Simulating Light Behavior Using Vector through Path Tracing for Realistic Image Rendering

Fachriza Ahmad Setiyono - 13523162
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*rizacal.mamen@gmail.com*, *13523162@std.stei.itb.ac.id*

***Abstract*—Path tracing, a widely used technique in computer graphics, models light transport by tracing rays and computing their interactions with surfaces. The study begins by examining the use of photorealistic image rendering, the fundamental properties of light, including reflection, refraction, and scattering, and demonstrates how these behaviors can be mathematically represented using vectors. Key vector operations such as dot and cross products are analyzed to handle computations related to surface normals, ray directions, and light intensity.**

***Keywords*—Light Transport, Path Tracing, Physically Based Rendering, Vector Operations.**

I. Introduction

Rendering realistic digital images has been an ongoing challenge in the computer graphics field. Several rendering techniques have been developed by researchers but there are 2 techniques that are widely used, rasterization and ray tracing. These 2 techniques are fundamentally different, thus has its own strengths and weaknesses. Rasterization works by projecting triangles directly to the screen, while ray tracing traces numbers of rays from the camera to objects, which results in higher performance cost. Despite being slower, ray tracing is still widely used for rendering photorealistic images because tracing individual ray means we can accurately simulate how real light behaves in real life.

Photorealistic image rendering is a field within computer graphics which focuses on producing realistic images by simulating real life physics phenomena in a way that models light and its interaction with surfaces. This approach is often called Physically Based Rendering (PBR). PBR has a lot of uses, especially in the entertainment industry such as video games and movies. In the business industry, PBR can be used to market and visualize products. Using computer-generated image is often favored because of its lower cost and higher efficiency while still being able to resemble the real product accurately. PBR is also used for educational purposes, such as flight and medical simulations.

One of the most popular techniques to render photorealistic images is path tracing. Path tracing utilizes ray tracing and expands on it by allowing light bounce after hitting a surface. Illustration of how ray bounces is shown in Fig. 1.

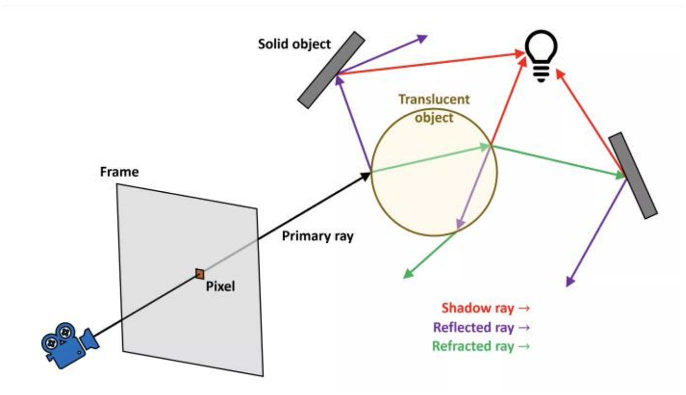


Fig. 1. Ray bounces in path tracing
(Source: https://www.techspot.com/articles-info/2485/images/2022-07-10-image.png)

Wave-particle duality theory states that light has the properties of both a wave and a particle. In an ideal environment, light as a particle moves in a straight line through a 3-dimensional space, while light as a wave “moves’ in a manner that resembles a wave function. Vector is used to represent light particles or rays as rays consist of an origin point and direction. This method of simulating light as rays is called geometric optic, as opposed to wave optic. However, simulating only the particle properties of light means that we cannot achieve physical phenomena caused by wave optics such as thin-film interference and diffraction. In this paper, we consider that only simulating light as a ray is sufficient to demonstrate the use of vector.

We will also analyze how vector algebraic operations are used in rendering techniques such as ray tracing and path tracing. In a state-of-the-art ray tracer, a more complex concept such as quaternion algebra and matrix operations might be used. However, that is outside the scope of this paper. This paper offers insights into how vectors serve as the foundation for modern rendering techniques and provides a basis for extending these methods to more advanced light simulations.

