Ray Tracing Curved Spacetime

Stanisław Kowalski, Cameron Eure, Oscar Haase

December 5th, 2022

1 Introduction

One conclusion of Einstein's theory of General Relativity is that there must be objects massive enough that not even light can escape their gravity: the elusive galactic entities known as black holes. It's no surprise that black holes are notoriously hard to image- how can you use light to look at something that light can't escape? The secret lies in looking at what's *around* it. Black holes are very energetic and distort the spacetime around themselves quite significantly, causing accretion disks to form and light to bend, and suddenly something that should be invisible becomes very prominent. The goal of this project is to leverage these phenomena to model what it might be like to look at a real black hole.

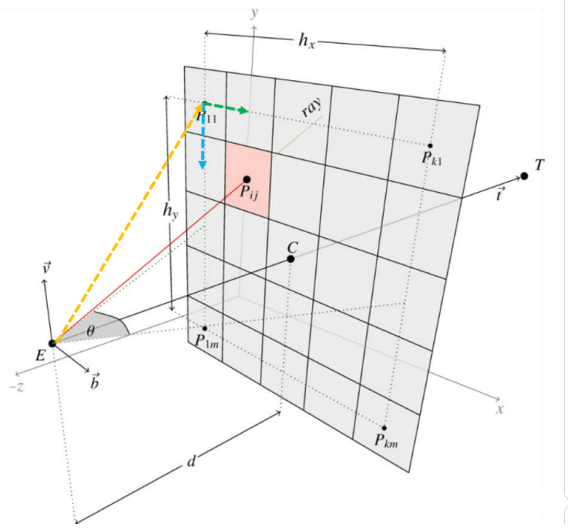


Figure 1: Visual depiction of ray tracing, modified from Wikipedia [10]

2 Methodology

2.1 Ray Tracing

Our primary method for our black hole visualization is by using ray tracing. Ray tracing is a technique for rendering objects on computers commonly used in video games and animated films. In reality, light is emitted from sources, reflects off objects, and bounces into our eyes. Ray tracing mimics this process in reverse. For each pixel on the screen, a different beam of light is shot out, reflects off objects, and bounces towards a light source. The computer then uses the information from the collision and reflection to determine what color to make the pixel on the image.

This is done mathematically in the language of vectors. The camera is defined using only a position vector and angular field of view. Thus the camera becomes a sort of window through which we need to define vectors for each beam of light. This is done by creating pixel sized unit vectors along the rows and columns of the window, and using linear combinations of them to point a vector through any pixel desired. These vectors are then used to propagate the light beam and mathematically detect collisions with objects.

2.2 Light in General Relativity

The primary assumption with this method is that light travels in a straight line. However, in reality, light can curve around massive objects, such as black holes. In these cases, light rays follow the straightest line- geodesics- through curved spacetime. The equations describing geodesics are a set of fourth order ordinary differential equations- these must be used to propagate the light instead of a simple straight line defined by a vector.

Each pixel still needs to be "defined" by the vector used in traditional ray tracing. Instead of defining the line, it must be used as the initial 4-velocity of the light ray, which is used as part of the initial conditions for the ODE solving. The only requirement for the 4-velocity of light is that it is a null vector. Presuming the camera is small enough compared to the curvature of space, we can define the time component of the 4-velocity as only the magnitude of the spatial direction the light is pointing in, resulting in the four-dimensional norm of the vector being zero.

As we've seen in class, there are a variety of meth-

ods of formulating the geodesic equations. We implemented two: the geometric derivation and lagrangian derivation. In both instances, the geodesic equations are the result of derivatives of the metric. As such, code was written which utilized the symbolic toolbox in MATLAB to formulate the equations for any given metric. Both methods were coded, and the performance difference between the two will be commented on later.

2.3 Propagation and Intersections of Light

The four second order ODE's require a total of eight initial conditions for numerical integrations, which come from the initial four-position and four-velocity. The four-velocity was defined above. The four-position is simply the position of the camera. Both of these need to be converted into the correct coordinate system of the metric through use of the Jacobian for the velocity or direct coordinate substitution for the position.

We now have all the pieces to fully propagate the light around the black hole- all that is left is to numerically integrate the equations. This was done with one of the built in ODE solvers in MATLAB. A variety were tested, to varying degrees of success. The primary criteria for our ideal solver was one which provided sufficient resolution to capture the complex motion of the light close to the event horizon while maintaining a relatively low error and computation time.

