Black Hole Simulation: Math & Physics Guide

A step-by-step guide to understanding and implementing the mathematics behind gravitational lensing.

Table of Contents

1. Core Concepts
2. Coordinate Systems
3. The Geodesic Equation
4. Numerical Integration (RK4)
5. Implementation Roadmap

Core Concepts

What Are We Actually Simulating?

Black holes don't "pull" on light like gravity pulls on a ball. Instead, they **curve spacetime itself**. Light always travels in straight lines, but "straight" means different things in curved space.

Analogy: Imagine drawing a straight line on a flat paper vs. on a sphere. On Earth, airplanes fly in "straight lines" (geodesics), but on a flat map, these paths look curved. That's exactly what happens with light near a black hole.

Key Physics Terms

Term	What It Means	Why It Matters
Geodesic	The straightest possible path in curved space	Light follows geodesics through curved spacetime
Schwarzschild Metric	A specific solution describing spacetime around a non-rotating black hole	Gives us the "shape" of curved space we need to navigate
Affine Parameter (λ)	An arbitrary "step size" for moving along the path	Like time, but works even for light (which doesn't experience time)
Schwarzschild Radius (Rs)	The radius of the event horizon	$R_s = (2GM)/c^2$ — the point of no return

Coordinate Systems

Why Polar Coordinates?

Near a black hole, everything is **radially symmetric** — the black hole looks the same from every angle. Polar coordinates naturally fit this symmetry.

Cartesian (x, y) → Your Screen

```
y
↑
|
|
+----→ x
```

Good for: Rendering graphics, drawing circles, positioning objects on screen

Bad for: Calculating gravity effects (black hole doesn't care about "left/right", only "how far away")

Polar (r, θ) → Black Hole's Perspective

```
  θ = angle
   ↗
  /
 /
• ← Black hole at origin
 \
  \__ r = distance
```

Good for: Physics calculations, gravity is based purely on distance `r`

Bad for: Drawing on screen (OpenGL wants x, y coordinates)

Conversion Formulas

You'll need these constantly:

Cartesian → Polar:

```
r = √((x - x_bh)² + (y - y_bh)²)
θ = atan2(y - y_bh, x - x_bh)
```

Where `(x_bh, y_bh)` is the black hole's position.

Polar → Cartesian:

```
x = x_bh + r * cos(θ)
y = y_bh + r * sin(θ)
```

Implementation Tip

Create these as helper functions early:

```
// In your code:
glm::vec2 cartesianToPolar(glm::vec2 pos, glm::vec2 blackHolePos);
glm::vec2 polarToCartesian(float r, float theta, glm::vec2 blackHolePos);
```

The Geodesic Equation

This is the heart of the simulation. It tells us **how the direction of light changes** as it moves through curved spacetime.

The Big Picture

Newton would say: "Gravity accelerates objects"

- F = ma → acceleration changes velocity → velocity changes position

Einstein says: "Gravity curves space"

- Curved space → geodesics bend → light's *direction* changes

We need to track:

1. **Position** in polar coordinates: `(r, θ)`
2. **Velocity** in polar coordinates: `(dr/dλ, dθ/dλ)`

Think of `dr/dλ` as "how fast we're moving toward/away from the black hole" and `dθ/dλ` as "how fast we're rotating around it".

The General Geodesic Equation

Don't panic at this formula — we'll break it down:

$$d^2x^u/d\lambda^2 = -\Gamma^u\_\alpha\beta * (dx^\alpha/d\lambda) * (dx^\beta/d\lambda)$$

Translation:

- Left side: "acceleration" of coordinate x^u (could be r or θ)
- Right side: Christoffel symbols (Γ) that encode spacetime curvature, multiplied by current velocities

Christoffel Symbols (Γ)

These are the magic numbers that encode how curved spacetime is. For the Schwarzschild metric (non-rotating black hole), physicists have already calculated these for us.

The ones we need:

For θ (angular) acceleration:

$$\Gamma^\theta\_r\theta = 1/r$$

For r (radial) acceleration:

$$\Gamma^r\_\theta\theta = -r * (1 - Rs/r)$$
$$\Gamma^r\_rr = Rs / (2*r^2 - 2*r*Rs)$$

Where `Rs` is the Schwarzschild radius.