II. Background

A. Euclidean Vector

Euclidean vector or a vector is a mathematical geometric object that consists of magnitude and direction. Vectors are denoted as lowercase letter with arrow or hat on top such as $\vec{u}$ and $\hat{v}$.

Vector can also be denoted as two uppercase letters with arrow on top $\overrightarrow{AB}$ to depict a connection from point A to point B. The magnitude of a vector is the distance between the two points while the direction is the direction of the vector displacement.

Vectors can also be written in a coordinate system such as (v_1, v_2, v_3) to represent a 3-dimensional vector $\vec{v}$. The coordinate system used here refers to $(0,0,0)$ as the origin point.

There are various types of vectors such as:

1. Zero Vector

Zero vector is a vector with magnitude of zero and no direction, denoted by $\vec{0}$. In a 3-dimension Euclidean space with coordinate system, the zero vector is written as $(0,0,0)$.

2. Position Vector

A point in an n-dimensional space is represented by n-dimensional vectors. Position vector is denoted with lowercase letter with arrow on top such as $\vec{u}$. In physical sense, the magnitude of position vector is the distance between the point and other relative arbitrary point, usually the origin point $\vec{0}$. This means that a single point has different position vectors in relation to different reference points.

3. Unit Vector

Unit vector is a vector with magnitude of 1, denoted by lowercase letter with a hat as in $\hat{v}$. We can say that the normalized vector $\hat{u}$ is the unit vector of a non-zero vector $\vec{u}$, or in mathematical notation,

$$\hat{u} = \frac{\vec{u}}{\|\vec{u}\|} \tag{1}$$

Because of this, the term unit vector and normalized vector is often synonymously used.

Unit vector is used a lot in the scope of computer graphics, notably to denote directions. Having the direction normalized simplifies the unit to be used by other operations that requires direction. For instance, a ray in a 3-dimensional space from point A to point B in reference to the origin point can be formed with a position vector and a unit vector as a direction with multiplier t as the ray distance. This can be written as:

$$P(t) = \vec{O} + t\hat{D} \tag{2}$$

Where:

$\vec{O}$:	Ray origin point
$\hat{D}$:	Ray normalized direction from A to B
t:	Ray Euclidean distance

Equation (2) is called the parametric equation for rays. If the ray direction were not normalized, then we must account for the length of the direction vector when calculating the distance t. Having said that, normalizing vector is considered a costly operation, so it is wise to normalize a vector only when it is needed.

4. Normal Vector

Normal vector or simply normal is a vector that is perpendicular to its surface. While normal vector does not have to be a unit vector, it is very common to normalize normal vector for the very same reason explained in the previous section. In computer graphics, normal is required to

y
(v_1, v_2)
v
x
z
(v_1, v_2, v_3)
v
y
x

Fig. 2. a) Vector in a 2-dimensional Euclidean space and b) Vector in a 3-dimensional space
(Source: [Vektor di Ruang Euclidean (bagian 1)-2024](#))

calculate the shading on a surface. Normal can be calculated by finding the cross product of the surface points.

B. Euclidean Vector Space

A set of Euclidean vectors forms a Euclidean vector space, or simply called a Euclidean space. Euclidean space of n-dimensional vectors is denoted with R^n. In this paper, we will mostly deal with 3-dimensional Euclidean space.

