Approximate Black Hole Renderings with Ray Tracing

Mack Qian
Rensselaer Polytechnic Institute
qianm@rpi.edu

May 2021

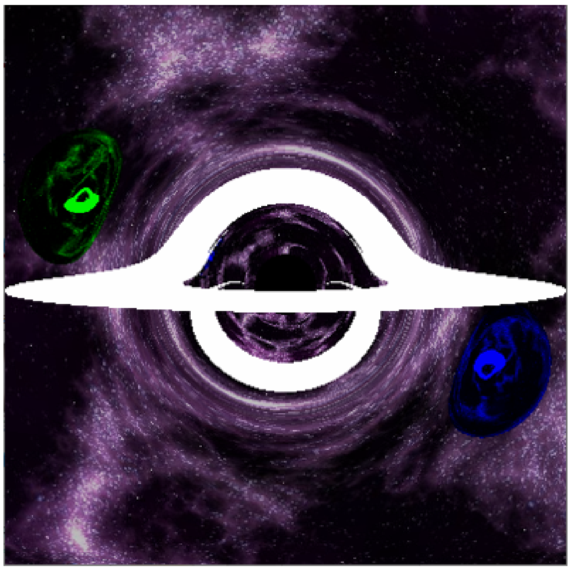

Figure 1: A complicated black hole scene, rendered with a disk, 2 reflective spheres, and 4x antialiasing

Abstract

In this paper, we describe a method to create approximate black hole renders based off of ray tracing. Rather than ray tracing with linear paths, we use photons and apply Newtonian gravitational forces to it, causing them to travel in non-linear and curved paths. This method is able to simulate predicted behaviors such as gravitational lensing of objects behind the black hole or the image distortion of an accretion disk around the black hole. This method is able to also support some common ray tracing visuals, such as antialiasing and recursive reflections.

1 Introduction

Ray tracing is usually done with straight and linear light paths, as expected from light in regular and normal scenes. However, there are cases where light can bend and travel in non-linear paths, such as when they are bent and lensed by gravity from galaxies over vast distances of space or by immense gravity wells cause by black holes [4]. Physically accurate simulations require a fair amount of complicated math and physics knowledge, but visually acceptable models can use Newtonian gravity to approximate light rays. Doing so can vastly simplify calculations while still being able to simulate some of the predicted behaviors somewhat accurately.

2 Related Work and Resources

Previous works on rendering black holes include work done for *Interstellar* [4], which describes methods for rendering spinning (Kerr) black holes. They use a modified ray tracing algorithm with ray bundles and Kerr metric to create a tool called DGNR (Double Negative Gravitational Renderer) to create the renderings used in the movie. Another earlier paper [5] created a rendering of a Schwarzchild (non-spinning) black hole with a thin accretion disk along with other effects such as Doppler shifting. There have also been a number of projects that we used as a reference for this project. This includes 2D visualization of black holes created by Daniel Schiffman [8], which provided the 2D basis for the photon path algorithms and Chris Orban's relativistic correction [6]. Ricardo Antonelli's "Starless" [2], which provided helpful visuals for helping to evaluate the accuracy of our algorithms.

The project used Homework 3 code as its basis. We also used Alex Peterson's Spacescape [7] and OpenCV [3] to generate the space scenes used in a few of the visualizations.

3 Implementation

We first create a new `BlackHole` class. This class takes in a mass and position, then computes the radius based off of the following equation:

$$r = \sqrt{\frac{2GM}{c^2}} \tag{1}$$

Where G is the gravitational constant, c is the speed of light, and M is the mass of the black hole. This equation is based off of the Schwarzchild metric used for calculations on non-spinning black holes. This is the approach used by Schiffman [8]. This radius denotes the event horizon of the black hole, where once light enters,it cannot escape back out. However the actual image of the black hole will appear about 2.5 times larger than this radius [2] as low incoming light rays get pulled in and "absorbed". This computed radius will then help determine when a light ray should continue on its path or stop if it is within the sphere defined by the radius. One major assumption is that we make G and c arbitrary values of 1 and 10. We do this as it keeps the scale of the simulation, as using the real values of G and c (6.674×10^{-11} and $299,792,458$ [9]) would cause the simulation to scale to very large values, which would be very impractical and inefficient. These two arguments along with a time step dt can be specified within the command line arguments or left to the default values. With those constants fixed, the given mass then controls the black hole radius, with bigger masses giving a bigger and stronger black hole. The black hole mass and position are added via the scene .OBJ files with the tag `bh`, which then the file parser reads and adds to a list of all black holes. This list allows support for multiple black holes in the scene, as they all will factor into the path computations.

