On quantum Schwarzschild gravitational field line models in C++

S. Halayka*

Tuesday 6th January, 2026 09:15

Abstract

This article elucidates a new conceptual approach to isotropic Schwarzschild gravitation. The focus is on the non-uniform angular distributions of hypothetical gravitational particles i.e. the emission of these particles is in a cosine-weighted pattern, and we show that the effective gravitational strength increases in a way that aligns with the predictions of the Schwarzschild solution.

1 Introduction

The C++ programmers were given a fundamental overview of isotropic Newtonian gravitation in [1], and a computational framework to simulate gravitational fields with the use of pseudorandomly generated field lines. This earlier research indicated the potential of an isotropic arrangement of such lines to produce the well-known inverse-square law of Newtonian gravity.

This current paper aims to extend that framework for the investigation of the relation between the numerically produced isotropic gravitation, and the gravitational time dilation effects foretold by the Schwarzschild solution of general relativity [2–4]. This current paper opens the way to more general computational scenarios through the use of an axis-aligned bounding box (AABB), replacing the spherical receiver in [1], which will allow simulations that are more suited to grid-based, voxel-based, or rectangular geometries usually employed in real-time or offline C++ physics engines.

This paper considers Planck units throughout, choosing the natural unit convention $c = G = h = k = 1$. This significantly reduces the complexity of mathematical expressions, and allows gravitational quantities, which include length, mass, and time, to have dimensionally consistent forms without explicit constants.

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:

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

*sjhalayka@gmail.com

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

The event horizon area is:

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

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

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

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:

$$g = \frac{-\alpha}{2r_r^3}. \tag{6}$$

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

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

The relativistic acceleration a_S is established for every region where $r_e < R$. In this case, the same gradient-based expression is applied, but is not restricted by the weak-field condition:

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

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;
        }

        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;
        }

        return count;
}
```

4 Discussion

According to the results, behaviour in curved spacetime usually related with tensor calculus and differential geometry can be reached by a quantized geometric method.

From the perspective of computation, the approach provides a great deal of benefit to C++-based physics engines. Applying AABB receivers instead of spherical symmetry allows the method to be immediately usable together with voxel grids, spatial partitioning structures and real-time simulation pipelines. Since the model does not involve any differential operators, tensor algebra or explicit metric computation, it is thus suitable for parallel ray-based systems like GPU kernels or Monte Carlo physics engines, and this is why the model is regarded as very effective for numerical relativity approximations, gravitational visualization, game engine physics, and research that uses ray tracing to represent the dynamics of emergent spacetime.

5 Conclusion

In this study, it was shown, through the modeling of gravitational emission using cosine-weighted directionality that the resulting process is similar to diffuse illumination in electromagnetic path tracing a technique widely used in the field of computer graphics. The similarity is important: it indicates that the gravitational field line models, with non-uniform angular weighting, display behaviours that are comparable to radiative transport phenomena thus highlighting the simulations of such situations as being fundamentally statistical and directional.

The sensitivity of the model to the receiver size selection is one of the main practical aspects of this study. Numerical experiments showed that the ratio of the emitter's radius to the receiver's radius plays a big role in how the computed gravitational field behaves. For instance, when using the parameter $r_r = r_e \times 0.01$, the simulation suggests that the gravitational force can actually become repulsive depending on the arrangements. This unexpected result highlights the need for precise geometric calibration and indicates that repulsion is not merely a numerical artifact but rather a feature of the directional field-line formulation that emerges.

The complete source code tied to this article is offered at this link:

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

The feature of repulsive gravitation is not a singular one but rather a dominant attribute of any theory that makes use of non-normal emission patterns, even if these emissions are represented as randomly oriented gravitons instead of explicit field lines. The continuity of repulsion across such dissimilar frameworks hints at the existence of a deeper structural attribute of cosine-weighted or otherwise biased emission models. Therefore, despite the fact that these models provide intuitive and computationally easy-to-use analogues to curved spacetime behaviour, they also bring about phenomena that go against classical expectations and thus require further theoretical investigation.

6 Conflict of interest

There are no conflicts of interest to report.

7 Data availability

The C++ code is freely available, providing a way to reproduce the data.

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)

‘

‘

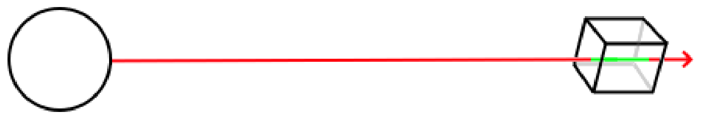

Figure 1: This figure shows an axis-aligned bounding box and an isotropic emitter, looking from slightly above. An example field line (red) and intersecting line segment (green) are given. The bounding box is filled with these green intersecting line segments. It is the gradient of the density of these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

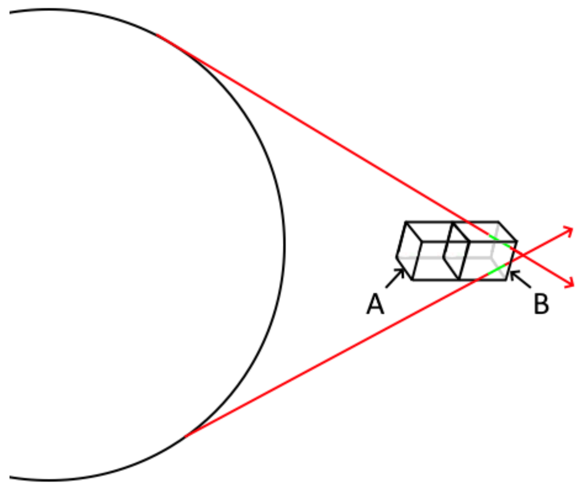

Figure 2: Axis-aligned bounding box A and box B are separated by the value epsilon $\epsilon = 2r_r$. Box B contains more field lines than box A, resulting in repulsive gravitation. Note that the field lines are not normal to the surface.