

Ray Tracing a Black Hole in C#

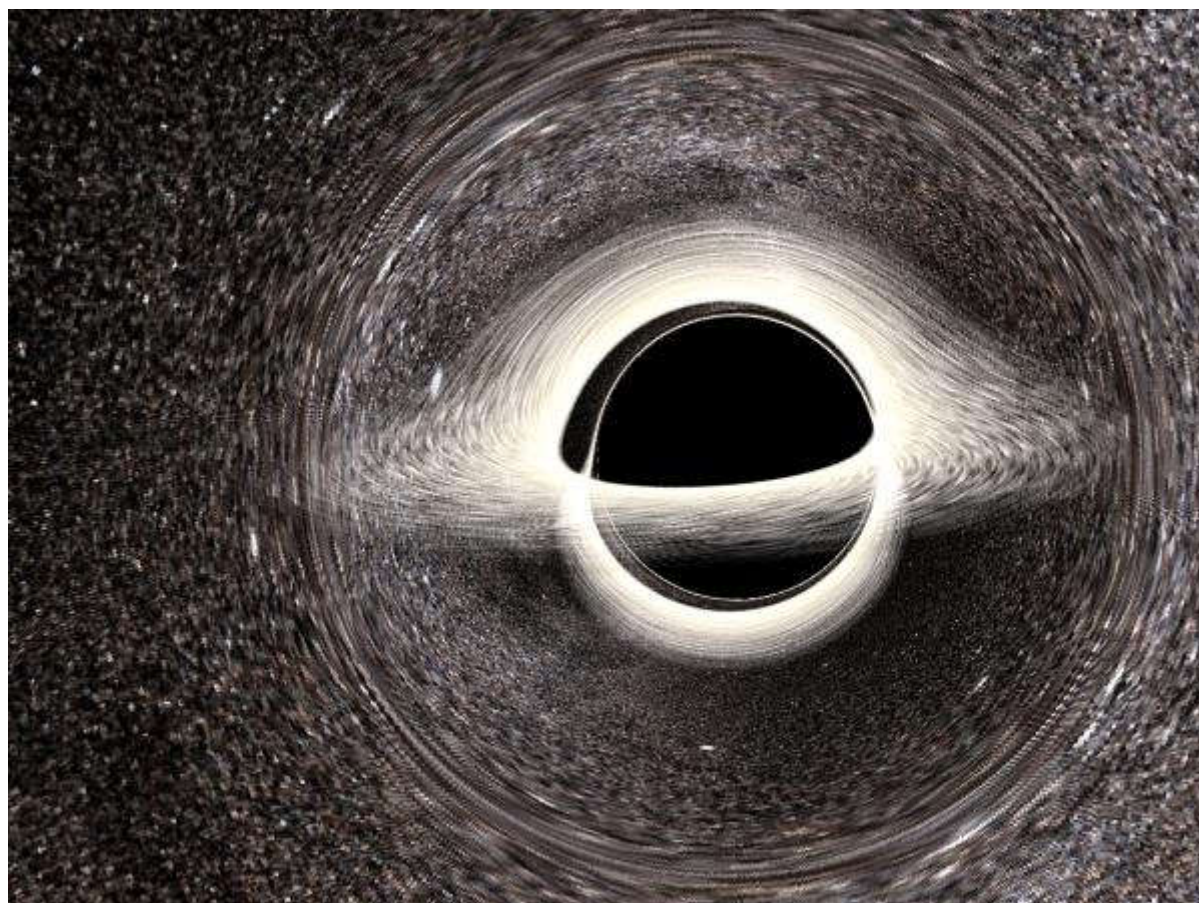


Mikolaj Barwicki, 24 May 2015

Approach to visualisation of black hole surrounding using non-linear ray-tracing. Inspired by "Interstellar"



[Download source code from GitHub](#)



Note

This article uses MathJax engine to render math equations.

Introduction

So, how does a black hole look like?

A naive answer is: well, since no light can escape the extreme gravity of a black hole, it must be ultimately black.

And that is true, but it's not as interesting as a question of how light behaves **around** a black hole, and what effects can be observed. They can be quite spectacular.