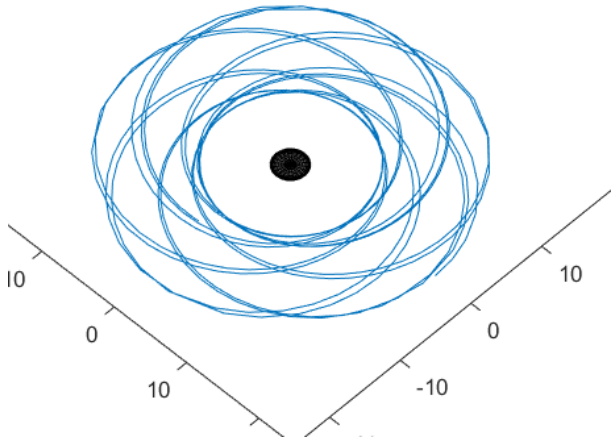


Figure 2: Example trajectory of massive particle, showcasing periapsis shift

In the end, ode23t was chosen. This solver utilizes a second/third order RK optimized for so called stiff systems. Stiffness is an important factor to consider, and unfortunately, it varies depending on the metric and resulting geodesics. Ode23t seemed to be the best compromise.

MATLAB's ode solvers return a value for each of the four spacetime components for each value of the affine path parameter, λ. The final value of λ determined how long the integration ran for, and as a result, directly impacted the performance. As such, it was necessary to choose an optimal final value. Through various tests, it was determined that a value of $\lambda = 100$ gave sufficient time for the light to be influenced by the blackhole and then escape to infinity.

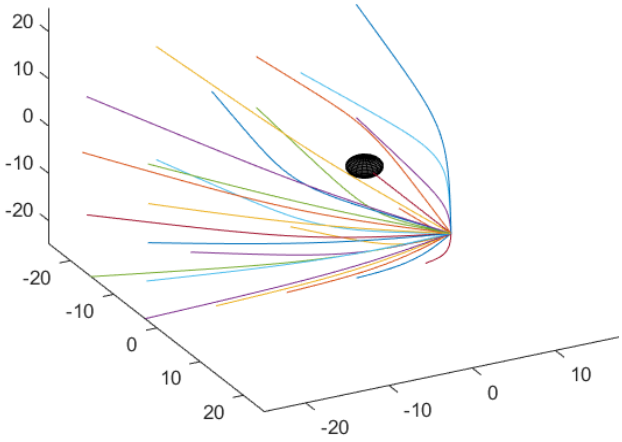


Figure 3: Example trajectories of light rays around a black hole

Once its path was determined, there were three possible interactions for the light beam. It can either fall into the black hole, escape the curvature due to the black hole and travels in a straight line to infinity, or intersect the accretion disk at some point. These interactions were detected as follows:

- To detect falling into the black hole, it was a simple matter of checking if the integrator even made it to the final value of λ, as it would stop itself if it detected the error getting too large (a result of the coordinate singularity at the event horizon). The resulting pixel was made black.
- If the integrator did make it to the max value of λ, then the spherical angles were mapped to a projection of the celestial sphere, and the corresponding pixel used for the pixel on the image.
- To detect if the beam intersected the accretion disk, all that was necessary was to see if and when the beam's trajectory crossed $\theta = \frac{\pi}{2}$ between a prescribed inner and outer radius of the disk.

All of this information was used to determine which color pixel should be.

3 Results

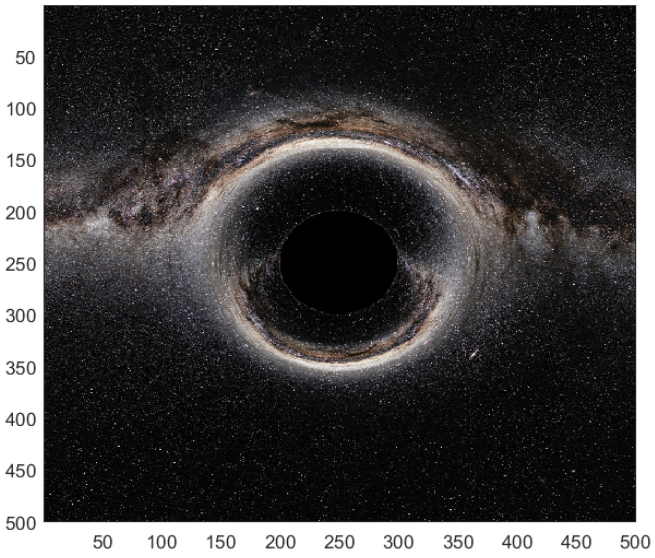


Figure 4: Schwarzschild black hole without accretion disk, showcasing the warping of light