3.1 Path Algorithm

We use the following algorithm to replace the basic ray cast intersection code in the `TraceRay` algorithm in the homework:

```
def bhRayTrace(Ray r,Hit h):
    pos=r.origin
    vel=r.dir*c
    for 10K iterations:
        temp=pos
        k1v,k1a=step(pos,vel)
        k2v,k2a=step(pos+dt*0.5*k1v
            ,vel+dt*0.5*k1a)
        k3v,k3a=step(pos+dt*0.5*k2v
            ,vel+dt*0.5*k2a)
        k4v,k4a=step(pos+dt*k3v
            ,vel+dt*k3a)
        pos += (1/6.0)*dt
            *(k1v+2*k2v+2*k3v+k4v)
        reScaleV(vel
            ,(1/6.0)*dt
                *(k1a+2*k2a+2*k3a+k4a))
        for b in blackholes:
            dist=(pos-center).length()
                if dist<b.r:
                        return absorbed
        Ray tracer=Ray(temp,pos-temp)
        addSegment(tracer,0,1)
        intersection=newIntsct(pos,vel,h)
        if intersection found:
            return intersection
    return fail
```

In order to begin computing the path, we first create a photon from the ray given in main the ray tracing algorithm. This ray can be either from the main camera view or recursive rays like those from reflections or refractions. The origin and direction are used to initialize the photon's position and velocity. For the velocity, we just multiply it by c since the direction is already a unit vector. Then the algorithm loops over a set number of iterations, which we define to be 10K. This is to put a hard limit so photons eventually stop and don't continue to infinity. We then use the following step algorithm to compute a new acceleration and velocity:

```
def step(pos,vel):
    acc=vec3(0,0,0)
    for each b in blackholes:
        r=vec3(b.center-pos)
        r_s=r.length()
        r.Normalize()
        temp=((G*b.M)/r_s^2)*r
        fac=temp.Dot3(vel)
        fac/=vel.length()
        parallel=vel
        parallel.normalize()
        parallel.scale(fac)
        acc+=temp-parallel
    reScale(vel,dt*acc)
    return vel,acc
```

We compute the acceleration due to all black holes in the scene. To do this, we iterate over all black holes, and first use the vector form of the Newtonian gravity equation to compute the acceleration:

$$\mathbf{F} = \frac{GMm}{||\mathbf{r}||^2}\hat{\mathbf{r}} \tag{2}$$

Where M is the mass of the black hole, $\mathbf{r}$ is the vector from the position of the photon to the center of the black hole, and $\hat{\mathbf{r}}$ is the unit vector in the direction of $\mathbf{r}$. We make the (unphysical) assumption that the photon's mass is 1, so then the force also equals the acceleration acting on the photon. We then apply a relativistic correction described by Orban [6]. If we did not apply this correction, then our photons will be able to go faster than the speed of light, which leads to a lot more rays being sucked into the black hole. The correction is to essentially only lets the acceleration change the direction of the velocity by rotating the velocity by then angle between it and the acceleration. To adapt the 2D case provided by Orban to 3D, we do the following vector computations:

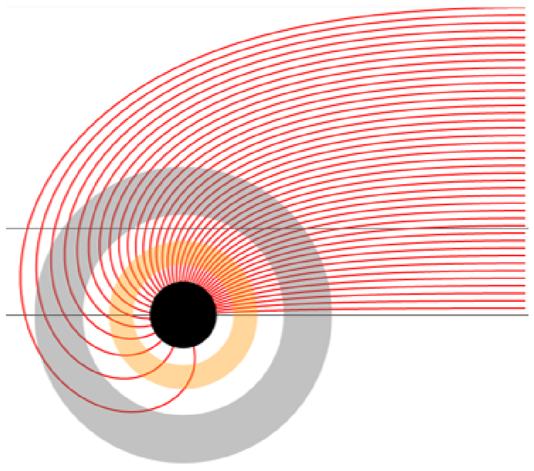
Figure 2: A 2D example of using just Newton's Equation with no correction [8]

$$\mathbf{a}_{||} = \frac{\mathbf{a}\cdot\mathbf{v}}{||\mathbf{v}||}\hat{\mathbf{v}}$$

$$\mathbf{a}_{\perp} = \mathbf{a} - \mathbf{a}_{||}$$

Where $\mathbf{a}$ is the computed acceleration, $\mathbf{v}$ is the current velocity of the photon, $\hat{\mathbf{v}}$ is the unit vector of the

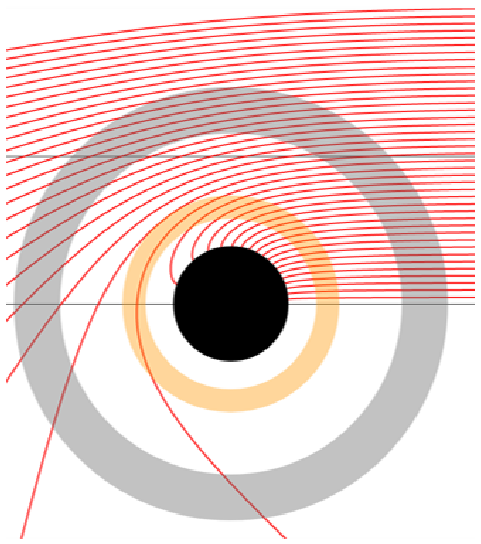
Figure 3: A 2D example of using Newton's Equation with correction [8]

velocity, and $\mathbf{a}_\perp$ is the component of the acceleration perpendicular to the velocity. We find the parallel component by projecting the acceleration onto the velocity, then subtracting it from the total acceleration to get the perpendicular component. Then to actually modify the velocity, we simply just add the computed acceleration to the vector and rescale the result back to c. Adding with out rescaling will still make the velocity increase in magnitude, so the rescaling is necessary to preserve the speed limit.

These step calculations help us use Runge Kutta in order to get more accurate computations with larger timesteps. We average the steps using the the way described by RK4 and update the position and velocity. As before with updating the velocity, we add and rescale to ensure the magnitude stays at c.

