On the quantum Schwarzschild gravitational theories consisting of field lines, for C++ programmers

S. Halayka*

Saturday 15$^{\text{th}}$ November, 2025 10:51

Abstract

This paper contains a short introduction to isotropic Schwarzschild gravitation. It is found that with non-normal (e.g. cosine weighted) messenger particle emission comes increased gravitational strength – matching that from the Schwarzschild solution. It is also found that there can be repulsive gravitation, at least for small receivers close to the Planck scale.

1 Introduction

First see [1] for a short tutorial for C++ programmers on isotropic Newtonian gravitation. In [1] we build an isotropic gravitational field through the use of pseudorandomly generated field lines. In [1] we use a sphere as the receiver.

In this paper, we find a match between the numerical gravitation and the gravitational time dilation from Schwarzschild's general relativity [2–4]. Here, we use an axis-aligned bounding box (AABB) as the receiver.

In this paper we use Planck units, where $c = G = \hbar = k = 1$, which simplifies the equations. Note that a length of 1 means 1 Planck length, not 1 metre.

2 Method

Where r_e is the emitter's Schwarzschild radius, r_r is the receiver AABB radius (e.g. half of the AABB side length), and 1e11 and 0.01 are arbitrary constants:

$$r_e = \sqrt{\frac{1\text{e}11 \log(2)}{\pi}}, \tag{1}$$

$$r_r = r_e \times 0.01. \tag{2}$$

The event horizon area is:

$$A_e = 4\pi r_e^2. \tag{3}$$

The entropy (e.g. field line count) is:

$$n_e = \frac{A_e}{4\log(2)} = 1\text{e}11. \tag{4}$$

*sjhalayka@gmail.com

Where R is the distance from the emitter's centre, the derivative is:

$$\alpha = \frac{\beta(R+\epsilon) - \beta(R)}{\epsilon}. \tag{5}$$

Here β is the get intersecting line density function. The gradient strength is:

$$g = \frac{-\alpha}{r_r^2}. \tag{6}$$

From this we can get the Newtonian acceleration a_N, where $r_e \ll R$:

$$a_N = \frac{gR\log 2}{8M_e} = \sqrt{\frac{n_e \log 2}{4\pi R^4}} = \frac{M_e}{R^2}. \tag{7}$$

We can also get a general relativistic acceleration a_S, where $r_e < R$:

$$a_S = \frac{gR\log 2}{8M_e}, \tag{8}$$

At close proximity, where $r_e \approx R$, the metric produced is related to the Schwarzschild metric – curved space, curved time:

$$t = \sqrt{1 - \frac{r_e}{R}}, \tag{9}$$

$$\frac{\partial t}{\partial R} = \frac{r_e}{2tR^2}. \tag{10}$$

$$a_S \approx \frac{\partial t}{\partial R}\frac{2}{\pi} = \frac{r_e}{\pi t R^2}. \tag{11}$$

At far proximity, where $r_e \ll R$, $t \approx 1$, and $\frac{\partial t}{\partial R} \approx 0$, the metric produced is Newtonian – curved space, practically flat time.

In general relativity, acceleration is also dependent on the kinematic time dilation of the gravitated body – internal process forms a resistance to gravitation. Where the speed is $v \approx c$, like that for neutrinos, the gravitational attraction is twice of that predicted by Newtonian gravity because of a lack of said resistance.