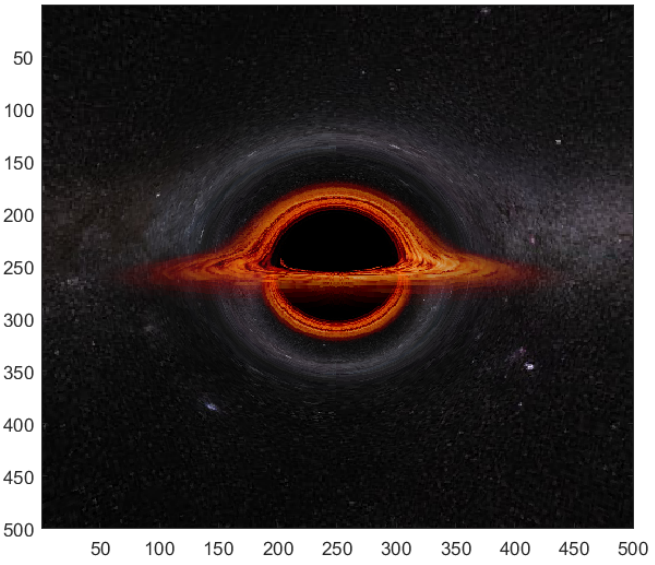


Figure 5: Schwarzschild black hole with accretion disk

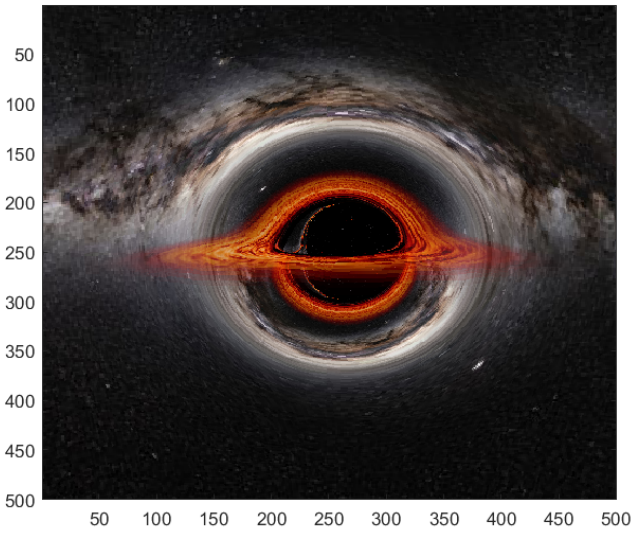


Figure 6: Kerr black hole with accretion disk- notice the more oblong shape and inner (corotating) photon sphere

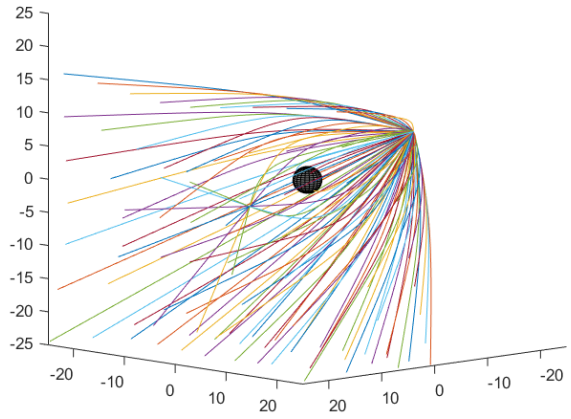


Figure 7: Light trajectories showcasing an Einstein ring- notice a number of light rays converging to a single focal point behind the black hole.