All vectors within the Euclidean space must satisfy the properties of algebraic operations in a Euclidean space:

a) $\vec{u} + \vec{v} = \vec{v} + \vec{u}$
b) $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$
c) $\vec{u} + 0 = 0 + \vec{u} = \vec{u}$
d) $\vec{u} + (-\vec{u}) = 0$
e) $k(\vec{u} + \vec{v}) = k\vec{u} + k\vec{v}$
f) $(k + m)\vec{u} = k\vec{u} + m\vec{u}$
g) $k(m\vec{u}) = (km)\vec{u}$
h) $1\vec{u} = \vec{u}$

Vectors also have other operations defined in this space, such as:

1. Dot Product

A dot product of two non-zero Euclidean vectors $\vec{u}$ and $\vec{v}$ with angle θ is defined by geometric definition as:

$$\vec{u} \cdot \vec{v} = \|\vec{u}\|\|\vec{v}\| \cos\theta \tag{3}$$

In computer graphics, dot product or inner product is heavily used to calculate the product of $\cos\theta$ between two unit vectors. Looking at (3), if two of the vectors are normalized, then we can simply write it as:

$$\hat{u} \cdot \hat{v} = \cos\theta \tag{4}$$

Note that the result of dot product operation is a scalar and if both vectors are orthogonal, the resulting dot product is 0.

2. Cross Product

A cross product is only defined in a 3-dimensional Euclidean space and is denoted by $\vec{u} \times \vec{v}$. The result of a cross product between 2 vectors is another vector that is perpendicular to both vector $\vec{u}$ and $\vec{v}$. Mathematically, cross product is defined as:

$$\vec{u} \times \vec{v} = \|\vec{u}\|\|\vec{v}\| \sin\theta\, \hat{n} \tag{5}$$

Where:

θ: Angle between vector $\vec{u}$ and $\vec{v}$

$\hat{n}$: Unit vector of the resulting vector

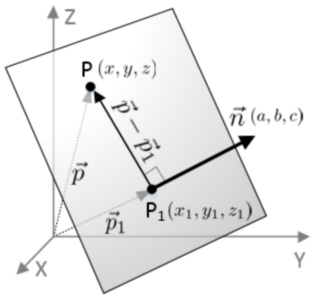


Fig. 4. Calculating plane normal using cross product.
(Source: https://www.songho.ca/math/plane/plane.html)

Cross products can be used to calculate the normal of a surface or plane P that is defined by set of points in space. Let P and P_1 be 3-dimensional position vector inside a defined plane P that are not parallel to each other, then the plane normal $\vec{n}$ can be calculated by finding the cross product between P and P_1. Note that you cannot use two position vectors that are parallel to calculate the normal because the angle would be either 0° or 180°, which will result in the product of $\sin\theta$ in (5) being zero, thus nullifying the normal vector. In computer graphics, we usually use quad or triangle vertices as the points relative to the surface to calculate the normal vector.

C. The Rendering Equation

Reference [1], [2] introduces the rendering equation (also called the reflectance equation) to the computer graphic field, an equation that models the interaction between light and surface. We can write a more generalized form of the equation as:

$$L_0(x, \omega_o) = \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \hat{n} \cdot \omega_i \, d\omega_i + L_e(x, \omega_o) \tag{6}$$

Where:

x: Position in space

$\hat{n}$: Unit normal vector at x

ω_o: Direction of outgoing radiance

ω_i: Negative direction of incoming radiance

L_o: Outgoing light radiance

L_i: Incoming light radiance

L_e: Emitted light radiance

f_r: Bidirectional Reflectance Distribution Function (BRDF)

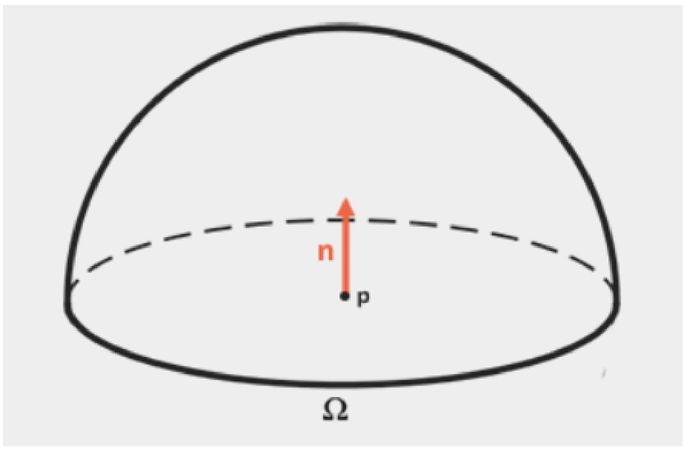


Fig. 3. Hemisphere oriented around normal
(Source: https://learnopengl.com/PBR/Theory)