After the velocity and position updates, we check whether or not the photon position has entered the event horizon of any of the black holes in the scene. This is simply done by checking whether or not the distance between the photon and the blackholes' centers are below the computed radius. If it is, we return a flag saying it was absorbed. This flag then signals to the `TraceRay` algorithm to simply return the color black. If it passes the distance checks, it will move on to checking for intersections, which we will detail in the next subsection. If an intersection is found, then it will return a flag denoting that. Then the `TraceRay` algorithm will continue based off of the material stored in `Hit h`, whether it be hitting a light or diffuse surface and returning a color, or hitting a reflective surface and recursing further. If all iterations execute and the photon was not absorbed or no intersection was found, then we return a failure flag. This signals to the main ray trace algorithm to simply return the color magenta. This was very helpful in debugging, though it does force scenes to be enclosed in a skybox in order to function properly (though one could just return the background color and it would act just like the original ray tracing algorithm). While helpful for debugging, it can cause issues if the timestep is very small, as then for each iteration, the photon will barely move. Bigger iteration caps are needed to use small time steps, but for our purposes, 10K iterations were sufficient for timesteps greater than 0.01. Also the use of RK4 mitigates the need for using very small timesteps, having 0.1-0.5 giving good results. The use of `tracer` was for helping to visualize the paths taken by the photon, allowing us to debug the path and intersection algorithms a lot easier. It does abuse the `Ray` class a bit as normally it expects an origin and a unit vector for the direction. However, by passing it the previous position and the from it to the next computed position, then using 0 and 1 as the starting and ending parameters in `addSegment`, we are able to create segments that extend exactly from the old position to the new one.

3.2 Intersection Computation

With this new ray tracing algorithm, we created a new intersection algorithm for checking intersections with primitives. It is as follows:

```
def newIntsct(pos, vel, hit):
    vel.normalize()
    Ray r= Ray(pos, vel)
    insct=rayCast(r,h)
    if insct:
        ipos=h.point
        dist=(ipos-pos).length()
        if dist<dt*c+EPSILON:
            return intersection
        else if dist <2*(dt*c+EPSILON)
            and count<2:
                dt*=0.1
                count++
    return no intersection
```

We first initialize a `Ray` based off of the current position and velocity of the photon, making sure to normalize the velocity as it should be a unit vector. Then we run the original raycasting intersection algorithm to see if something is intersected. If the scene is an enclosed box, then this should always return a hit. With this hit, we check whether or not this hit is close enough to the current position of the photon. We quantify "close enough" by noticing that at maximum, the photon will travel the distance $dt * c$. Then the worst case for this photon is that its velocity is perpendicular to the primitive the point is intersecting. If it is above this threshold, then it can still at least travel one time step's worth of velocity with out passing the intersecting primitive. If is below, then we return the flag for an intersection, as if it were to continue, it would phase through the intersecting solid. In practice, this threshold needed an EPSILON term as constantly adding and updating the position leads to some floating point errors. As for when the the distance is above the threshold, we run an additional check to see if the photon is somewhat close to the intersecting primitive. If it is, we shrink the time step to 10% the original amount and increment a count variable. This helps us speed up the computations a bit by allowing us to use higher timesteps but still preserving the intersection algorithm. The count variable prevents us from infinitely shrinking the timestep, as then it will never intersect anything. This solution works suprisingly well, though it is a bit inefficient. However, as long as a normal ray cast intersection algorithm can be defined for a primitive, then this new algorithm will work as well. There might be potential issues if the object is concave though, but for our purpose, where we only deal with planes, cylinder rings, and spheres, it works well.

4 Results

We believe or method works relatively well for being an Newtonian approximation. Most figures will be put at the end of the document.

4.1 Accuracy

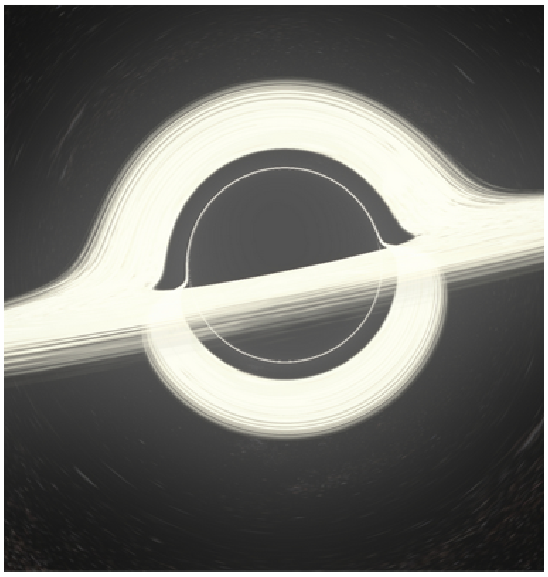

Figure 4: One of Antonelli's Renders [2]

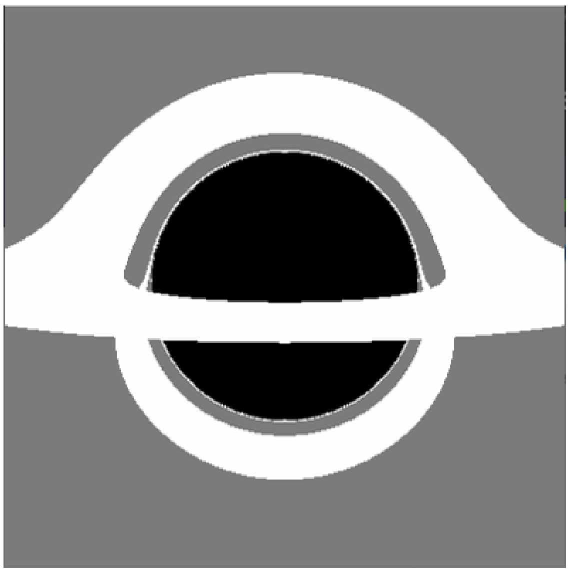

Figure 5: Our Renders

We compare our images to Antonelli's black hole renders and get similar results, shapewise. His differs from our by also including transparency in the ring and having the ring be off axis.

