Solving a rendering equation using the Monte Carlo method

1st Severyn Shykula
UCU APPS
Computer Science
Lviv, Ukraine
shykula.pn
@ucu.edu.ua

2nd Petro Prokopets
UCU APPS
Computer Science
Lviv, Ukraine
prokopets.pn
@ucu.edu.ua

3rd Mykola Vysotskyi
UCU APPS
Computer Science
Lviv, Ukraine
vysotskyi.pn
@ucu.edu.ua

4th Nazar Tkhir
UCU APPS
Computer Science
Lviv, Ukraine
tkhir.pn
@ucu.edu.ua

5th Mykhailo Moroz
ZibraAI, mentor
Lead 3D Research engineer
Kyiv, Ukraine
michael08840884
@gmail.com

I. Introduction

Solving a rendering equation using the Monte Carlo method is a computational approach in computer graphics to simulate the complex process of light transport within a scene. The rendering equation describes the distribution of light in a three-dimensional environment. It accounts for how light interacts with surfaces, materials, and the surrounding medium, making it a fundamental equation in computer graphics for realistic image synthesis. The Monte Carlo method, in this context, involves using random sampling techniques to approximate the solution to the rendering equation. Instead of trying to solve the equation analytically, which can be extremely challenging due to its complexity, Monte Carlo methods rely on statistical sampling to estimate the integral of the rendering equation.

II. Concepts for familiarization

- Normalize
  If $\mathbf{v} = \sqrt{v_x^2 + v_y^2 + v_z^2}$ is a vector, then its normalized version, denoted as $\hat{v}$, is given by:

$$\hat{v} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{\mathbf{v}}{\sqrt{v_x^2 + v_y^2 + v_z^2}}$$

- Dot product
  The dot product of two vectors $\mathbf{U}$ and $\mathbf{V}$, denoted as

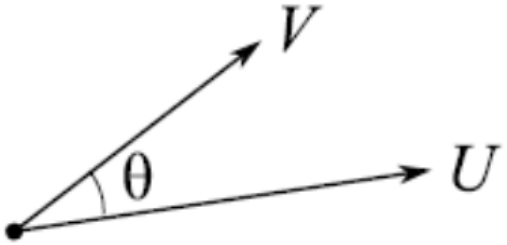

Fig. 1. Visualisation of dot product

$\mathbf{U} \cdot \mathbf{V}$, is defined as:

$$\mathbf{U} \cdot \mathbf{V} = u_1v_1 + u_2v_2 + \cdots + u_nv_n = \|\mathbf{U}\|\|\mathbf{V}\| \cos\theta$$

In this project, we will use this to determine the angle through the cosine since the cosine makes it unambiguous.

- Cross product
  The cross product of two vectors $\mathbf{U}$ and $\mathbf{V}$, denoted as

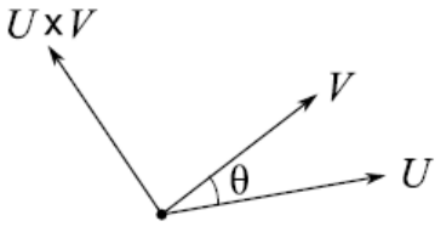

Fig. 2. Visualisation of cross product

$\mathbf{U} \times \mathbf{V}$, is defined as:

$$\mathbf{U}\times\mathbf{V} = (u_2v_3 - u_3v_2)\,\mathbf{i} + (u_3v_1 - u_1v_3)\,\mathbf{j} + (u_1v_2 - u_2v_1)\,\mathbf{k}$$

We will use this for finding normal vector as when a ray hits a glass surface, the ray tracer must determine if it is entering or exiting the glass to compute the refraction ray.

- Linear Operators Theory
  Linear operators act on functions like matrices act on vectors or discrete representations.

$$h(u) = (M \circ f)(u)$$

Here M is linear operator, f and h are functions of u. Basic linearity relation holds:

$$M \circ (af + bg) = a(M \circ f) + b(M \circ g)$$

Here a and b are scalars, f and g are functions. Example with integration:

$$(K \circ f)(u) = \int k(u, v)f(v)dv$$

III. Implementation

A. Main idea

As was mentioned, path tracing – a Monte Carlo method in computer graphics, is utilized for rendering images of three-dimensional scenes. Essentially, the algorithm involves integrating all illuminance arriving at a specific point on an object's surface. Subsequently, this illuminance is modulated by a surface reflectance to calculate the proportion directed towards the viewpoint camera. This integration process is iterated for each pixel in the output image.