3 C++ code

Some helper functions are:

```
double intersect_AABB(
        const vector_3 min_location,
        const vector_3 max_location,
        const vector_3 ray_origin,
        const vector_3 ray_dir,
        double& tmin,
        double& tmax)
{
        tmin = (min_location.x - ray_origin.x) / ray_dir.x;
        tmax = (max_location.x - ray_origin.x) / ray_dir.x;

        if (tmin > tmax) swap(tmin, tmax);
```

```
        double tymin = (min_location.y - ray_origin.y) / ray_dir.y;
        double tymax = (max_location.y - ray_origin.y) / ray_dir.y;

        if (tymin > tymax) swap(tymin, tymax);
        if ((tmin > tymax) || (tymin > tmax)) return 0;
        if (tymin > tmin) tmin = tymin;
        if (tymax < tmax) tmax = tymax;

        double tzmin = (min_location.z - ray_origin.z) / ray_dir.z;
        double tzmax = (max_location.z - ray_origin.z) / ray_dir.z;

        if (tzmin > tzmax) swap(tzmin, tzmax);
        if ((tmin > tzmax) || (tzmin > tmax)) return 0;
        if (tzmin > tmin) tmin = tzmin;
        if (tzmax < tmax) tmax = tzmax;
        if (tmin < 0 || tmax < 0) return 0;

        vector_3 ray_hit_start = ray_origin;
        ray_hit_start.x += ray_dir.x * tmin;
        ray_hit_start.y += ray_dir.y * tmin;
        ray_hit_start.z += ray_dir.z * tmin;

        vector_3 ray_hit_end = ray_origin;
        ray_hit_end.x += ray_dir.x * tmax;
        ray_hit_end.y += ray_dir.y * tmax;
        ray_hit_end.z += ray_dir.z * tmax;

        double l = (ray_hit_end - ray_hit_start).length();

        return l;
}

vector_3 random_cosine_weighted_hemisphere(const vector_3& normal)
{
        double u1 = dis(generator);
        double u2 = dis(generator);

        double r = sqrt(u1);
        double theta = 2.0 * pi * u2;

        double x = r * cos(theta);
        double y = r * sin(theta);

        double z = sqrt(1.0 - u1);

        vector_3 n = normal;
        n.normalize();

        vector_3 arbitrary;
        if (fabs(n.x) > 0.9)
                arbitrary = vector_3(0, 1, 0);
        else
                arbitrary = vector_3(1, 0, 0);

        vector_3 tangent = n.cross(arbitrary);
```

```
        tangent.normalize();

        vector_3 bitangent = n.cross(tangent);
        bitangent.normalize();

        vector_3 result;
        result.x = tangent.x * x +
                bitangent.x * y + n.x * z;

        result.y = tangent.y * x +
                bitangent.y * y + n.y * z;

        result.z = tangent.z * x +
                bitangent.z * y + n.z * z;

        return result.normalize();
}

std::optional<double> intersect(
        const vector_3 location,
        const vector_3 normal,
        const double receiver_distance,
        const double receiver_radius)
{
        const vector_3 circle_origin(receiver_distance, 0, 0);

        if (normal.dot(circle_origin) <= 0)
                return std::nullopt;

        vector_3 min_location(
                -receiver_radius + receiver_distance,
                -receiver_radius,
                -receiver_radius);

        vector_3 max_location(
                receiver_radius + receiver_distance,
                receiver_radius,
                receiver_radius);

        double tmin = 0, tmax = 0;

        double AABB_hit = intersect_AABB(
                min_location,
                max_location,
                location,
                normal,
                tmin,
                tmax);

        if (AABB_hit > 0)
                return AABB_hit;

        return std::nullopt;
}
```

The following code uses the Newtonian gravitation, where the random unit vector points in the same direction as the accompanying normal:

```
// Beta function, for Newtonian gravitation
double get_intersecting_line_density(
	const long long unsigned int n,
	const double emitter_radius,
	const double receiver_distance,
	const double receiver_radius)
{
	double count = 0;

	generator.seed(static_cast<unsigned>(0));

	for (long long unsigned int i = 0; i < n; i++)
	{
		const vector_3 p = random_unit_vector();

		vector_3 normal = p;
		vector_3 location = normal;

		location.x *= emitter_radius;
		location.y *= emitter_radius;
		location.z *= emitter_radius;

		std::optional<double> i_hit = intersect(
			location,
			normal,
			receiver_distance,
			receiver_radius);

		if (i_hit)
			count += *i_hit / (2.0 * receiver_radius);
	}

	return count;
}
```

The following code uses the Schwarzschild gravitation, where the random unit vector generally points in a different direction than the normal, using cosine weighting:

```
// Beta function, for Schwarzschild gravitation
double get_intersecting_line_density(
	const long long unsigned int n,
	const double emitter_radius,
	const double receiver_distance,
	const double receiver_radius)
{
	double count = 0;

	generator.seed(static_cast<unsigned>(0));

	for (long long unsigned int i = 0; i < n; i++)
	{
		vector_3 location = random_unit_vector();

		location.x *= emitter_radius;
		location.y *= emitter_radius;
		location.z *= emitter_radius;
```

```
        vector_3 surface_normal = location;
        surface_normal.normalize();

        vector_3 normal =
            random_cosine_weighted_hemisphere(
                surface_normal);

        std::optional<double> i_hit = intersect(
            location, normal,
            receiver_distance, receiver_radius);

        if (i_hit)
            count += *i_hit / (2.0 * receiver_radius);
    }

    return count;
}
```

4 Conclusion

It's worth noting that gravitation, if cosine weighted, is a diffuse process similar to electromagnetic lighting in importance sampling in path tracing from computer graphics.

Experimentation with the receiver size is necessary to understand the intricacies of the code. For instance, where $r_r = r_e \times 0.01$, it is found that gravitation can be repulsive.