4.2 Performance

The performance is alright. Most black hole scenes take around 1-5 minutes to render, with more complicated scenes with reflections and antialiasing taking upwards of 10 minutes. Performance can be increased by using a higher time step, but will cause issues with the accuracy of the renders.

5 Limitations and Future Work

5.1 Lighting

One of the biggest limitations for this method is the lack of support for global and local illumination. Most of the figures rendered used a bright ambient light to help with the lighting, with the exception of a few of the "color box" scenes. The shadow checking algorithm used in normal ray tracing will not work. If we were to start at a point and cast toward the centroid of a light source, there could be cases where this ray ends up being occluded by being bent into the black hole or another object. However, light from the light source could be bent by the black hole and be able to hit the point we started at. There is also the problem that centroids in regards to lighting are not defined for some primitives, like spheres and rings. A potential direction to go in is to look at implementing forward ray tracing, as then we could accurately simulate shadows. Another potentially easier route for slightly better results is to look at photon mapping(not what we were doing) and use the modified ray tracing algorithm to affect the paths the photons take. This would still need some definitions for generating photons for some primitives, but it would be able to generate caustics by gathering the photons. For example, if circular light source were directly above a black hole, then a ring or a circular caustic could be formed depending on the scene.

5.2 Performance

Another limitation is the performance. While manageable for this project, it would handle very complicated scenes poorly. Scenes with many black holes would slow the performance of the path calculations, while scenes with many objects will considerably slow down the program, as the intersection runs for each iteration of the path computation. The expensive intersection algorithm could be solved by implementing spatial data structures, as then it would allow us to do less intersection checks. The intersection algorithm itself has some issues, as it had a tendency to phase through objects quite a lot, so a more robust and faster algorithm would definitely be an improvement. And generally for performance, multithreading the ray tracing operations would speed things up. This would be relatively easy to do, as each ray trace is independent of one other, so concurrency would not run into many issues, with the exception of perhaps something like the path segments for debugging (but one would presumably want to use it to debug on a non-multithreaded program first for accuracy). There could also be the option to use the GPU instead, though that would be difficult to do since it would probably require doing a lot more things from scratch.

6 Miscellaneous

6.1 Sphere of Influence

Originally, we had planned to use a Sphere of Influence approach, similar to that used by Yapo in his rendering of lunar eclipses [10]. We had it so that normal ray tracing was run until it intersected a predefined sphere of influence, then would get affected by the gravitational field. This ended up looking bad as there was a quite obvious circle of distortion, which could be alleviated by making the sphere bigger, but at that point, it would make more sense to just have the rays be bent from the beginning.

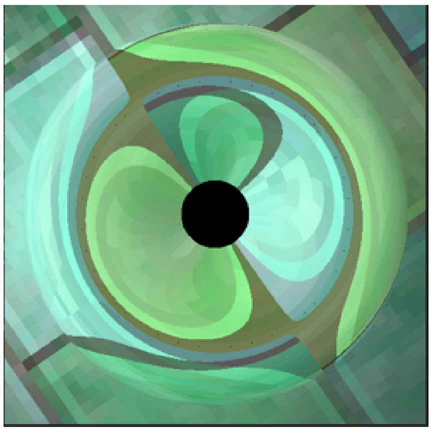

Figure 6: One of the early renders with SOI

6.2 Intersection Problems

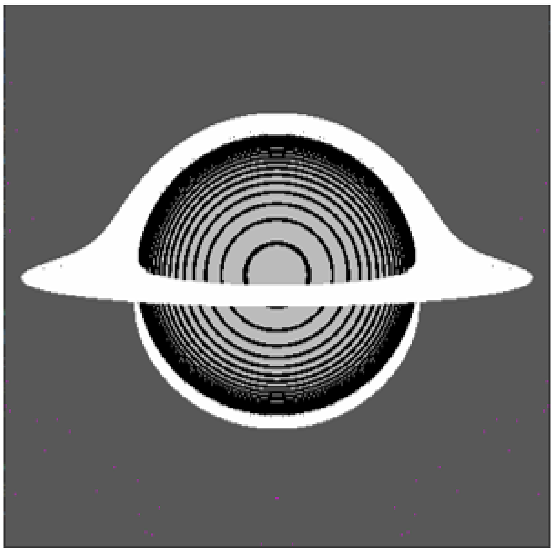

Figure 7: Buggy intersection algorithm render

The intersection algorithm was somewhat problematic, as often it would just phase right through the objects. We were able to solve this by using an adaptive timestep when the photons approached the intersecting objects. This was rather challenging to debug and is still not perfect. This figure showcases some of the buggy output with the debug magenta pixels mentioned in the path calculation algorithm. The gray pixels on the black hole are rays that shot through the black hole and intersected the back wall.