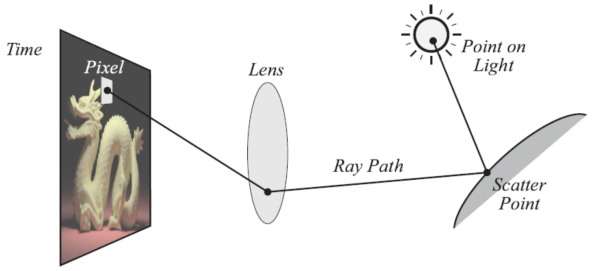

Fig. 3. Visualisation of Path Tracing. Image from [6].

B. Rendering equation

The Rendering equation of Path Tracing states that the quantity of light intensity exiting a specific point in a particular direction equals the sum of the emitted light intensity from that point in that direction, along with the incoming light intensity from other sources that are subsequently dispersed in that direction.

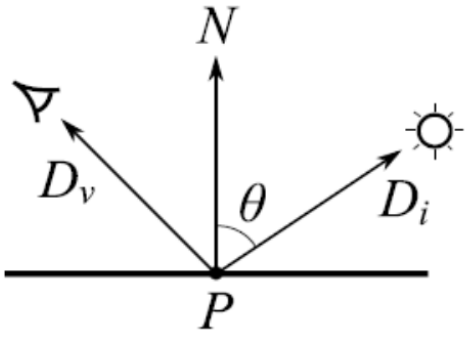

Fig. 4. Visualisation in terms of Rendering Equation, [6]

$$L_r(x, \omega_r) = L_e(x, \omega_r) + \int_{\Omega} L_r(x', -\omega_r) f(x, \omega_r, \omega_i) \cos\theta_i d\omega_i$$

Rendering Equation for Path Tracing

- $L_r(x, \omega_r)$: This represents the radiance leaving point x in the direction ω_r (unknown). Radiance is the amount of light flowing through an infinitesimal area in a particular direction. Here, ω_r is the direction in which the light is being observed or measured.
- $L_e(x, \omega_r)$: This term represents the emitted radiance at point x in the direction ω_r (known). In simpler terms, it's the light that is directly emitted by the surface at point x and is observable in the direction ω_r. This could be light emitted by a light source, such as a light bulb or the sun, or it could be light emitted by the surface itself, such as glowing materials.
- $\int_{\Omega} L_r(x', -\omega_r) f(x, \omega_r, \omega_i) \cos\theta_i d\omega_i$: This term represents the reflected radiance received at point x from all other points in the scene. Let's break it down further:
  - x': Represents all points in the scene that may contribute to the reflected light at point x.
  - $-\omega_r$: Represents the incoming light direction, which is opposite to the direction of observation ω_r.
  - $L_r(x', -\omega_r)$: This represents the radiance leaving point x' in the direction opposite to ω_r (unknown). It contributes to the reflected light observed at point x.
  - $f(x, \omega_r, \omega_i)$: Is the bidirectional reflectance distribution function (BRDF) (known). It describes how light is reflected at a surface point x given an incoming light direction ω_i and an outgoing light direction ω_r. In simpler terms, it describes the surface's reflective properties.
  - $\cos\theta_i$: Represents the cosine of the angle between the surface normal at point x and the incoming light direction ω_i (known). This factor accounts for the fact that surfaces perpendicular to the incoming light receive more light than surfaces at an angle.
  - $d\omega_i$: Represents the solid angle over which the incoming light is integrated. This essentially sums up the contributions of light from all directions.

It means that we have to calculate radiance (intensity of light), that is coming from some point on object surface (x) in some exact direction (ω_r) We can do this by adding the self-emitted radiance from x in direction ω_r (0 unless point x is a light source) and the reflected light. Here, we also must add (integrate) all of the light coming in to point x from all directions, modulated by the chance that it scatters in direction ω_r.