In other words, this equation states that at point x, the light coming in the direction of observer (e.g. our eyes) is the sum of infinitely many incoming lights from another point in space around the normal hemisphere multiplied by some factor. A hemisphere can be described as half a sphere aligned around the surface normal. Note that (6) considers every point visible from x as light sources.

The incoming lights are multiplied by the factor BRDF (f_r) and the dot product between the surface normal and the incoming light direction ($\hat{n} \cdot \omega_i$). The BRDF is a function that approximates how much an incoming light will contribute to the

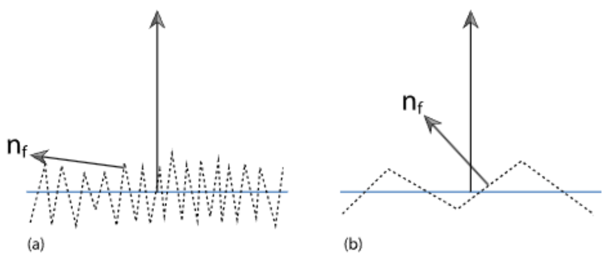


Fig. 5. Microfacet for a) very rough surface and b) smoother surface.
(Source: https://pbr-book.org/3ed-2018/Reflection_Models/Microfacet_Models)

final reflected light. There are many BRDF model such as Blinn-Phong Model, Cook-Torrance Model, and Oren-Nayar Model. Some models like Cook-Torrance is considered as a **microfacet model**. These models works by statistically modelling the scattering of light from a large collection of many microscopic mirror-like surfaces (microfacet) that defines a surface.

Microfacet models are used because we are limited by a lot of factors such as the screen pixel size. In Fig we can see the "macrosurface" as the blue line with the normal going straight up. If that is the smallest pixel that can be drawn to the screen, then we cannot represent how the surface is much rougher than it is, as shown by the jagged lines.

D. Path Tracing

Path tracing is a Monte Carlo method to solve the rendering equation. If we look back at (6), it is clear that the incoming light radiance from a direction ω_i must be known. But that incoming light from ω_i also need to be calculated using the rendering equation, adding layers of complex recursion. This makes it generally impossible to solve the equation analytically.

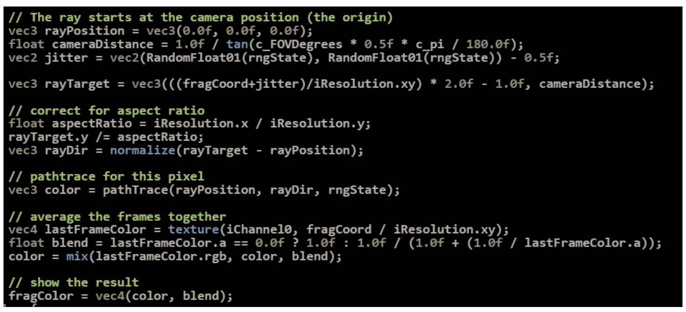
Path tracing works by generating several random rays from the surface normal hemisphere using some probability distribution and trace it until it hits another object in the space. Then we solve the rendering equation for said surface by only using those several ray samples and aggregating the result with the result from previous frames. It is important to divide the radiance by the probability of the sampled ray to ensure the result is comparably correct to the analytical version.

However, due to its unbiased nature, rendering images by sampling random lights as in path tracing will result in noise artifacts. This is why a naïve path tracer is only used to render reference images when testing and comparing other rendering algorithms.