6.3 “White” Hole

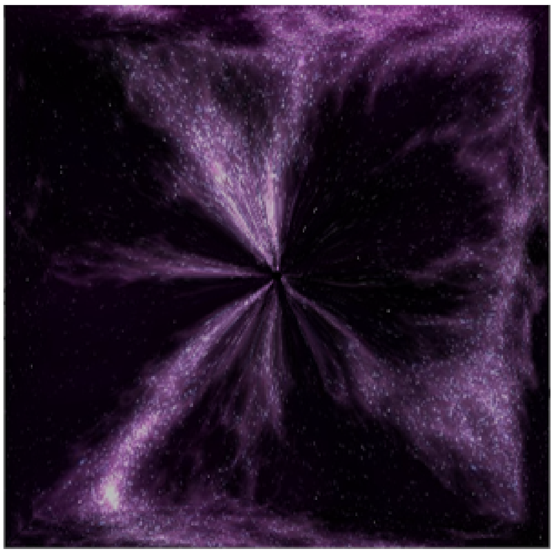

Figure 8: a “White Hole”

The figure was just generated as a novelty. Rather than having the photons get pulled in by the black hole, it is instead pushed away. This is simply done in code by negating the acceleration. It produces a rather interesting result.

References

[1] White blue graph paper grid. `https://www.xmple.com/wallpaper/white-blue-graph-paper-grid-512x512-c2-ffffff-4169e`

[2] Riccardo Antonelli. 2d black hole visualization. `https://rantonels.github.io/starless/`, 2015.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb’s Journal of Software Tools*, 2000.

[4] Oliver James, Eugenie Tunzelmann, Paul Franklin, and Kip Thorne. Gravitational lensing by spinning black holes in astrophysics, and in the movie interstellar. *Classical and Quantum Gravity*, 32, 02 2015.

[5] Jean-Pierre Luminet. Image of a spherical black hole with thin accretion disk. *Astronomy and Astrophysics*, 75:228–235, 04 1979.

[6] Chris Orban. Black hole derivation. https://www.asc.ohio-state.edu/orban.14/stemcoding/blackhole_derivation_slide1.png, 2019.

[7] Alex Peterson. Spacescape. https://alexcpeterson.com/spacescape/.

[8] Daniel Shiffman. 2d black hole visualization. https://thecodingtrain.com/CodingChallenges/144-black-hole-visualization, 2019.

[9] Eite Tiesinga, Peter J. Mohr, David B. Newell, and Barry N. Taylor. The 2018 codata recommended values of the fundamental physical constants. http://physics.nist.gov/constants.

[10] Theodore Yapo and Barbara Cutler. Rendering lunar eclipses. 2009.

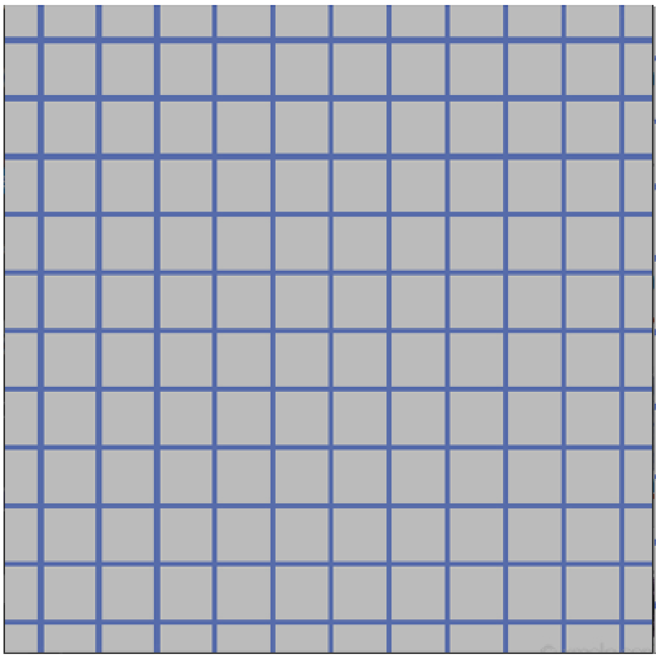

Figure 9: No black hole [1]