The full code for this paper can be found at:

`https://github.com/sjhalayka/schwarzschild_falloff_field_lines`

Note that while Quantum Graphity [5] uses gravitons, it is practically the same as using field lines, and so there is repulsion in some cases as well. The code for Quantum Graphity is at:

`https://github.com/sjhalayka/quantum_graphity`

It should be noted that repulsive gravitation plagues all non-normal (e.g. cosine weighted) theories. It should also be noted that getting rid of the field lines altogether, by replacing them with randomly emitted gravitons, does not solve the problem of repulsive gravitation.

References

[1] Halayka. Newtonian gravitation from scratch, for C++ programmers. (2024)

[2] 't Hooft. Dimensional reduction in quantum gravity. (1993)

[3] Susskind. The World as a Hologram. (1994)

[4] Misner et al. Gravitation. (1970)

[5] Konopka et al. Quantum Graphity: a model of emergent locality (2008)

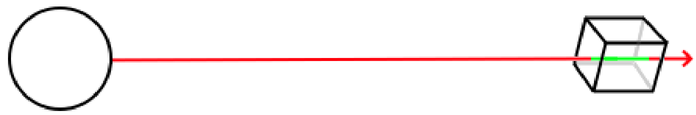

Figure 1: This figure shows an axis-aligned bounding box and an isotropic emitter, looking from slightly above. An example field line (red) and intersecting line segment (green) are given. The bounding box is filled with these green intersecting line segments. It is the gradient of the density of these line segments that forms the gravitational acceleration.

‘

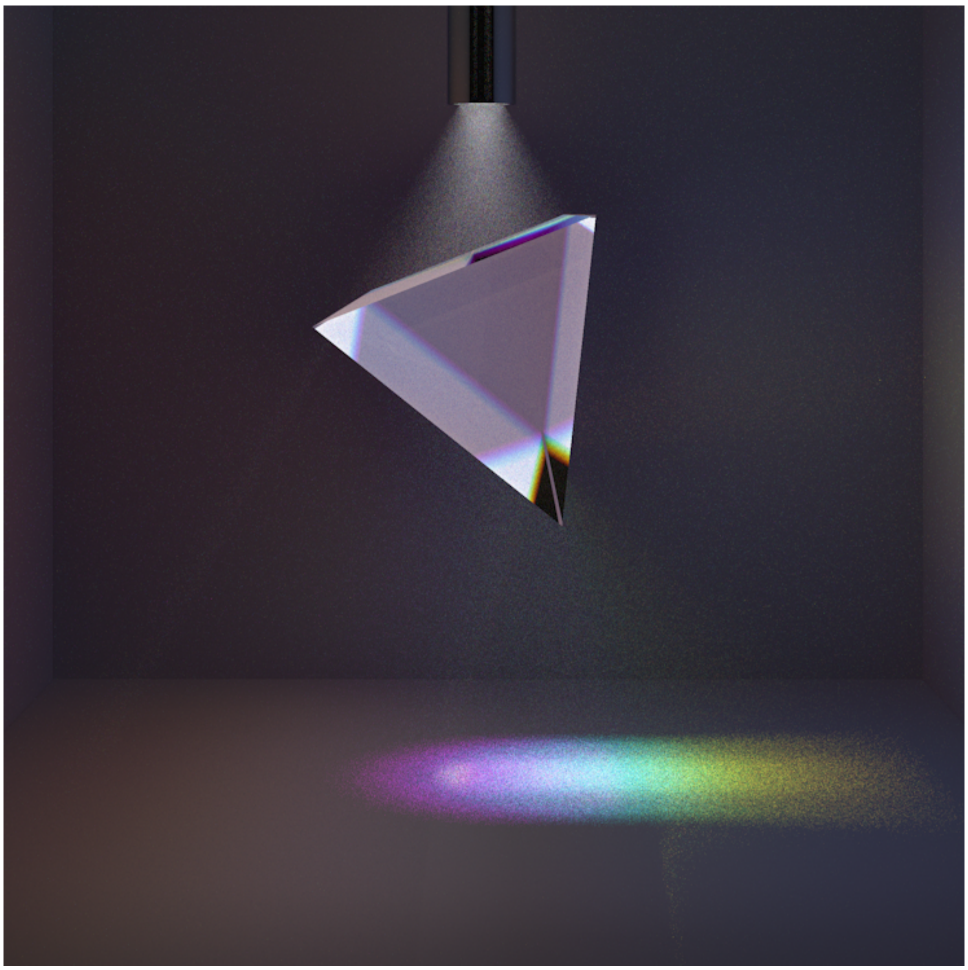

Figure 2: Backward path tracing in the Vulkan graphics API. The scene has a white barrel light that is dispersed by a prism into a spectrum of colours. There is some fog.