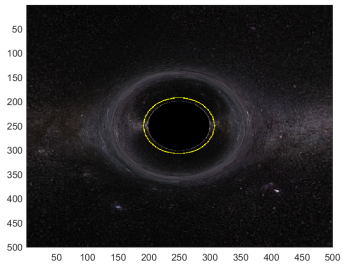


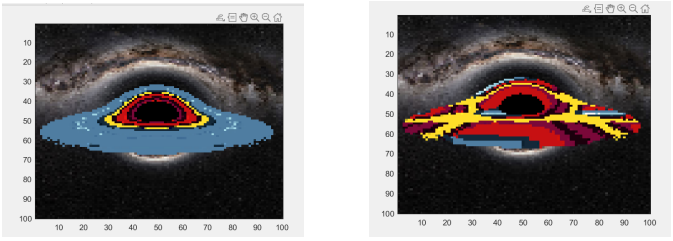
Figure 8: Highlighting the Einstein ring shown in Figure 6

4 Issues

The process was not without its hiccups. Rigorous testing and debugging was necessary before the final result, such as that observed in Figure 9. However, there were also plenty of issues present that were not the result of human error.

One such issue was the computation speed. The shader was by no means fast, but there were some approaches we took to mitigate that. For example, we shifted to using solely using the geodesic equations formulated from the Lagrangian method, as they were much more stable and computed noticeably faster. As well as this, the computation speed was vastly affected by which ODE solver was used. As stated previously, ode23t remained the best compromise between speed, resolution, and stability. However, both of these issues pale in comparison to the largest time save: parallelization. Ray tracing is "trivially parallelizable", meaning that it's entirely possible to run each pixel independently, simultaneously- one pixel does not depend on any other. Through the use of MATLAB's parallel computing toolbox, we were able to leverage multithreaded processing to drastically improve the speed of the code.

Finally, an issue that continues to plague us: the physical interpretation of the metric coordinates. General relativity is notorious for having unintuitive math that is difficult to make sense of physically. As demonstrated in class, there is a difference between the coordinate radius and physical radius in the Schwarzschild metric. This needs to be accounted for in the code. This is only further exacerbated by the Kerr metric, which uses oblate spherical coordinates. As such, in the future it would be necessary to change the code's interpretation of coordinates for each metric.



(a) Broken (b) Fixed

Figure 9: Debugging the accretion disk code using a non radially symmetric test image

5 Further Studies

In the future, with minimal modification, we can incorporate other analytical metrics, such as a more general Weyl-Papapetrou family solver (including binary systems of Kerr black holes), warp metrics (such as the famous Alcubierre and more recent positive-energy warp fields), and broadly any metric with one pair of off-diagonal elements. Optimally for computation time and ease of debugging, we would have a program analytically create formulas for the Christoffel symbols and extract shared variables, such as a, Σ, and Δ in the most common expression of the Kerr metric, a program to convert that to a C#-like shader code (HLSL) giving all the Christoffel symbols in an array at a given x^α position, and proceed with the GPU-based ODE solver we currently have in HLSL WebGL-based online shader software Shadertoy. However more immediately available to us and with lower complexity, is a hybrid analytical-numerical differentiation approach, where the derivatives for the parameters determining a metric are evaluated analytically as far as possible, and then numerically where the complexity becomes too large. This limits the losses due to machine error (a key factor due to the single precision, 32-bit architecture of the GPU) from both finite differences and accumulating error due to formulaic complexity. The derivatives of the metric can then be efficiently evaluated as an array for 4 matrices, and recombined with the inverse metric (which can be computed algebraically for a metric with a single pair of off-diagonal entries, such the Weyl-Papapetrou family) to yield the Christoffel symbols in a near-optimally efficient manner, allowing renders of extremely complex geometries.

The trajectories of charged particles in curved spacetimes, e.g. the Reissner–Nordström metric (charged static black hole), can be computed just as easily as the orbital precession and light ray trajectories above, with similar limitations (e.g. analytic soln.) as above on curvature, by adding an additional $+\frac{e}{m}F^\alpha_\beta \dot{x}^\beta$ term to the RHS of the geodesic equation.

Additionally, we wanted to study the formation of accretion disks around a black hole using post-Newtonian corrected Navier-Stokes equations, but found that adding a dispersive $\mu\nabla^2\vec{u}$ viscosity term to the constant radius solution to the Schwartzchild equation could model radial layer interaction which would lead to the formation of spiral arms. Both could be modeled with the MATLAB PDE toolkit, but the latter could potentially be analytically solved using methods popular in quantum mechanics. If solved, it would be simple to substitute the evaluated time-dependent texturemap in place of our static NASA image for the black hole accretion disk.