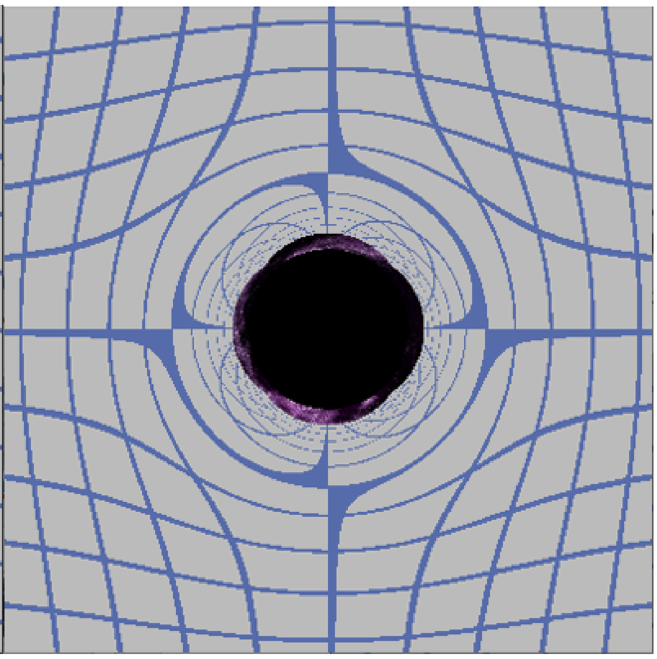

Figure 10: With black hole

Figure 11: Images showcasing the distortion

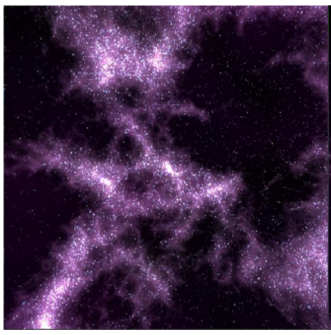

Figure 12: No Blackhole

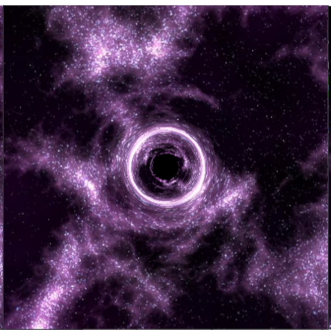

Figure 13: M=100

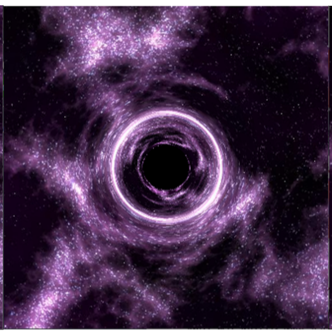

Figure 14: M=200

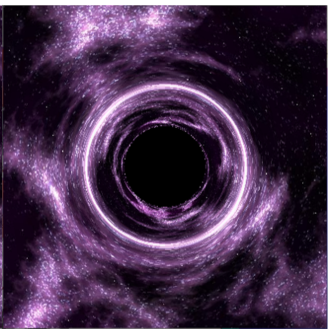

Figure 15: M=400

Figure 16: The generated space box with varying masses of black holes

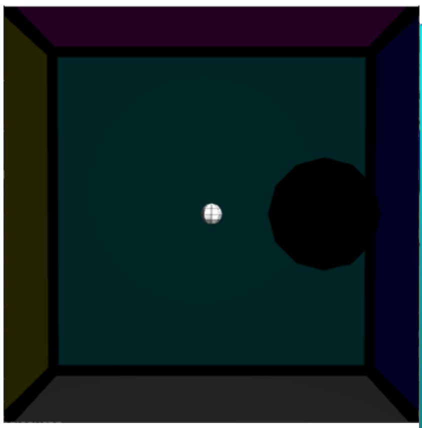

Figure 17: Primitive View (from the side)