Deriving Our Two Key Equations

Equation 1: Angular Acceleration ($d^2\theta/d\lambda^2$)

Starting from the geodesic equation for θ:

$$d^2\theta/d\lambda^2 = -\Gamma^\theta\_r\theta * (dr/d\lambda) * (d\theta/d\lambda) - \Gamma^\theta\_\theta r * (d\theta/d\lambda) * (dr/d\lambda)$$

Since $\Gamma^\theta\_r\theta = \Gamma^\theta\_\theta r = 1/r$ (symmetry), and they appear twice:

$$d^2\theta/d\lambda^2 = -2 * (1/r) * (dr/d\lambda) * (d\theta/d\lambda)$$

Simplified:

$$d^2\theta/d\lambda^2 = -(2/r) * (dr/d\lambda) * (d\theta/d\lambda)$$

Physical Interpretation: As you move radially (dr/dλ), your angular motion (dθ/dλ) changes. Closer to the black hole (smaller r) = stronger effect.

Equation 2: Radial Acceleration ($d^2r/d\lambda^2$)

Starting from the geodesic equation for r:

$$d^2r/d\lambda^2 = -\Gamma^r\_rr * (dr/d\lambda)^2 - \Gamma^r\_\theta\theta * (d\theta/d\lambda)^2$$

Plugging in our Christoffel symbols and simplifying (trust the algebra for now):

$$d^2r/d\lambda^2 = (c^2 * Rs)/(2*r^2) + r * (d\theta/d\lambda)^2$$

Physical Interpretation:

- First term: $(c^2 * Rs)/(2*r^2)$ — gravity pulls you inward (always positive for r > Rs)
- Second term: $r * (d\theta/d\lambda)^2$ — angular momentum tries to push you outward (centrifugal-like effect)

Code Implementation Pattern

Your geodesic function structure:

```
void calculateAccelerations(LightRay& ray, BlackHole& bh) {
    // Extract current state
    float r = ray.r;
    float dr_dlambda = ray.dr_dlambda;
    float dtheta_dlambda = ray.dtheta_dlambda;
    float Rs = bh.schwarzschildRadius;

    // Calculate accelerations using the two equations above
    float d2theta_dlambda2 = /* Equation 1 */;
    float d2r_dlambda2 = /* Equation 2 */;

    // Store results
    ray.d2r_dlambda2 = d2r_dlambda2;
    ray.d2theta_dlambda2 = d2theta_dlambda2;
}
```

Numerical Integration (RK4)

The Problem with Simple "Euler" Method

You might think: "I have accelerations, I'll just do this":

```
  velocity += acceleration * dt;
  position += velocity * dt;
```

This is called **Euler's method**. It's fast but **inaccurate** for curved paths.

Why it fails: When gravity is changing rapidly (near a black hole), taking big straight steps leads to errors. Your ray might:

- Jump through the event horizon
- Take unrealistically sharp turns
- Drift off the true geodesic path

The Solution: Runge-Kutta 4 (RK4)

RK4 is like "looking ahead" before taking a step. It samples the path at 4 different points and averages them for a much more accurate step.

RK4 Algorithm Breakdown

Think of it like this: You're driving on a winding mountain road in the dark.

- **Euler method**: Look straight ahead, assume road continues straight, gun it
- **RK4 method**:
  1. Check what's immediately ahead (k1)
  2. Take a half-step and check again (k2)
  3. Take a different half-step and check (k3)
  4. Take a full step and check (k4)
  5. Average all 4 checks with special weights

The RK4 Formula

For a system with state `y` (position, velocity) and derivative `f(y)` (velocity, acceleration):

```
k1 = f(y_n)
k2 = f(y_n + (dt/2)*k1)
k3 = f(y_n + (dt/2)*k2)
k4 = f(y_n + dt*k3)

y_n+1 = y_n + (dt/6) * (k1 + 2*k2 + 2*k3 + k4)
```

Translating to Our Light Ray

Our "state" has 4 variables:

1. `r` (radial position)
2. `θ` (angular position)
3. `dr/dλ` (radial velocity)
4. `dθ/dλ` (angular velocity)

One RK4 step:

```
Step 1 (k1): Calculate accelerations at current position
Step 2 (k2): Create temporary ray halfway along k1, calculate
accelerations
Step 3 (k3): Create temporary ray halfway along k2, calculate
accelerations
Step 4 (k4): Create temporary ray fully along k3, calculate accelerations

Final update: Weighted average of all 4 samples
```

