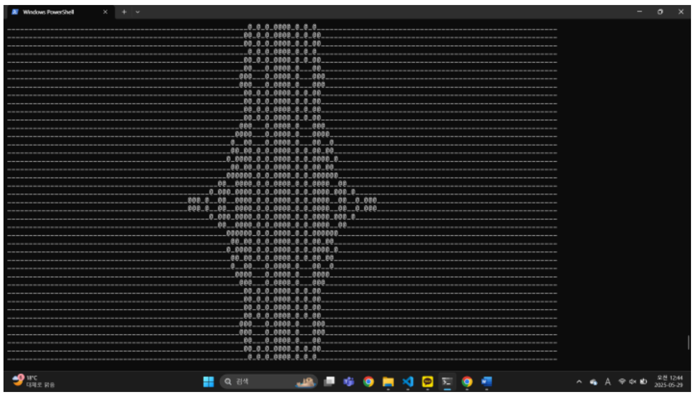


[Fig. 11] The complete program UI. (I put a random background image for testing…)

4.2. Issues during development

Before I got successful ray tracing results as shown in [Fig. 8], I had some issues with the algorithm. At first, I tried naively solving Eq.1 in a four-dimensional spacetime coordinate. However, two problems arose. The first problem was that each computation step was very costly (since it involves calculating numerous values for the metric and Christoffel symbols). The second problem was that the light path's step size easily exploded or vanished, due to poor parameterization.

To resolve this problem, I read more materials such as [1] and implemented a better algorithm described at section 3.3. Hopefully, I got the desired results thereafter. [Fig. 12] shows an erroneous rendering result I got with the naïve algorithm.



[Fig. 12] An erroneous rendering result of the naïve algorithm. The image looks very glitchy due to exploding or vanishing update steps.

5. Program Manual and Demonstration

5.1. How to run and use the program

The program build is located at the **./build** directory of the submission zip file. You will see the executable "BlackHoleSimulator.exe" there. (It was built and tested in **Windows 11** environment)

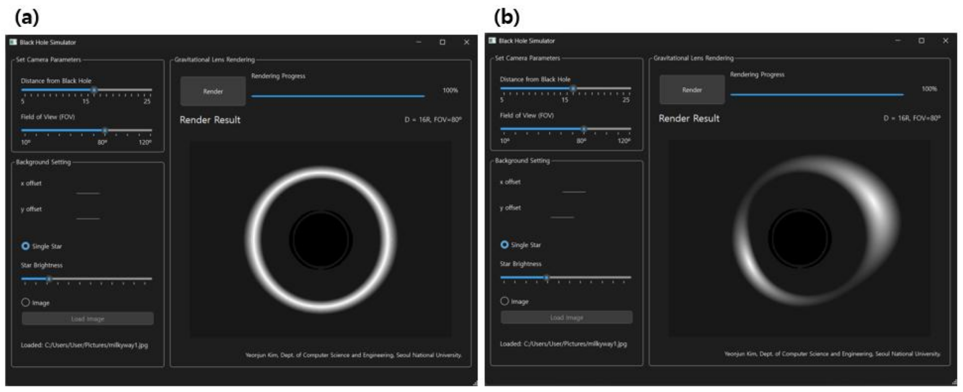
When you run the program, you will see the user interface as seen in [Fig. 11]. You can configure the camera setting, through two horizontal sliders 'D' and 'FOV'. You can also configure the background. When you hit one of the two radio buttons, you can either set the background as a single star to simulate the Eistein ring or set the background as your custom image. If the settings are complete, you can hit the 'Render' button to run the rendering algorithm. It takes little time to complete. The progress percentage is displayed at the progress bar. When the rendering is complete, the rendered result will appear at the right bottom side of the window.

The source code is in the **./source** directory, but it is also available at my GitHub.