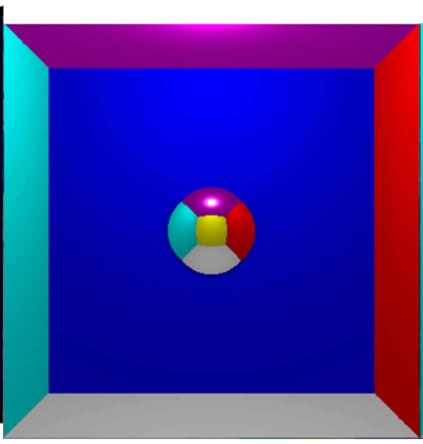

Figure 18: No black hole

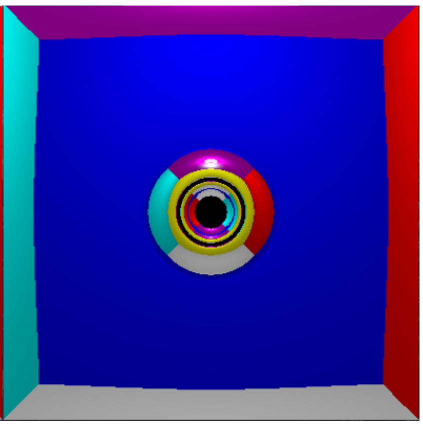

Figure 19: With black hole

Figure 20: Images showcasing reflections

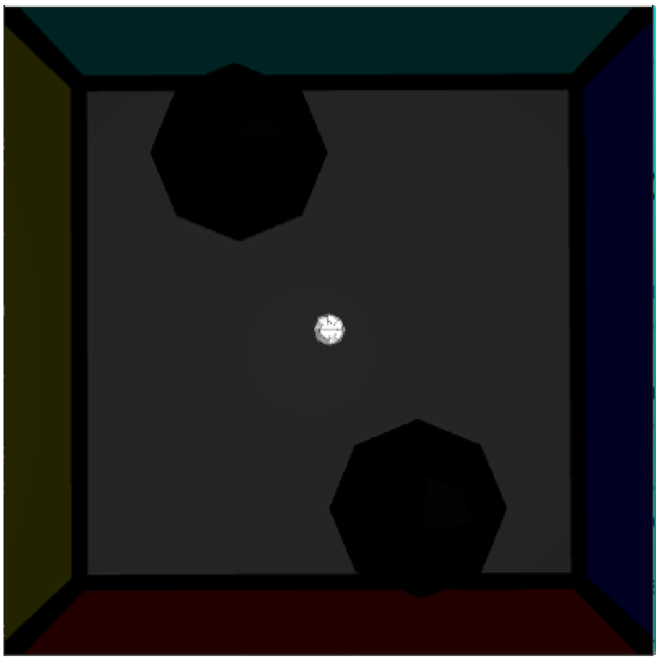
Figure 21: Primitive view, top down

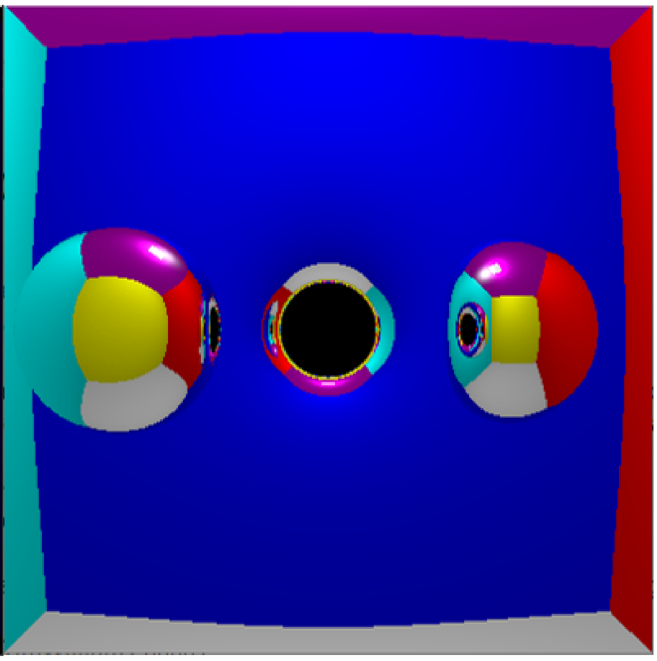
Figure 22: With black hole

Figure 23: Images showcasing multiple reflections

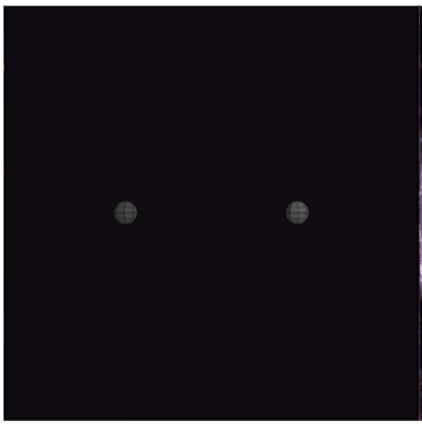
Figure 24: Primitives view

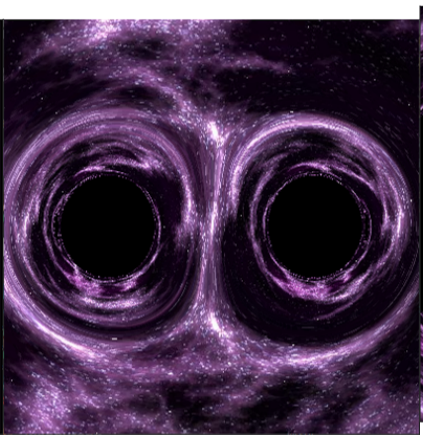
Figure 25: Ray Traced

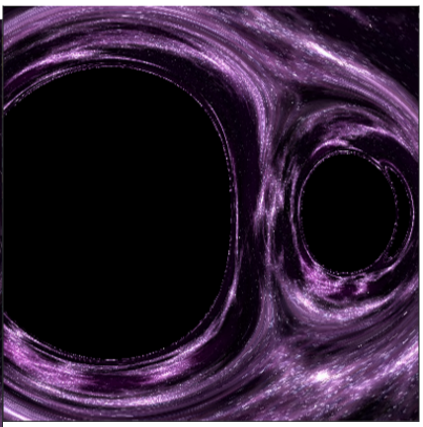
Figure 26: Same positions, but left one has double the mass

Figure 27: Mutiple black holes

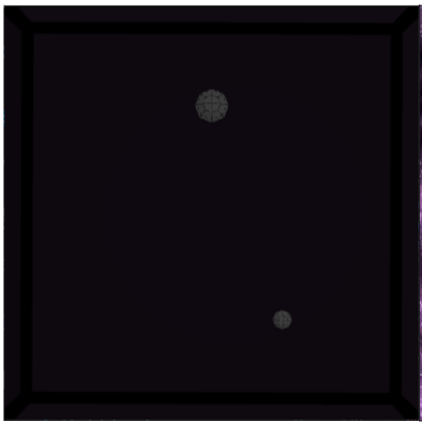
Figure 28: Primitive view, top down

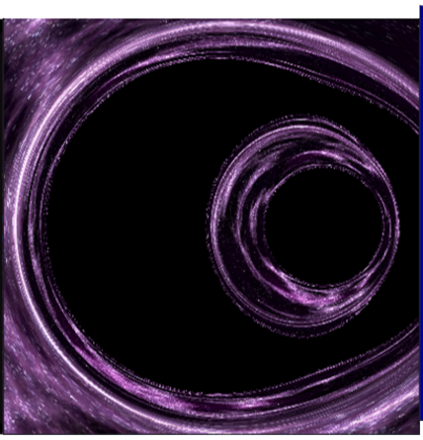
Figure 29: With black hole

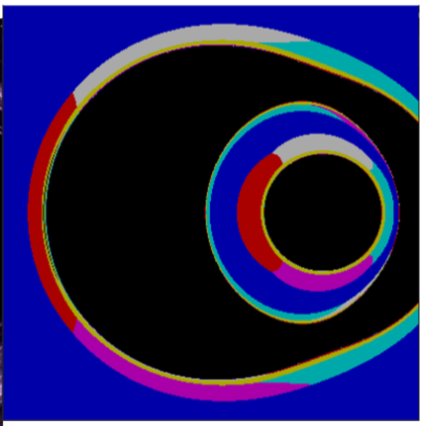
Figure 30: With black hole and color box