6 Contributions

Oscar Haase came up with the idea for the ray tracing in MATLAB, revised the report and the presentation, and wrote the MATLAB ray tracing codes and particle/light ray trajectory tracing codes.
Stanisław Kowalski researched advanced concepts, created GPU shader code, assisted Oscar with debugging, and helped revise report and the presentation. Additionally, Kowalski supplied bananas before the presentation.
Cameron Eure helped with the foundation for the project, created and revised the Powerpoint, created and revised the report, and checked on conceptually and researched topics regarding the project.

Bibliography

[1] Gabriel Gambetta. Basic raytracing, 2022.

[2] Oliver James, Eugénie Tunzelmann, Paul Franklin, and Kip Thorne. Gravitational lensing by spinning black holes in astrophysics, and in the movie interstellar. *Classical and Quantum Gravity*, 32, 03 2015.

[3] O. Jalili K. Mehdizadeh. Charged particles in curved space-time. *J Theor Appl Phys*, 10(47–52), 2016.

[4] V.S. Manko and E. Ruiz. Metric for two equal kerr black holes. *Physical Review D*, 96(10), 2017.

[5] 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, 2019.

[6] Y.J. Segman. Warp drive with positive energy. *Journal of High Energy Physics, Gravitation and Cosmology*, 7(906-913), 2021.

[7] Sholloway. Visualizing black holes with general relativistic ray tracing, Mar 2022.

[8] Frank Grave Thomas Muller. Catalogue of spacetimes. 2010.

[9] Wikipedia contributors. General relativity — Wikipedia, the free encyclopedia, 2022. [Online; accessed 5-December-2022].

[10] Wikipedia contributors. Ray tracing (graphics) — Wikipedia, the free encyclopedia, 2022. [Online; accessed 5-December-2022].

A Appendix

A.1 Geometric Formulation of Geodesics

```
clear
syms lambda t(lambda) r(lambda) theta(lambda) phi(lambda)

G = 1;
M = 1;
c = 1;
r_s = 2*G*M/c^2;

metricName = 'Schwarz'
metric = [-(1-r_s/r) 0 0 0; ...
          0 1/(1-r_s/r) 0 0; ...
          0   0    r^2    0; ...
          0 0 0 r^2*sin(theta)^2]

% OR

% G = 1;
% M = 1;
% c = 1;
% J = 1;
% r_s = 2*G*M/c^2;
% a = J/(M*c);
% sig = r^2 + a^2*cos(theta)^2;
% del = r^2 - r_s*r + a^2;
%
% metricName = 'Kerr'
% metric = [-(1-(r_s*r)/sig) 0 0 -2*r_s*r*a*sin(theta)^2/sig; ...
%           0 sig/del 0 0; ...
%           0 0 sig 0; ...
%           -2*r_s*r*a*sin(theta)^2/sig 0 0 (r^2+a^2+r_s*r*a^2*sin(theta)^2/sig)*sin(theta)^2];
```

Declaration of Metric

```
symbols = cell(4,4,4);

invMetric = inv(metric);
vars = [t, r, theta, phi];

for nu = 1:4
    theseMus = [];
    for mu = 1:4
        theseAlphs = [];
        for alph = 1:4
            thisVal = 0.5*tIndex(invMetric,alph,':') * ...
                (diff(tIndex(metric,':',mu),tIndex(vars,':',nu)) + ...
                diff(tIndex(metric,':',nu),tIndex(vars,':',mu)) - ...
                [diff(tIndex(metric,mu,nu),t); ...
                diff(tIndex(metric,mu,nu),r); ...
                diff(tIndex(metric,mu,nu),theta); ...
                diff(tIndex(metric,mu,nu),phi)]);
            symbols{alph,mu,nu} = thisVal;
        end
    end
end
```

Calculation of Christoffel Symbols

```
geodesics = cell(4);

for mu = 1:4
    summation = 0;
    for alph = 1:4
        for beta = 1:4
            summation = summation + ...
                symbols{mu,alph,beta} * ...
                diff(tIndex(vars,':',alph),lambda) * ...
                diff(tIndex(vars,':',beta),lambda);
        end
    end
    geodesics{mu} = diff(tIndex(vars,':',mu),lambda,2) == -summation;
end

filename = ['geodesics_' metricName '.mat'];
save(filename, 'geodesics', 'r_s', 'G', 'M', 'c');
```