C. Solving equation

We will solve the equation described above by solving Fredholm Integral Equation of second kind. It is repre-

sented as:

$$\phi(x) = f(x) + \lambda \int_a^b K(x,t)\phi(t)\,dt$$

where $\phi(x)$ is the unknown function, $f(x)$ is a given function, $K(x,t)$ is the kernel of the integral equation, and λ is a parameter. In our case $\phi(x)$ is $L_r(x,\omega_r)$, $f(x)$ is $L_e(x,\omega_r)$, $K(x,t)$ is $f(x,\omega_r,\omega_i)\cos\theta_i$, what is Light Transport operator (it describes how light interacts with surfaces (BRDF) and is transported throughout a scene, essential for simulating realistic lighting in computer graphics) and $\phi(t)$ is $L_r(x',-\omega_r)$. So, we can rewrite our equation in the following way:

$$L = L_e + T \circ L$$

Where L – vector of values that we are trying to find, L_e – vector of radiance values of points (known), and T – light transport matrix. Here $T \circ L$ is:

$$(T \circ L)(x,\omega_r,\omega_i) \equiv \int_\Omega L_r(x',-\omega_r) f(x,\omega_r,\omega_i)\cos\theta_i d\omega_i$$

Now, we can solve this system:

$$L = L_e + T \circ L$$

$$IL - TL = L_e$$

$$(I - T)L = L_e$$

$$L = (I - T)^{-1} L_e$$

Since I represents the identity matrix, we have:

$$L = (I - T)^{-1} L_e$$

Using the Neumann series expansion for $(I - T)^{-1}$:

$$L = (I + T + T^2 + T^3 + \dots)L_e$$

$$L = (L_e + L_e T + L_e T^2 + L_e T^3 + \dots)$$

This series converges, and the n-th term corresponds to n bounces of light. It means that E – emission directly from light sources, EK – direct illumination of surface, EK^2 – one indirect ray bounce, EK^3 – two indirect ray bounce, and so on.

D. Monte Carlo method

As we use the Monte Carlo method, we should use the next formula:

$$L_r(x,\omega_r) = L_e(x,\omega_r) + \frac{1}{N}\sum_{i=1}^{N} \frac{L_r(x',-\omega_r) f(x,\omega_r,\omega_i)\cos\theta_i}{p(\omega_i)}$$

The equation represents the rendering equation, which describes how light interacts with surfaces in a scene.
$L_r(x,\omega_r)$ denotes the radiance leaving point x in direction ω_r. This is the outgoing radiance from a surface point in a particular direction.
$L_e(x,\omega_r)$ represents the emitted radiance at point x in direction ω_r. This term accounts for any light emitted directly from the surface.
The integral term $\int_\Omega L_r(x',-\omega_r) f(x,\omega_r,\omega_i)\cos\theta_i d\omega_i$ accounts for indirect lighting. Here, $L_r(x',-\omega_r)$ is the incoming radiance from a neighboring point x' in the direction opposite to ω_r, $f(x,\omega_r,\omega_i)$ is the bidirectional reflectance distribution function (BRDF) describing how light is reflected from the surface, $\cos\theta_i$ accounts for the angle between the incident light direction ω_i and the surface normal, and $d\omega_i$ represents the solid angle over which we integrate.
In the Monte Carlo method, we approximate this integral by taking multiple samples (N) of the incoming radiance and averaging the results. Each sample contributes $\frac{L_r(x',-\omega_r) f(x,\omega_r,\omega_i)\cos\theta_i}{p(\omega_i)}$, where $p(\omega_i)$ is the probability density function used for sampling the incoming light direction. Then, we average numerous samples for each pixel to produce a smoother image. Based on the above, we should implement the next algorithm.

```
for each pixel (i,j) do
    Vec3 C = 0
    for (k=0; k < samplesPerPixel; k++) do
        Create random ray in pixel:
            Choose random point on lens P_lens
            Choose random point on image plane P_image
            D = normalize(P_image − P_lens)
            Ray ray = Ray(P_lens, D)
        castRay(ray, isect)
        if the ray hits something then
            C += radiance(ray, isect, 0)
        else
            C += backgroundColor(D)
        end if
    end for
    image(i,j) = C / samplesPerPixel
end for
```