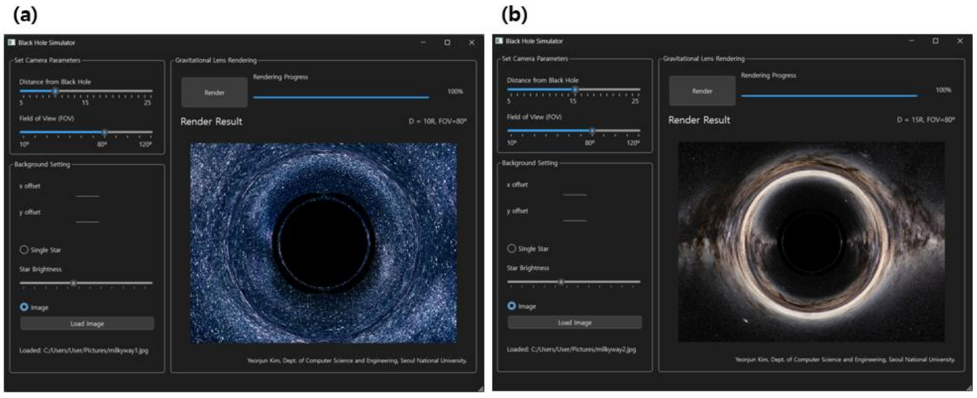
https://github.com/Superfish83/BlackHoleSimulator

5.2. Program result

[Fig. 13] and [Fig.14] demonstrate some simulation results of gravitational lens effect.



[Fig. 13] Simulation result of (a) a perfect Einstein ring, (b) a star slightly deviated from the center.



(a) D = 10R, FOV=80°
Loaded: C:/Users/User/Pictures/milkyway1.jpg

(b) D = 15R, FOV=80°
Loaded: C:/Users/User/Pictures/milkyway2.jpg

Yeonjun Kim, Dept. of Computer Science and Engineering, Seoul National University.

[Fig. 14] Simulation results of the milky way, distorted by the gravitation lens effect.

6. Limitations & Future Development Plan

In my project, I successfully implemented the simulation algorithm for the gravitational lens effect of a Schwarzschild black hole. However, the resolution and rendering throughput is still quite disappointing. In the future, I may implement more advanced optimization techniques introduced in [1], such as skipping ray computations with linear interpolation algorithms.

Simulating gravitational lens effect of more general black holes (like Kerr black holes, which have nonzero angular momentum), or continuous mass distribution is a much more challenging task. To further improve my program into more general simulation software, I will need to study more physics and mathematics as well as computer graphics algorithms.

Appendix 1: The Physics Underlying the Simulation Algorithm

In General Relativity, the information about a curved spacetime is represented with the metric tensor, which can be expressed as a 4x4 matrix. For example, the Minkowski metric, the metric of a flat spacetime, is expressed as Eq.3.

$$\eta = \begin{pmatrix} -c^2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

The elements of the metric tensor are then used to find the line element in the curved spacetime, which then defines the distance between two points.

$$ds^2 = -c^2dt^2 + dx^2 + dy^2 + dz^2 \tag{4}$$

The metric is determined by the distribution of energy over spacetime. The following equation Eq.5 (Einstein field equation) states the relationship between the geometry of spacetime (the left side) and the distribution of energy and matter (the right side) [3]

$$G_{\mu\nu} + \Lambda g_{\mu\nu} = \kappa T_{\mu\nu} \tag{5}$$

Or, equivalently,

$$R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu} \tag{6}$$

The **Schwarzschild metric** is one of the solutions of Eq. 5. It states the spacetime around a single black hole, which is not spinning and has no electrical charge. (Such black holes are called 'Schwarzschild black holes'.) Eq. 7 is the Schwarzschild metric expressed in spherical coordinates (ct, r, θ, ϕ).

$$g_{\mu v} = \begin{pmatrix} -\left(1 - \frac{2GM}{rc^2}\right) & 0 & 0 & 0 \\ 0 & \left(1 - \frac{2GM}{rc^2}\right)^{-1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \sin^2 \theta \end{pmatrix} \tag{7}$$

When $r = \frac{2GM}{c^2}$, g_{00} becomes 0 and g_{11} explodes. This special place is called the **event horizon**, and the special radius $R_S = \frac{2GM}{c^2}$ is called the **Schwarzschild radius** of the black hole.