Creation of Geodesics

A.2 Lagrangian Formulation of Geodesics

```
clear
syms lambda t(lambda) r(lambda) theta(lambda) phi(lambda)

% G = 1;
% M = 1;
% c = 1;
% r_s = 2*G*M/c^2;
%
% metricName = 'Schwarz'
% metric = [-(1-r_s/r) 0 0 0; ...
%           0 1/(1-r_s/r) 0 0; ...
%           0   0    r^2    0; ...
%           0 0 0 r^2*sin(theta)^2]

% OR

G = 1;
M = 1;
c = 1;
J = 2;
r_s = 2*G*M/c^2;
a = J/(M*c);
sig = r^2 + a^2*cos(theta)^2;
del = r^2 - r_s*r + a^2;

metricName = 'Kerr'
metric = [-(1-(r_s*r)/sig) 0 0 -r_s*r*a*sin(theta)^2/sig; ...
          0 sig/del 0 0; ...
          0 0 sig 0; ...
          -r_s*r*a*sin(theta)^2/sig 0 0 (r^2+a^2+r_s*r*a^2*sin(theta)^2/sig)*sin(theta)^2]
```

Declaration of Metric (same as Geometric version)

```
syms dt dr dtheta dphi ddt

vars = [t r theta phi];
dvars = diff(vars,lambda);

L = 0.5 * sum(metric .* (dvars.'*dvars),'all')
```

Calculating Lagrangian

```
geodesics = cell(4);

% tIndex is a function I wrote that just allows me
% to index vectors/matrices of symbolic functions
for i = 1:4
    thisVar = tIndex(vars,':',i);
    thisDVar = tIndex(dvars,':',i);
    eqtn = diff(diff(L,thisDVar),lambda) == ...
        diff(L,thisVar);
    geodesics{i} = symfun(isolate(eqtn,diff(thisDVar,lambda)),lambda);
end

filename = ['geodesics_lagrange_' metricName '.mat'];
save(filename, 'geodesics', 'r_s', 'G', 'M', 'c');
```