The movie "Interstellar" is inspiring, because it includes extraordinary renderings of a hypothetical black hole, with a radiating disc of matter revolving around it (called "accretion disc"). According to some of the movie's "Making of..." materials, this is probably the first time in history where black hole renderings which were generated so accurately to what astrophysicists predict.

Are some of these effects possible to recreate on a regular PC? To some extent yes, using a fairly simple, standard computer graphics technique, fairly complex math, robust numerical analysis methods and parallel computing concepts.

The code presented in this article is an extended version of the approach presented in [1], enhanced with texturing, camera path calculation and a rudimentary parallel computing mechanism.

Background

In order to appreciate the article, it is important to have rudimentary awareness of concepts listed below.

Ray Tracing

[Ray tracing](#) is a standard computer graphics technique used to simulate images of 3-D scenes modelled in computer memory. Essentially, it involves mathematical simulation of imaginary "light rays" sent from the observer into the scene. A ray tracing algorithm simulates the behaviour of light as it hits objects on the scene, taking into the account surface shading, reflection and refraction. It is well suited for simulation of various physical phenomena, and is capable of rendering artificial images of exquisite level of realism.

Black Holes Theory

Wikipedia has a very elaborate article on [black holes](#). For the purposes of this article, it is interesting to realize non-obvious behaviour of light travelling close to a black hole - the effect called [gravitational lensing](#). Gravity bends light rays so that looking at a black hole, with an accretion disc, we are actually looking at the front of the disc as well as at the back of the disc, observed from above, and below, simultaneously!

The visualisation technique requires to calculate the color of each image's pixel from the attributes of an observed object which is hit by a ray of light cast from the observer to the observed object. This is (relatively) simple in "normal" ray-tracing, where light rays are straight. However, in the vicinity of a source of extreme gravity, which a black hole is, light no longer travels along straight lines. Therefore, it is necessary to simulate trajectories of photons hitting the observer.

Kerr Metric

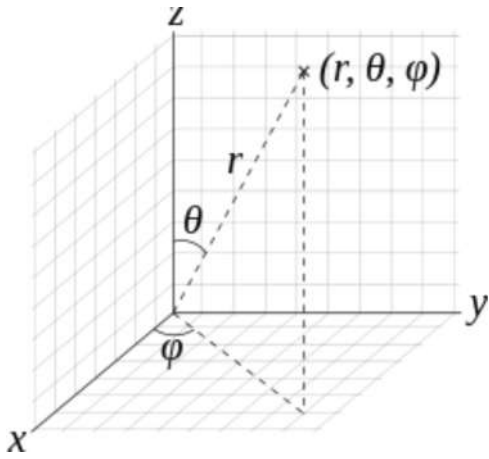
The code described in this article is based on math derived from [Einstein's field equations](#) of general relativity. The simulation is based on the solution known as [Kerr metric](#), which describes geometry of spacetime around an uncharged, spherically-symmetric, and rotating black hole.

The trajectory of a photon is derived from the line element equation expressed in [Boyer-Lindquist coordinates](#):

$$ds^2 = -\frac{\Delta}{\Sigma}(dt - a\sin^2\theta d\phi)^2 + \frac{\sin^2\theta}{\Sigma}((r^2 + a^2)d\phi - a dt)^2 + \frac{\Sigma}{\Delta}dr^2 + \Sigma d\theta^2$$

where:

r, θ and ϕ are spherical coordinates as used in physics:



(source: Wikipedia)

and:

$$\Delta = r^2 - 2Mr + a^2$$

$$\Sigma = r^2 + a^2 \cos^2 \theta$$

a = angular momentum factor - non-zero since our black hole is rotating

The equation above can be used to formulate a system of Ordinary Differential Equations (ODEs) that describe path of a photon travelling through the field around the black hole.

With an ability to calculate trajectory of a photon, it is possible to calculate light rays for the purposes of ray tracing algorithm. The fact that the rays' equations are expressed in spherical coordinates (r, θ, ϕ) rather than commonly used Cartesian coordinates (x, y, z) **does not matter** - as it is the color of the point hit by the ray which is required, and not its coordinates.