III. METHOD

To get a better understanding of how vectors are used in path tracing, let us implement path tracing and render a simple scene. We use shadertoy to write GLSL code and render it.

First, lets set up the ray that we will trace, starting from the camera. Consider the camera as the origin point.

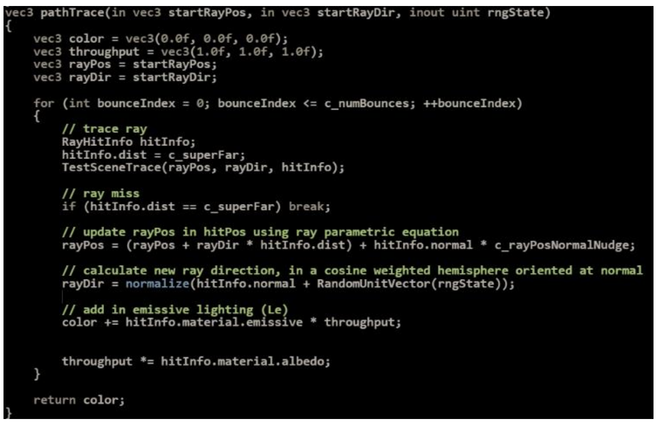


```
// The ray starts at the camera position (the origin)
vec3 rayPosition = vec3(0.0f, 0.0f, 0.0f);
float cameraDistance = 1.0f / tan(c_FOVDegrees * 0.5f * c_pi / 180.0f);
vec2 jitter = vec2(RandomFloat01(rngState), RandomFloat01(rngState)) - 0.5f;

vec3 rayTarget = vec3(((fragCoord+jitter)/iResolution.xy) * 2.0f - 1.0f, cameraDistance);

// correct for aspect ratio
float aspectRatio = iResolution.x / iResolution.y;
rayTarget.y /= aspectRatio;
vec3 rayDir = normalize(rayTarget - rayPosition);

// pathtrace for this pixel
vec3 color = pathTrace(rayPosition, rayDir, rngState);

// average the frames together
vec4 lastFrameColor = texture(iChannel0, fragCoord / iResolution.xy);
float blend = lastFrameColor.a == 0.0f ? 1.0f : 1.0f / (1.0f + (1.0f / lastFrameColor.a));
color = mix(lastFrameColor.rgb, color, blend);

// show the result
fragColor = vec4(color, blend);
```

Fig. 7. Ray set up.
(Source: author's code)

Here we set the rayPosition at the camera or the origin point. The ray direction would then be set to the viewport, which spans from the -1 to 1, from the left and bottom of the screen to the right and top.

Now we can set up the main path tracing loop. The way this works is by using a for-loop with each iteration indicating each light bounce. While light in real life bounces a lot of time until it runs out of energy, the amount of energy loss after the first few bounces is enough to make it insignificant to the render result. For the sake of simplicity, we will only simulate the diffuse reflection part only. This means in every bounce; the light ray will generate a totally random cosine-weighted direction oriented around the normal hemisphere.