Calculating Geodesics

A.3 Appendix C: Ray Tracing Code

```
clear

% Camera specifications- leaving out time component for now...
camPos = [25 0 3];
targetPos = [0 0 0];

FOV = pi/2;
verticalPixels = 500;
horizontalPixels = 500;

%Formulate vectors used for camera transformation
vertical = [0 0 1];
camDir = targetPos - camPos;
horizontal = cross(vertical,camDir);

unitCamDir = camDir./norm(camDir);
unitHorizontal = horizontal./norm(horizontal);
unitVertical = cross(unitCamDir,unitHorizontal);

outerPixelHorzDist = tan(FOV/2);
outerPixelVertDist = outerPixelHorzDist*(verticalPixels-1)/(horizontalPixels-1);

horizontalStep = 2*outerPixelHorzDist/(horizontalPixels-1) .* unitHorizontal;
verticalStep = 2*outerPixelVertDist/(verticalPixels-1) .* unitVertical;

ray_11 = unitCamDir - outerPixelHorzDist.*unitHorizontal + outerPixelVertDist.*unitVertical;

%Define system of diffeqs
metricName = 'Schwarz';
% OR
% metricName = 'Kerr';

% load(['geodesics_' metricName '.mat']);
% OR
load(['geodesics_lagrange_' metricName '.mat'])

eq1 = geodesics{1};
eq2 = geodesics{2};
eq3 = geodesics{3};
eq4 = geodesics{4};

eqs = [eq1; eq2; eq3; eq4];

[V,S] = odeToVectorField(eqs);

geoFunc = matlabFunction(V,'vars',{'lambda','Y'});

%Drawing the screen
viewPort = zeros(verticalPixels*horizontalPixels,3);
diskPic = imread('diskTexture_2.png');
background = imread('stars.jpg');
```

```
dimsBG = size(background);
dimsDisk = size(diskPic);

lambda_f = 100;

diskRadius = r_s + 15;

parfor pixel = 1:horizontalPixels*verticalPixels
    warning('off','MATLAB:ode23t:IntegrationTolNotMet');
    i = mod(pixel-1,horizontalPixels)+1;
    j = ceil(pixel/verticalPixels);

    %Ray direction
    thisRay = ray_11 + horizontalStep*j - verticalStep*i;

    thisRaySph = cartToSphereVel(thisRay,camPos);
    camPosSph = cartToSpherePos(camPos);

    timeComp = norm(thisRay);

    %Define initial conditions based on camera pos and ray dir
    initCond = [camPosSph(1); thisRaySph(1); 0; timeComp; ...
        camPosSph(2); thisRaySph(2); camPosSph(3); thisRaySph(3)];

    soln = ode23t(geoFunc,[0 lambda_f], initCond);

    theseRadii = soln.y(1,:);
    radMask = (theseRadii < diskRadius) & (theseRadii > (r_s+1));

    theseThetas = soln.y(5,:);
    delThetas = [0 theseThetas-pi/2] .* [theseThetas-pi/2 0];
    thetaMask = delThetas < 0;

    thesePhis = soln.y(7,:);

    accretionMask = radMask & thetaMask(2:end);

    if soln.x(end) < (lambda_f - 10)
        backColor = zeros(1,1,3);
    else
        thisTheta = soln.y(5,end);
        thisPhi = soln.y(7,end);

        raw_u = thisTheta/pi;
        raw_v = 1 - (thisPhi/(2*pi)+0.5);

        u = mod(floor(raw_u*dimsBG(1)),dimsBG(1))+1;
        v = mod(floor(raw_v*dimsBG(2)),dimsBG(2))+1;

        backColor = double(background(u,v,:));
    end

    if any(accretionMask)
        rIntersections = theseRadii(accretionMask);
        thisR = rIntersections(1);
        phiIntersections = thesePhis(accretionMask);
        thisPhi = phiIntersections(1);

        normalizedR = (thisR/diskRadius);

        raw_u = normalizedR/2*cos(thisPhi)+0.5;
        raw_v = normalizedR/2*sin(thisPhi)+0.5;

        u = mod(floor(raw_u*dimsDisk(1)),dimsDisk(1))+1;
        v = mod(floor(raw_v*dimsDisk(2)),dimsDisk(2))+1;

        accretionColor = double(diskPic(u,v,:));

        opacity = exp(1-1/(1-normalizedR^2));

        if sum(accretionMask) > 1
            thisR_2 = rIntersections(2);
```

```
            thisPhi_2 = phiIntersections(2);

            normalizedR_2 = (thisR_2/diskRadius);

            raw_u_2 = normalizedR_2/2*cos(thisPhi_2)+0.5;
            raw_v_2 = normalizedR_2/2*sin(thisPhi_2)+0.5;

            u_2 = mod(floor(raw_u_2*dimsDisk(1)),dimsDisk(1))+1;
            v_2 = mod(floor(raw_v_2*dimsDisk(2)),dimsDisk(2))+1;

            accretionColor_2 = double(diskPic(u_2,v_2,:));

            opacity_2 = exp(1-1/(1-normalizedR_2^2));

            thisColor_2 = accretionColor_2*opacity_2 + backColor*(1-opacity_2);
            thisColor = accretionColor*opacity + thisColor_2*(1-opacity);
        else
            thisColor = accretionColor*opacity + backColor*(1-opacity);
        end
    else
        thisColor = backColor;
    end

    viewPort(pixel,:) = thisColor;
end

viewPort = reshape(viewPort,horizontalPixels,verticalPixels,3);
image(uint8(viewPort));

function [spherical] = cartToSphereVel(cart,pos)
    x = pos(1);
    y = pos(2);
    z = pos(3);
    r = sqrt(x^2+y^2+z^2);
    J = [x/r y/r z/r; ...
        x*z/(r^2*sqrt(x^2+y^2)) y*z/(r^2*sqrt(x^2+y^2)) -(x^2+y^2)/(r^2*sqrt(x^2+y^2)); ...
        -y/(x^2+y^2) x/(x^2+y^2) 0];
    spherical = J*cart.';
    spherical = spherical.';
end

function [spherical] = cartToSpherePos(cart)
    x = cart(1);
    y = cart(2);
    z = cart(3);
    r = sqrt(x^2+y^2+z^2);
    theta = acos(z/r);
    if x > 0
        phi = atan(y/x);
    elseif x < 0 && y >= 0
        phi = atan(y/x) + pi;
    elseif x < 0 && y < 0
        phi = atan(y/x) - pi;
    elseif x == 0 && y > 0
        phi = pi/2;
    elseif x == 0 && y < 0
        phi = -pi/2;
    elseif x == 0 && y == 0
        phi = 0;
    end
    spherical = [r theta phi];
end
```

If you made it this far thanks for reading