Runge-Kutta Method

In order to simulate the trajectory of a photon, it is necessary to employ numerical analysis. This code uses Cash-Karp method of calculating numerical solutions of an ODE system. The Cash-Karp method belongs to a generic family of [Runge-Kutta](#) numerical integration methods.

The Program

Implementation of Kerr Metric to Determine Photon Trajectory

The math used to calculate trajectories of photons is quoted below for illustration. Understanding it is **not necessary** to understand the idea of the rendering engine. The equations are just formulas which, when processed by the generic numerical analysis engine, allow for ray intersection detection, and thus for determining pixel color.

To reduce the complexity of calculations, a number of variables are assumed to be 'normalized', i.e., M (mass of black hole) and E (energy of photon) are assumed to be 1, mass of photon is assumed to be 0. L (orbital angular momentum) and K (Carter's constant) need to be calculated for each ray.

The following system of eight differential equations describe the path of a photon in the vicinity of a Kerr black hole (as presented in [1]):

$$r' = \frac{\Delta}{\Sigma} p_r$$

$$\theta' = \frac{1}{\Sigma} p_\theta$$

$$\phi' = \frac{2ar + (\Sigma - 2r) \frac{L}{\sin^2 \theta}}{\Delta \Sigma}$$

$$t' = \frac{1 + (2r(r^2 + a^2) - 2arL)}{\Delta\Sigma}$$

$$p'_r = \frac{(r-1) * (-K) + 2r(r^2 + a^2) - 2aL}{\Delta\Sigma} - \frac{2p_r^2(r-1)}{\Sigma}$$

$$p'_\theta = \frac{\sin \theta \cos \theta (\frac{L^2}{a^2} - a^2)}{\Sigma}$$

$$p'_\phi = 0$$

$$p'_t = 0$$

Note that in the code, all coefficients are negated. This is because we are simulating the photon's position "backwards" - from the observer to the photon's origin.

Now t' , p_ϕ and p'_t can be ignored, as they don't impact the shape of photon's trajectory. In the end, five ODEs are integrated using Runge-Kutta algorithm.

The code implementing these equations is found in:

```
public class KerrBlackHoleEquation : IODESystem
{
    ...
    /// <summary>
    /// Function that returns the equations that are included in the
    /// coupled differential equation set.
    /// Coupled differential equations describing motion of photon.
    /// </summary>
    /// <param name="y">Vector describing current state of the ODE system</param>
    /// <param name="dydx">Coefficient vector of the differential equations</param>
    public unsafe void Function(double* y, double* dydx) {...}
    ...
}
```

Initial Conditions

For the numerical integration to complete, each ray requires a set of initial conditions, i.e., the values of ODE system coefficients that describe the initial state of the calculation. This corresponds to the state of the simulation at the point of observer, so the initial conditions need to take into account the 'target' of the ray. In other words, it is necessary to be able to 'direct' a ray at a selected coordinate of the image pixel space, and this must be reflected in the ODE system's initial state values for expressed in spherical coordinates. Thus the input to ray's calculation will be (r_0, θ_0, ϕ_0) - the observer location and (x_0, y_0) - coordinates in pixel space. The pixel coordinates transform into initial values of p'_r and p'_θ :

$$p'_{r0} = \frac{\Sigma \cos x_0 \cos y_0}{\Delta\epsilon}$$

$$p'_{\theta0} = \Sigma \frac{\sin y_0}{\alpha\epsilon}$$

where α is an aperture coefficient.

Moreover, two constants (**L** and **K**) need to be calculated for the purposes of the ray's computation (as presented in [1]):

$$s = \Sigma - 2r_0$$

$$\epsilon = \sqrt{s \frac{r_0'^2}{\Delta} + s \theta_0'^2 + \Delta \sin^2 \theta_0 \phi_0'^2}$$

$$L = \frac{(\Sigma \Delta \phi_0' - 2ar_0 \epsilon) \sin^2 \theta_0}{s \epsilon}$$

$$K = (\theta_0' \Sigma)^2 + A^2 \sin^2 \theta_0 + \frac{L^2}{\sin^2 \theta_0}$$

The above are implemented in here:

```
public class KerrBlackHoleEquation : IODESystem
{
    ...
    /// <summary>
    /// Set initial conditions for a starting point of the ray that is being simulated.
    /// </summary>
    /// <param name="y0">Vector of coefficients describing state of the system</param>
    /// <param name="ydot0">Vector of ODE coefficients that is initialized by this function
    call</param>
    /// <param name="x">x-coordinate of the ray</param>
    /// <param name="y">y-coordinate of the ray</param>
    public unsafe void SetInitialConditions(double* y0, double* ydot0, double x, double y) {...}
    ...
}
```