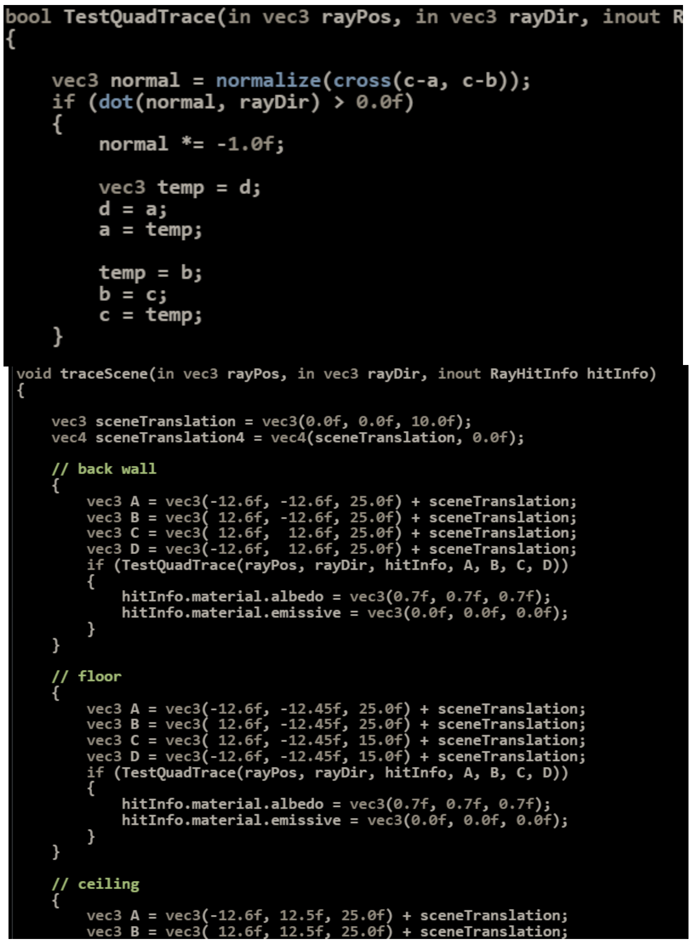


```
vec3 pathTrace(in vec3 startRayPos, in vec3 startRayDir, inout uint rngState)
{
    vec3 color = vec3(0.0f, 0.0f, 0.0f);
    vec3 throughput = vec3(1.0f, 1.0f, 1.0f);
    vec3 rayPos = startRayPos;
    vec3 rayDir = startRayDir;

    for (int bounceIndex = 0; bounceIndex <= c_numBounces; ++bounceIndex)
    {
        // trace ray
        RayHitInfo hitInfo;
        hitInfo.dist = c_superFar;
        TestSceneTrace(rayPos, rayDir, hitInfo);

        // ray miss
        if (hitInfo.dist == c_superFar) break;

        // update rayPos in hitPos using ray parametric equation
        rayPos = (rayPos + rayDir * hitInfo.dist) + hitInfo.normal * c_rayPosNormalNudge;

        // calculate new ray direction, in a cosine weighted hemisphere oriented at normal
        rayDir = normalize(hitInfo.normal + RandomUnitVector(rngState));

        // add in emissive lighting (Le)
        color += hitInfo.material.emissive * throughput;


        throughput *= hitInfo.material.albedo;
    }

    return color;
}
```

Fig. 9. Path tracing loop.
(Source: autho's code)

After tracing the ray, if the ray hits an object, it will grab the material data and calculate the radiance for that position. However, if the ray misses an object, it will simply add zero or black. Usually in most other path tracers, when missing an object, a sky illumination is used. It is used to add "ambient" feel and lighting