The metric tells the line element in the curved spacetime, as express in Eq. 8.

$$ds^2 = -\left(1 - \frac{2GM}{rc^2}\right) c^2 dt^2 + \left(1 - \frac{2GM}{rc^2}\right)^{-1} dr^2 + r^2 d\theta^2 + r^2 \sin^2 \theta \, d\phi^2 \tag{8}$$

General Relativity states that objects, even light, move through the shortest path in spacetime. Such lines are called **geodesics**. As the massive object yields curvature in spacetime, the path of light also curves, and creates distorted images of objects in the background. This phenomenon is called **gravitational lens**.

Then how do we find geodesic? Now that we know the relationship between the metric tensor and the line element, we can define the total distance between point A and B, denoted L. (Eq. 9)

$$L = \int_A^B ds = \int_A^B \sqrt{\left| g_{ij} \frac{dx^i}{d\lambda} \frac{dx^j}{d\lambda} \right|} \, d\lambda \tag{9}$$

Now we apply calculus of variations, so that the light's path (a parametric curve parametrized with λ) minimizes L. This yields the **geodesic equation**, or Eq. 1.

$$\frac{d^2 x^\mu}{d\lambda^2} + \Gamma^\mu_{\rho\sigma} \frac{dx^\rho}{d\lambda} \frac{dx^\sigma}{d\lambda} = 0 \tag{1}$$

Where $\Gamma^\mu_{\rho\sigma}$ is the Christoffel symbol, a geometric quantity derived from the metric tensor.

$$\Gamma^\mu_{\rho\sigma} = \frac{1}{2} g^{\mu v} \left(\partial_\rho g_{\sigma v} + \partial_\sigma g_{v\rho} - \partial_v g_{\rho\sigma} \right) \tag{2}$$

Solving this will lead to the solution, which becomes the light's path near a black hole!

Given the initial condition of a light ray (as follows), we can numerically compute the path using Euler's method.

$$x^\mu(0), \qquad \frac{dx^\mu}{d\lambda}(0), \qquad \frac{d^2 x^\mu}{d\lambda^2}(0)$$

The equation used for update step will be expressed as Eq. 10a, 10b, and 10c. However, this method failed due to large computational cost and error, as described in section 4.2.

$$\frac{d^2 x^\mu}{d\lambda^2}(\lambda_0) = \left| -\Gamma^\mu_{\rho\sigma} \frac{dx^\rho}{d\lambda} \frac{dx^\sigma}{d\lambda} \right|_{\lambda = \lambda_0} \tag{10a}$$

$$\frac{dx^\mu}{d\lambda}(\lambda_0+\delta\lambda) = \frac{dx^\mu}{d\lambda}(\lambda_0) + \delta\lambda\left[\frac{d^2x^\mu}{d\lambda^2}(\lambda_0)\right] \tag{10b}$$

$$x^\mu(\lambda_0+\delta\lambda) = x^\mu + \delta\lambda\left[\frac{dx^\mu}{d\lambda}(\lambda_0)\right] + \frac{(\delta\lambda)^2}{2}\left[\frac{d^2x^\mu}{d\lambda^2}(\lambda_0)\right] \tag{10c}$$

Appendix 2: Deriving Eq.3

Instead of working in the 4-dimensional spacetime, I used a more optimized method introduced in section 3.3. It reduces Eq. 1 into a much simpler equation Eq. 3. This section will cover the derivation procedure of Eq. 3, from Eq. 1 and the Schwarzschild metric.

Let me first clearly write the summation symbols in Eq. 1 and Eq. 2 which were omitted according to Einstein summation rule.

$$\frac{d^2x^\mu}{d\lambda^2} = -\sum_{\rho=1}^{4}\sum_{\sigma=1}^{4}\Gamma^\mu_{\rho\sigma}\frac{dx^\rho}{d\lambda}\frac{dx^\sigma}{d\lambda} \tag{1a}$$

$$\Gamma^\mu_{\rho\sigma} = \frac{1}{2}\sum_{v=1}^{4} g^{\mu v}\left(\partial_\rho g_{\sigma v} + \partial_\sigma g_{v\rho} - \partial_v g_{\rho\sigma}\right) \tag{2a}$$