RK4 Implementation Pattern

```
void rk4_step(LightRay& ray, BlackHole& bh, float dt) {
    // Save initial state
    State initial = getCurrentState(ray);

    // k1: Evaluate at current position
    calculateAccelerations(ray, bh);
    State k1 = makeStateFromDerivatives(ray);

    // k2: Evaluate at midpoint using k1
    applyState(ray, initial + k1 * (dt/2));
    calculateAccelerations(ray, bh);
    State k2 = makeStateFromDerivatives(ray);

    // k3: Evaluate at midpoint using k2
    applyState(ray, initial + k2 * (dt/2));
    calculateAccelerations(ray, bh);
    State k3 = makeStateFromDerivatives(ray);

    // k4: Evaluate at endpoint using k3
    applyState(ray, initial + k3 * dt);
    calculateAccelerations(ray, bh);
    State k4 = makeStateFromDerivatives(ray);
    // Combine with RK4 weights
    State finalDelta = (k1 + 2*k2 + 2*k3 + k4) * (dt / 6.0f);
    applyState(ray, initial + finalDelta);
}
```

Helper State Struct

To make RK4 cleaner, create a simple state holder:

```
struct RayState {
    float r;
    float theta;
    float dr_dlambda;
    float dtheta_dlambda;

    // Enable math operations: state + state, state * scalar, etc.
};
```

Implementation Roadmap

Step 1: Expand Your LightRay Struct

Current (`BlackHole.hpp:18`):

```
struct LightRay {
    glm::vec2 position;  // Cartesian
    glm::vec2 velocity;  // Cartesian
    std::vector<glm::vec2> trail;
    void step(float deltaTime);
};
```

Add polar coordinates:

```
struct LightRay {
    // Cartesian (for rendering)
    glm::vec2 position;
    std::vector<glm::vec2> trail;

    // Polar (for physics)
    float r;            // distance from black hole
    float theta;        // angle around black hole
    float dr_dlambda;   // radial velocity
    float dtheta_dlambda; // angular velocity

    // Accelerations (calculated by geodesic function)
    float d2r_dlambda2;
    float d2theta_dlambda2;

    // Control
    bool active;  // false when hits event horizon

    void step(float deltaTime);  // Will be replaced with RK4
};
```

Step 2: Create Coordinate Conversion Functions

Add these as free functions or static methods:

```
// Convert Cartesian position to polar relative to black hole
glm::vec2 cartesianToPolar(glm::vec2 pos, glm::vec2 blackHolePos);

// Convert polar to Cartesian for rendering
glm::vec2 polarToCartesian(float r, float theta, glm::vec2 blackHolePos);
```

Math you'll use: See Coordinate Systems section above.

Step 3: Implement Geodesic Function

This calculates the accelerations:

```
void calculateAccelerations(LightRay& ray, const BlackHole& bh) {
    float r = ray.r;
    float dr = ray.dr_dlambda;
    float dtheta = ray.dtheta_dlambda;
    float Rs = bh.schwarzschildRadius;
    float c = C; // speed of light constant

    // Equation 1: angular acceleration
    ray.d2theta_dlambda2 = -(2.0f / r) * dr * dtheta;

    // Equation 2: radial acceleration
    ray.d2r_dlambda2 = (c*c * Rs) / (2.0f * r*r) + r * dtheta*dtheta;
}
```

Math you'll use: See The Geodesic Equation section above.

Step 4: Implement RK4

This is the most complex part. You have two approaches:

Option A: Simple but verbose — manually write out k1, k2, k3, k4 steps

Option B: Create a helper `RayState` struct and use operator overloading for cleaner code

Pseudocode:

1. Save initial state (r, θ, dr/dλ, dθ/dλ)
2. Calculate k1 (accelerations at current state)
3. Create temporary state at `initial + k1 * dt/2`, calculate k2
4. Create temporary state at `initial + k2 * dt/2`, calculate k3
5. Create temporary state at `initial + k3 * dt`, calculate k4
6. Update ray: `initial + (k1 + 2*k2 + 2*k3 + k4) * dt/6`

Math you'll use: See Numerical Integration section above.

Step 5: Initialize Light Rays

In your main loop, create rays with initial conditions:

```
// Example: Horizontal line of rays moving right
for (int i = 0; i < numRays; i++) {
    LightRay ray;

    // Cartesian: start on left side of screen
    ray.position = glm::vec2(100.0f, 300.0f + i * 10.0f);

    // Convert to polar relative to black hole at (400, 300)
    glm::vec2 polar = cartesianToPolar(ray.position, blackHolePos);
    ray.r = polar.x;
    ray.theta = polar.y;

    // Initial velocity: moving right at speed of light
    glm::vec2 velocity = glm::vec2(C, 0.0f);
    // Convert velocity to polar (this is tricky - see note below)
    ray.dr_dlambda = /* ... */;
    ray.dtheta_dlambda = /* ... */;

    ray.active = true;
    rays.push_back(ray);
}
```

Note on velocity conversion: This is subtle. You need to convert the velocity direction, not just the position. Look up "polar velocity conversion" or derive from the chain rule.