Also, the radius of black hole's event horizon is calculated:

$$r_{hor} = 1 + \sqrt{1 - a^2}$$

Accretion Disc

The accretion disc around the black hole extends from arbitrarily selected radius to the innermost stable orbit, which is also calculated from black hole's parameters (as indicated in [4]):

$$r_s = 3 + Z_2 - \sqrt{(3 - Z_1)(3 + Z_1 + 2Z_2)}$$

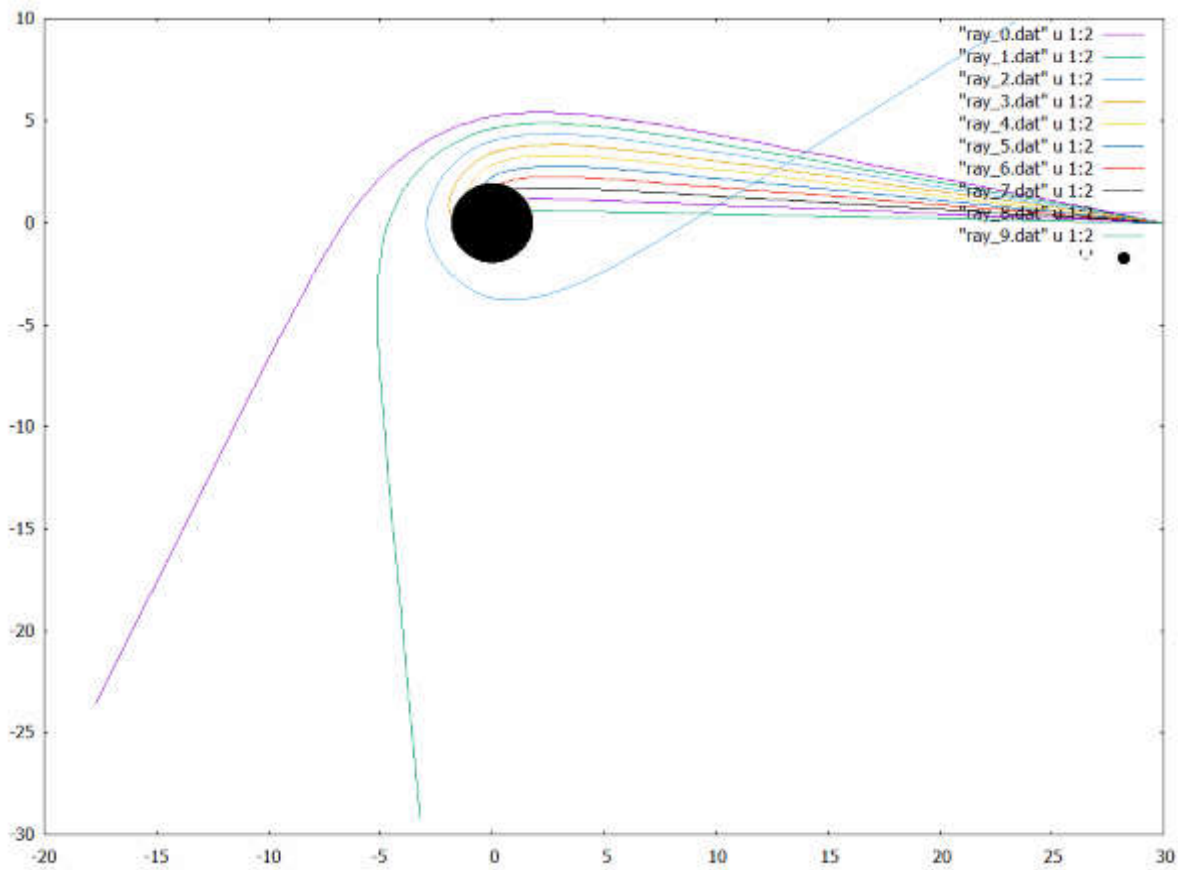
where:

$$Z_1 = 1 + \sqrt[3]{1 - a^2} (\sqrt[3]{1 + a} + \sqrt[3]{1 - a})$$

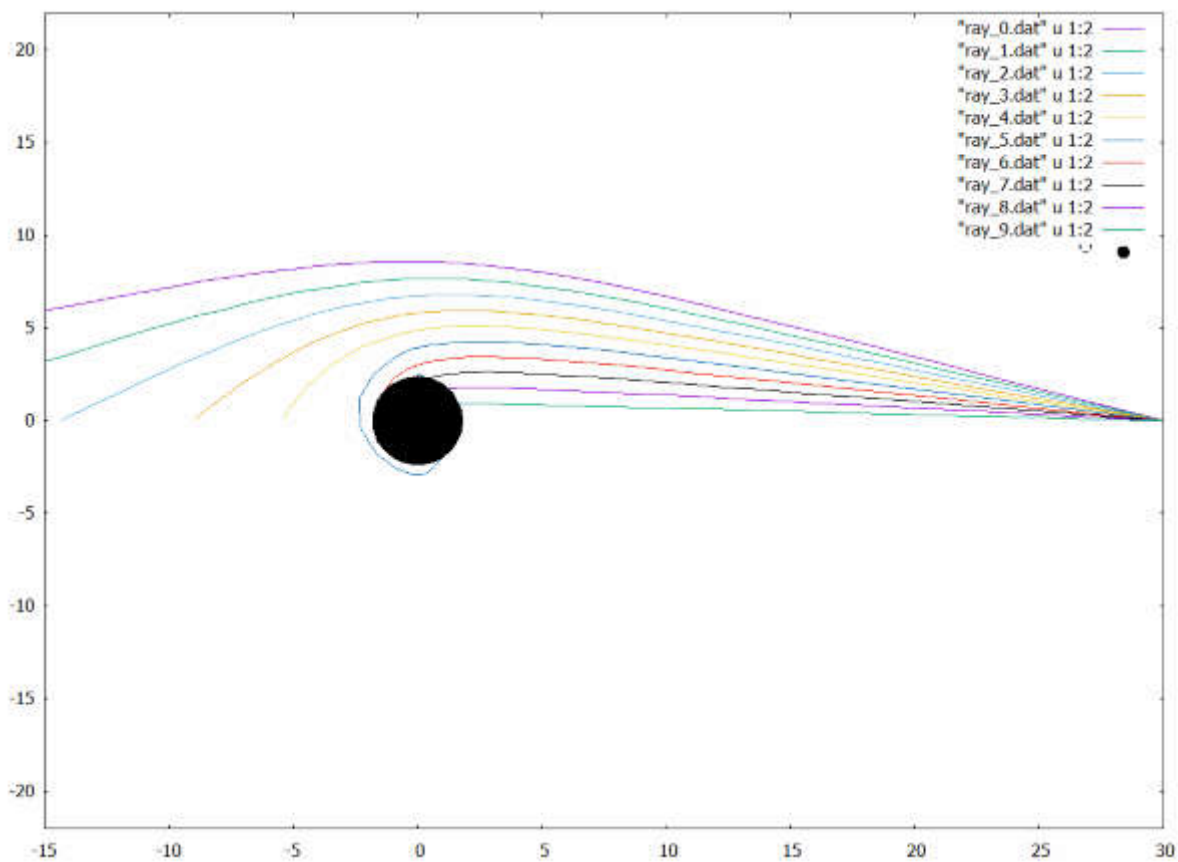
$$Z_2 = \sqrt{3a^2 + Z_1^2}$$

Sample Ray Paths

The following diagrams illustrate sample paths of rays calculated by the program. These were generated by plotting several rays cast along a vertical plane cross-cutting determined by black hole's central point (the singularity) and the observer's location. The spherical coordinates were recalculated to Cartesian, and plots were drawn along XZ axes (x - horizontal, z - vertical):



Here the rays are hitting the accretion disc, so they end on a coordinate $z = 0$:



Note how light rays bend - this means that an observer of a black hole will actually see light coming from the universe around the scene. This implies peculiar requirements for a texture used to model the surrounding universe - the texture must be 'all around'!