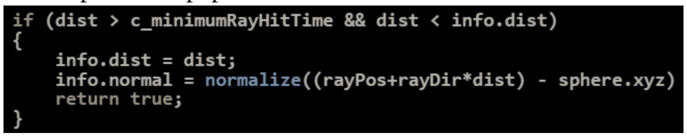
On each loop, we trace the ray by checking if the ray with the given origin and direction will intersect with the objects on the scene. For the sake of simplicity, let's just assume that only planes and spheres exist. The traceScene function will check if the ray intersect will any of the objects by checking it one-by-one. Obviously, this is not the best way to handle this. Much more "smart" and performant method of checking ray-object intersection have been developed such as BVH tree, which groups triangles into some sort of binary tree. But this is outside the scope of this paper.



```
bool TestQuadTrace(in vec3 rayPos, in vec3 rayDir, inout R
{

    vec3 normal = normalize(cross(c-a, c-b));
    if (dot(normal, rayDir) > 0.0f)
    {
        normal *= -1.0f;

        vec3 temp = d;
        d = a;
        a = temp;

        temp = b;
        b = c;
        c = temp;
    }
```

```
void traceScene(in vec3 rayPos, in vec3 rayDir, inout RayHitInfo hitInfo)
{

    vec3 sceneTranslation = vec3(0.0f, 0.0f, 10.0f);
    vec4 sceneTranslation4 = vec4(sceneTranslation, 0.0f);

    // back wall
    {
        vec3 A = vec3(-12.6f, -12.6f, 25.0f) + sceneTranslation;
        vec3 B = vec3( 12.6f, -12.6f, 25.0f) + sceneTranslation;
        vec3 C = vec3( 12.6f,  12.6f, 25.0f) + sceneTranslation;
        vec3 D = vec3(-12.6f,  12.6f, 25.0f) + sceneTranslation;
        if (TestQuadTrace(rayPos, rayDir, hitInfo, A, B, C, D))
        {
            hitInfo.material.albedo = vec3(0.7f, 0.7f, 0.7f);
            hitInfo.material.emissive = vec3(0.0f, 0.0f, 0.0f);
        }
    }

    // floor
    {
        vec3 A = vec3(-12.6f, -12.45f, 25.0f) + sceneTranslation;
        vec3 B = vec3( 12.6f, -12.45f, 25.0f) + sceneTranslation;
        vec3 C = vec3( 12.6f, -12.45f, 15.0f) + sceneTranslation;
        vec3 D = vec3(-12.6f, -12.45f, 15.0f) + sceneTranslation;
        if (TestQuadTrace(rayPos, rayDir, hitInfo, A, B, C, D))
        {
            hitInfo.material.albedo = vec3(0.7f, 0.7f, 0.7f);
            hitInfo.material.emissive = vec3(0.0f, 0.0f, 0.0f);
        }
    }

    // ceiling
    {
        vec3 A = vec3(-12.6f, 12.5f, 25.0f) + sceneTranslation;
        vec3 B = vec3( 12.6f, 12.5f, 25.0f) + sceneTranslation;
```

Fig. 6. Plane normal calculation.
(Source: author's code)



```
if (dist > c_minimumRayHitTime && dist < info.dist)
{
    info.dist = dist;
    info.normal = normalize((rayPos+rayDir*dist) - sphere.xyz)
    return true;
}
```

Fig. 8. Sphere normal calculation.
(Source: author's code)