Step 6: Main Simulation Loop

```
while (!glfwWindowShouldClose(window)) {
    glClear(GL_COLOR_BUFFER_BIT);

    // Update physics
    for (auto& ray : rays) {
        if (ray.active) {
            // Check event horizon
            if (ray.r < blackHole.schwarzschildRadius) {
                ray.active = false;
                continue;
            }

            // Advance ray one step using RK4
            rk4_step(ray, blackHole, dt);

            // Convert polar back to Cartesian for rendering
            ray.position = polarToCartesian(ray.r, ray.theta, blackHolePc

            // Add to trail
            ray.trail.push_back(ray.position);
        }
    }

    // Render black hole
    renderBlackHole();

    // Render ray trails
    for (const auto& ray : rays) {
        renderTrail(ray.trail);
    }
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Step 7: Trail Rendering

You'll need to render lines. Quick method:

```
void renderTrail(const std::vector<glm::vec2>& trail) {
    // Create line mesh from trail points
    // Use GL_LINE_STRIP to connect them
    // Optional: Fade color along trail (brightest at tip)
}
```

Debugging Tips

Sanity Checks

1. **Ray going straight without black hole**:
   - Set black hole mass to zero
   - Rays should move in perfectly straight lines
2. **Schwarzschild radius calculation**:
   - For a solar mass black hole (2×10^{30} kg): Rs ≈ 3 km
   - Make sure your units are consistent (meters, not pixels!)
3. **Coordinate conversions**:
   - Convert position to polar, then back to Cartesian
   - Should get exactly the same position back
4. **RK4 step size**:
   - Start with large `dt` (like 1.0) to see if rays curve at all
   - Decrease `dt` (0.1, 0.01) for more accuracy
   - Too small `dt` = simulation runs slow but looks smooth

Common Issues

Problem	Likely Cause	Fix
Rays jump through black hole	Euler method, not RK4	Implement RK4
Rays don't curve	Forgot to call `calculateAccelerations()`	Check RK4 implementation
Rays curve wrong direction	Sign error in geodesic equations	Double-check the minus signs
Simulation super slow	`dt` too small, or too many steps	Increase dt, reduce ray count
Weird spiral patterns	Units inconsistent (mixing pixels and meters)	Normalize units or use dimensionless values

---

Advanced: Understanding the Physics Deeper

Why These Specific Equations?

The Schwarzschild metric describes spacetime curvature around a non-rotating black hole:

$$ds^2 = -(1 - Rs/r)c^2dt^2 + (1 - Rs/r)^{-1}dr^2 + r^2d\theta^2$$

This is the "shape" of curved spacetime. The Christoffel symbols come from taking derivatives of this metric. The geodesic equation then uses these symbols to compute how paths curve.

If you want to dive deeper:

- Look up "Schwarzschild metric derivation"
- Study "Christoffel symbols calculation"
- Read about "geodesic equation in general relativity"

Null Geodesics

Light travels on **null geodesics** ($ds^2 = 0$). This is why we can ignore the time component in our simulation — light doesn't experience time. We only track spatial path.

Units and Scaling

In reality:

- $c = 299{,}792{,}458$ m/s
- $G = 6.674\times10^{-11}$ $m^3/(kg\cdot s^2)$
- Solar mass black hole: Rs ≈ 3,000 meters

For simulation, you might want to use **geometric units** where c = G = 1, or scale everything to pixels.

Resources for Further Learning

- **YouTube Tutorial**: The video you referenced is excellent for visual understanding
- **Equations**: Schwarzschild geodesics on Wikipedia
- **RK4 Tutorial**: Numerous online tutorials for "Runge-Kutta 4th order"
- **General Relativity**: Leonard Susskind's lecture series (if you want the full physics)

Final Encouragement

The math looks intimidating, but break it down:

1. **Polar coordinates**: Just trig functions
2. **Geodesic equations**: Two formulas you plug values into
3. **RK4**: A recipe you follow step-by-step

Start small: Get one ray to curve slightly. Then add more rays. Then perfect the physics. You've got this!

Good luck! When you're ready to test your implementation, try:

- Single ray aimed to just miss the black hole → should curve around
- Ray aimed directly at center → should spiral in
- Ray far away → should barely curve

If these work, you've nailed it!