Ray Intersection Detection

As ray's path is calculated, the ray tracer needs to decide whether the ray hasn't hit an object, which would conclude the computation. Three cases are possible:

1. Ray hits the black hole - this happens when $r < r_{hor}$
2. Ray escapes to infinity - in practice this is assumed when $r > r_0$
3. Ray hits the accretion disc - this happens when $\theta = \pm \frac{\pi}{2}$ and $r_s < r < r_{disc}$

In order to detect if ray hits the horizontal plane, the Runge-Kutta steps are alternated using shrinking positive and negative step values when the calculation determines the ray "hits through" the horizontal plane of $\theta = \pm \frac{\pi}{2}$. The procedure becomes sort of a "binary search" of the intersection point.

The logic that drives the ray tracing algorithm is implemented in **RayTracer** class:

```
/// <summary>
/// Ray tracer class that is responsible for all image calculations.
/// </summary>
public class RayTracer
{
    ...
}
```

Image Quality

The Ray Tracer class is able to render images with variable quality. Following levels of quality are available:

- **Low** - Calculate one ray per 4x4 square of the target image. (Useful for testing eg. camera paths.)
- **Medium** - Perform adaptive anti-aliasing on neighbouring 4x4 squares of image - if colour difference between two neighbouring 4x4 squares is larger than threshold, trace 1 ray per each pixel within the square.
- **High** - Perform adaptive anti-aliasing on neighbouring pixels - if colour difference between two neighbouring pixels is larger than threshold, perform super-sampling on pixels.
- **UltraHigh** - perform super-sampling under no conditions - 16 rays per each pixel.

Texture Mapping

There are two requirements for texturing in the scene - the accretion disc, and the universe around the scene.

The universe is modelled by mapping a high-resolution star-sky texture onto a sphere which surrounds the scene, with black hole's singularity as the center point, and R_0 of observer's location as the radius. Thus any ray that is integrated in subsequent steps of algorithm and has r coordinate exceeding R_0 is assumed as 'escaped to infinity', and its current coordinates get mapped onto the rectangular bitmap of resolution (x_{res}, y_{res}) along the following formula:

$$x = \left(\frac{\phi}{2\pi} * x_{res} \right) \mod x_{res}$$
$$y = \left(\frac{\theta}{\pi} * y_{res} \right) \mod y_{res}$$

Note how spherical coordinates make the mapping trivial and r is irrelevant for this mapping!
The high-quality, hi-res star-sky texture was taken from [here](#).

The accretion disc texture also needs to be high resolution. The mapping is calculated when ray tracer detects ray's intersection with the disc. The mapping onto a rectangular bitmap is as follows:

$$x = \left(\frac{\phi}{2\pi} * x_{res} \right) \mod x_{res}$$
$$y = \frac{(r - r_s)}{(r_{disc} - r_s)} * y_{res}$$

Camera Path

The program uses the ray tracing classes to generate subsequent frames of an animation. For an animation to be interesting, each frame needs to differ, ie. the scene settings need to follow a predefined path. The scene settings are reflected by the **SceneDescription** class:

```
/// <summary>
/// Scene description class
/// </summary>
public class SceneDescription
{
    /// <summary>
    /// Camera position - Distance from black hole
    /// </summary>
    public double ViewDistance { get; set; }

    /// <summary>
    /// Camera position - Inclination (vertical angle) in degrees
    /// </summary>
    public double ViewInclination { get; set; }

    /// <summary>
    /// Camera position - Angle (horizontal) in degrees
    /// </summary>
    public double ViewAngle { get; set; }

    /// <summary>
    /// Camera tilt - in degrees
    /// </summary>
    public double CameraTilt { get; set; }

    /// <summary>
    /// Camera aperture - need to manipulate the camera angle.
    /// </summary>
    public double CameraAperture { get; set; }

    /// <summary>
    /// Camera yaw - if we want to look sideways.
    /// Note: this is expressed in % of image width.
    /// </summary>
    public double CameraYaw { get; set; }
}
```