Since g is diagonal, Eq. 2a simplifies to

$$\Gamma^\mu_{\rho\sigma} = \frac{1}{2}\left\{g^{\mu\sigma}\left(\partial_\rho g_{\sigma\sigma}\right) + g^{\mu\rho}\left(\partial_\sigma g_{\rho\rho}\right) + g^{\mu\mu}\left(\partial_\mu g_{\rho\sigma}\right)\right\} \tag{2b}$$

Recall Eq.7, the metric tensor. But now, we only care for the equator in the coordinate system. i.e. only where r and ϕ change, and θ is fixed to $\theta = \frac{\pi}{2}$. (See [Fig. 2]) So,

$$g_{\mu v} = \begin{pmatrix} -\left(1-\frac{R_S}{r}\right) & 0 & 0 & 0 \\ 0 & \left(1-\frac{R_S}{r}\right)^{-1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \end{pmatrix} \tag{11}$$

Now, let's find the partial derivatives and Christoffel symbols.

We can see

$$\partial_r g_{tt} = -\frac{R_S}{r^2}, \qquad \partial_r g_{rr} = -\frac{R_S}{(r-R_S)^2}, \qquad \partial_r g_{\theta\theta} = \partial_r g_{\phi\phi} = 2r$$

(All other partial derivatives are zero.) These yield

$$\Gamma^r_{tt} = \frac{1}{2}g^{rr}(\partial_r g_{tt}) = \frac{1}{2}\left(\frac{R_S}{r}-1\right)\left(-\frac{R_S}{r^2}\right) = \frac{R_S}{2r^2}\left(1-\frac{R_S}{r}\right) \tag{12}$$

$$\Gamma^r_{rr} = \frac{1}{2}g^{rr}(\partial_r g_{rr}) = \frac{1}{2}\left(\frac{r-R_S}{r}\right)\left(-\frac{R_S}{(r-R_S)^2}\right) = -\frac{R_S}{2r(r-R_S)} \tag{13}$$

$$\Gamma^r_{\phi\phi} = \frac{1}{2}g^{rr}\left(\partial_r g_{\phi\phi}\right) = -\frac{1}{2}\left(\frac{r-R_S}{r}\right)(2r) = R_S - r \tag{14}$$

(Other Christoffel symbols are zero, and we don't care for $\Gamma^r_{\theta\theta}$ since $d\theta = 0$)

Putting these into Eq.1,

$$\frac{d^2r}{d\lambda^2} = -\frac{R_S}{2r^2}\left(1-\frac{R_S}{r}\right)\left(\frac{d\phi}{d\lambda}\right)^2 + \frac{R_S}{2r(r-R_S)}\left(\frac{dr}{d\lambda}\right)^2 - (R_S - r)\left(\frac{d\phi}{d\lambda}\right)^2 \tag{16}$$

Here, the conservation laws for energy and angular momentum dictates:

$$\left(\frac{dt}{d\lambda}\right) = \frac{E}{1-\frac{R_S}{r}}, \left(\frac{d\phi}{d\lambda}\right) = \frac{L}{r^2} \tag{17}$$

Where E and L are (constant) energy and angular momentum, respectively. Basically, we put Eq. 17 into Eq. 16 to derive Eq.3.

We also change the variable, so it won't get too messy. Let

$$u = \frac{1}{r}$$

Then we keep simplifying Eq.16 with chain rule and Eq. 17. Because the algebra is too heavy, I will not write the full derivation process here. But this boils down to the **photon orbit equation**.

$$\frac{d^2u}{d\phi^2} = \frac{3R_S}{2}u^2 - u \tag{3}$$

References

[1] https://github.com/Python-simulation/Black-hole-simulation-using-python

[2] Wikipedia. *Ray Tracing*. https://en.wikipedia.org/wiki/Ray_tracing_(graphics). Retrieved in 2025-06-12.

[3] S. M. Carroll. (2004). *Spacetime and Geometry: An Introduction to General Relativity*. Addison Wesley.