Figure 31: Images showcasing multiple black holes offset on 2 axes

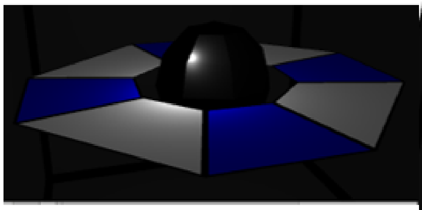
Figure 32: Primitive view, top of disk

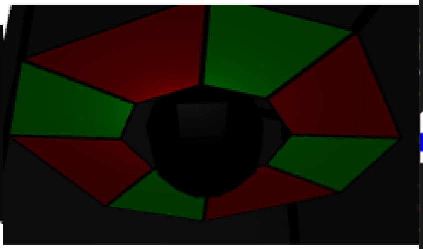
Figure 33: Primitive view, bottom of disk

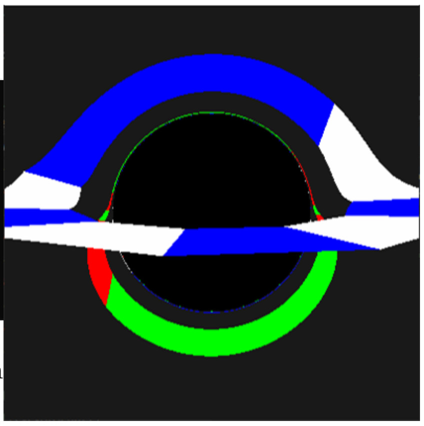
Figure 34: Render

Figure 35: Images showcasing accretion disk distortion

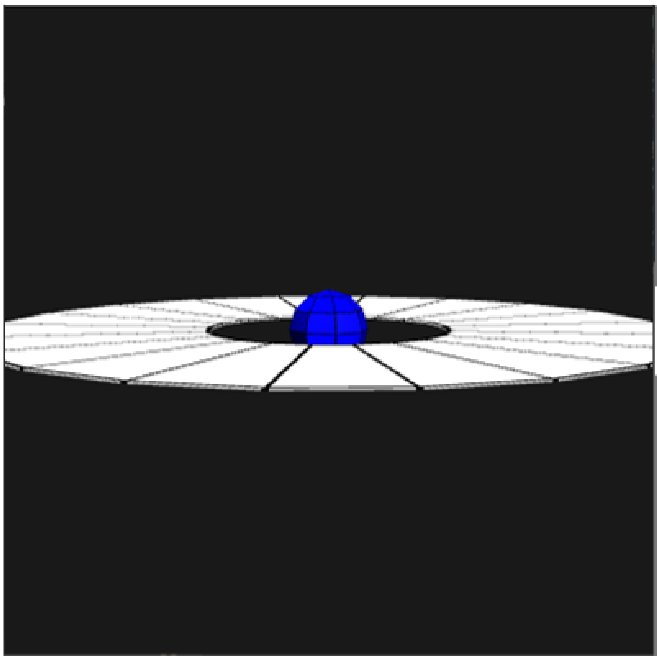
Figure 36: Primitive view

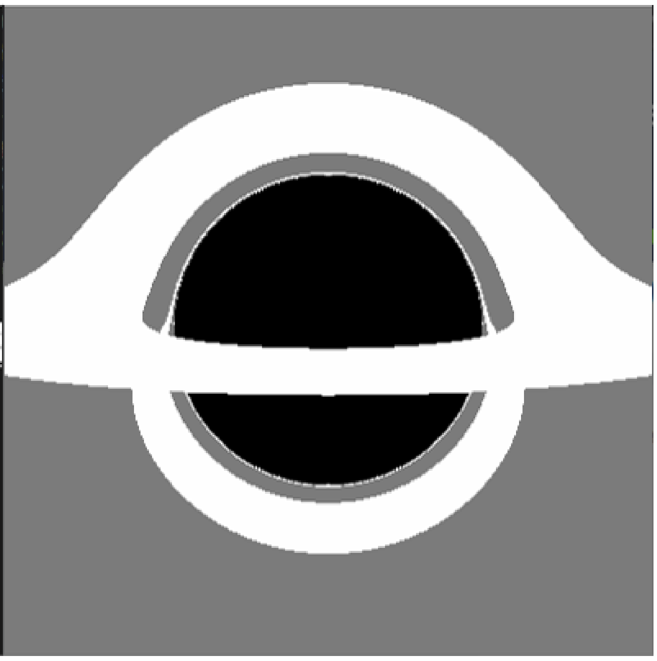
Figure 37: Render

Figure 38: Images showcasing accretion disk distortion with a more realistic disk