There needs to be a parametrized function that calculates the scene parameters for a given point in time of the animation. This is the responsibility of the **DefaultPathSceneGenerator** class:

```
public class DefaultPathSceneGenerator : ISceneGenerator
{
    public SceneDescription GetScene(int frame, double fps)
    {
        SceneDescription result = new SceneDescription();
        result.CameraAperture = 2.0;

        double t = frame / fps;

        double r = 600 - t * 2.53;

        // factor of attenuation of camera's sinusoidal motions (the closer to black hole - the
        calmer the flight is)
        double calmFactor = Math.Pow((600 - r) / 575, 20);

        double phi = t*3;
        double theta = 84
            + 8 * Math.Sin(phi * Math.PI / 180) * (1 - calmFactor) // precession
            + 3 * calmFactor;
    }
}
```



```

        result.ViewAngle = phi;
        result.ViewDistance = r;
        result.ViewInclination = theta;
        result.CameraAperture = 24.00/500.0*r + 3.2;
        result.CameraTilt = 8.0 * Math.Cos(phi * Math.PI / 180) * (1 - calmFactor);
        result.CameraYaw = calmFactor * 1.0; // we will be 'Landing' on the accretion disc...

        return result;
    }
}

```

The camera path coded above describes a spiral, which begins at $r = 600$ units and slowly revolving around the black hole, with "the fall" accelerating rapidly towards the end of the journey. At the end of the path, the camera turns slightly and appears to be "landing" on the accretion disc, for better views. The camera path also tilts the camera periodically, to simulate the orbit being slightly inclined away from the plane of the accretion disc. The whole journey takes ca. 4:30 minutes before the camera hits the event horizon.

Rendering Control File

In order to define long batches of computation (e.g., for multi-frame animations) the program uses a control file. Subsequent lines of control file have the following meaning:

- Specification of a class to be used as source of scene definitions for each frame
- Output path where frame images will be stored
- Frame resolution
- FPS - Frames per second
- Quality
- Frame lines: in the format {frame number} {frame status: 1 - pending, 2 - processing, 3 - finished}
- ...

Example of a control file is as given below:

```

GraviRayTraceSharp.Scene.DefaultPathSceneGenerator, GraviRayTraceSharp
.
640 480
25
High
1 1
2 1
3 1
...

```

Distributed Computation

The **Main()** method of the program accepts a single parameter: name of the control file to use. The file is opened in exclusive mode, and in case of failure another attempt is made within a specific amount of time. This means, that the control file becomes a trivial synchronization mechanism for computing distributed over a number of machines, e.g., in a LAN (if a control file and output path are published via UNC paths).

Using this mechanism, the code presented in the article has been used to compute a series of several thousands of 1600x1200 frames (ca. 4 minutes of 25 FPS animation). A pseudo-grid of up to 20 Intel I7-class PCs were used. The computation time in total was about 200 hours.

Not Implemented

Rendering engine presented in this article uses a very simple model of the accretion disc - it is assumed the disc is of 0 thickness, and its appearance is simulated through a texture. No red-shift effects are simulated.

Points of Interest

The project was an interesting opportunity to explore multi-discipline aspects of IT and physics:

- Computer graphics, ray tracing
- Numerical analysis - Runge-Kutta method of ODE calculations
- Distributed, parallel computing

...with a final bonus of pretty interesting images :-).

References

- [1] [Ray tracing in curved spacetime](#)
- [2] [Black hole ray tracer written in Python](#)
- [3] [Orbits in the Kerr metric](#) (Sean Marshall, Cornell University)
- [4] [Foundations of Black Hole Accretion Disk Theory](#) (Marek Abramowicz, P. Chris Fragile)
- [5] [Ray-tracing in pseudo-complex General Relativity](#)


License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Mikolaj Barwicki

Poland 

No Biography provided

Comments and Discussions

 **30 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/994466/Ray-Tracing-a-Black-Hole-in-Csharp> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | [Mobile](#)
Web04 | 2.8.171207.1 | Last Updated 24 May 2015

Article Copyright 2015 by Mikolaj Barwicki
Everything else Copyright © [CodeProject](#), 1999-2017