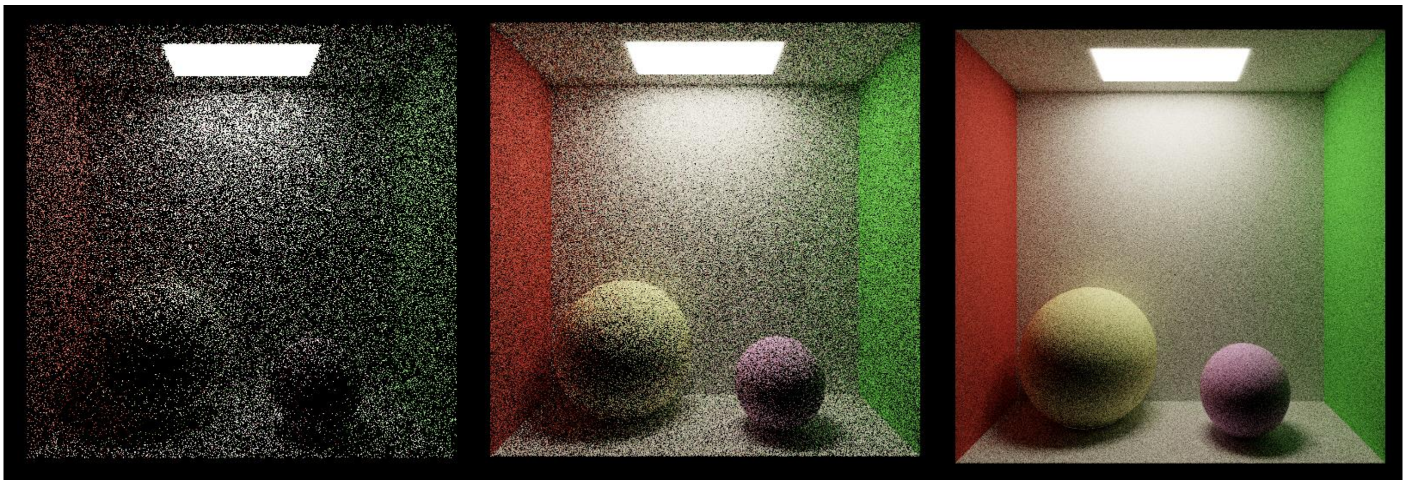


Fig. 10. Render result on a) 5 frames, b) 50 frames, and c) 500 frames.
(Source: author's code)

Let's also take a look at the plane intersection function. Here we can see how cross product used to calculate the normal of the plane. By doing cross product of vectors that start from a vertex and points to different vertices, we can get the normal vector. Note that the normal can be facing frontward or backward. If the orientation is wrong (relative to the ray), then we can simply swap the vertices and multiply the normal by -1.

On the contrary, calculating the normal for sphere is easier and does not require a cross product. Because each point on the sphere surface has the same distance to the center, we can simply do some basic vector operation to get the normal.

We can see that noise is very apparent in a naïve path tracer like this, especially on lower sample. This result can be further improved by:

1) Importance Sampling
As for now, the sampling is just a random cosine-weighted ray in the normal oriented hemisphere. But better methods have been developed such as direct light sampling, or even a different sampling method for different situations. These kinds of sampling methods are called importance sampling, because they sample what is important in the scene.

2) Multiple Importance Sampling
It is possible to do multiple importance sampling at once by doing multiple important sampling (MIS). However, you need to be careful when doing math to avoid energy loss or gain.

3) Increasing sample count
The easiest way to get better quality render is by increasing the sample count per pixel. However, the performance will be slower.

V. Conclusion

In this paper, we explored how vectors and their operations play a fundamental role in simulating light behavior in path tracing for realistic image rendering. By leveraging vector mathematics, such as dot products for light reflection and cross products for surface normals, we demonstrated how light interactions with surfaces can be accurately modeled. These techniques allow us to approximate complex optical phenomena, including diffuse, enabling photorealistic visuals in computer graphics. While path tracing simplifies light transport by treating light as rays, it effectively captures many real-world behaviors through vector-based calculations, making it a powerful and practical approach for rendering realistic images.

VI. Appendix

The source code for the path tracer code used in this paper can be found here: Pathtracer Algeo

VII. Acknowledgment

The author would like to express his gratitude to God Almighty who has given his blessings, so that the author can complete the paper entitled "Simulating Light Behavior using Vector through Path Tracing for Realistic Image Rendering" which was completed on time. The author would also like to thank Dr. Judhi Santoso, M.Sc. and Arrival Dwi Sentosa, S.Kom., M.T. as the lecturer in charge of the IF2123 Geometric and Linear Algebra Class 3 course for the guidance and teaching that has been carried out in this class. The author would also like to thank Mr. Dr. Ir. Rinaldi Munir, MT. as one of the lecturers in Geometric and Linear Algebra who provided references and learning resources for Geometric and Linear Algebra Mathematics through his website. Finally, the author would like to thank his parents, family, and all parties who helped the author in completing this paper.

References

[1] Immel, David S.; Cohen, Michael F.; Greenberg, Donald P. (1986). "A radiosity method for non-diffuse environments". Accessed on 2 January 2025.
[2] Kajiya, James T. (1986). "The rendering equation". Accessed on 2 January 2025.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 Desember 2024

[illegible]

Fachriza Ahmad Setiyono 13523162