Fig. 5. Path Tracing Main loop, [7]

E. Bidirectional Reflectance Distribution Function (BRDF)

The Bidirectional Reflectance Distribution Function (BRDF) characterizes how a surface reflects light, considering both the illumination and viewing angles. It is influenced by various factors, including the surface's structural and optical properties, such as shadowing, scattering, transmission, absorption, and emission. Additionally, the BRDF depends on wavelength and is influenced by facet orientation distribution and density.
In remote sensing, BRDF plays a crucial role in correcting view and illumination angle effects, deriving albedo, land cover classification, cloud detection, and atmospheric correction. It serves as a fundamental boundary condition for radiative transfer problems in the atmosphere, making it pertinent for climate modeling and energy budget investigations.

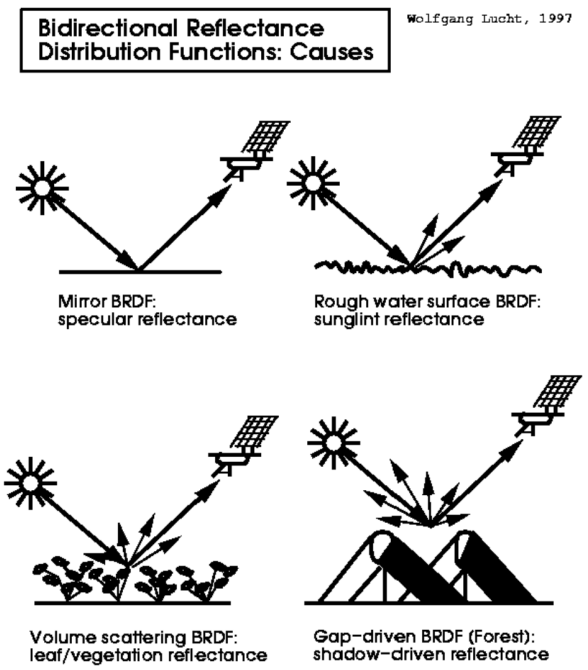

Fig. 6. Causes of BRDF, from Wolfgang Lucht, 1997, [4]

The modern definition of this function is as follows:

$$f_r(\omega_i, \omega_o) = \frac{dL_r(\omega_o)}{dE_i(\omega_i)} = \frac{dL_r(\omega_o)}{L_i(\omega_i)\cos\theta_i d\omega_i}$$

where L is luminance, E is illuminance, and θ_i is the angle between the direction ω_i and the normal n.

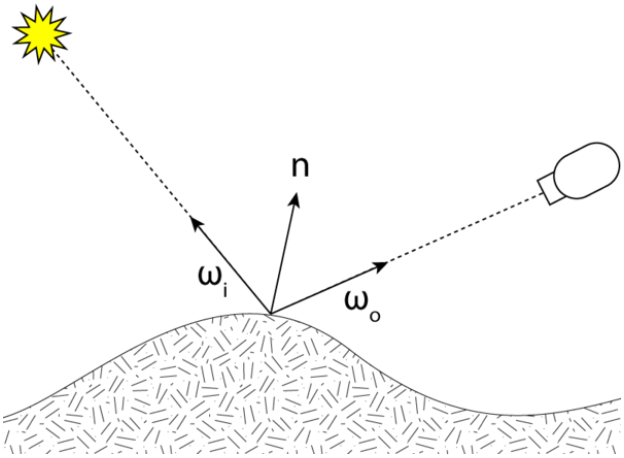

Fig. 7. BRDF formula visualization, [2]

F. Rays and shapes

- Ray A ray is a parametric line with an origin (o) and a direction (d). A point along the ray can be defined using a parameter, t:

$$P(t) = o + td$$

The core routines of the ray tracer intersect rays with geometric objects.

- Sphere We define a sphere with its center ($C(c_x, c_y, c_z)$) and radius (r). It's equation:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$$

In vector form:

$$(P - C) \cdot (P - C) = r^2$$

Where P are points of intersections of ray and sphere with center in point C.

G. Intersection handling

For finding the points, mentioned above, we should substitute ray equation from the sphere vector equation:

$$(o + td - C) \cdot (o + td - c) = r^2$$

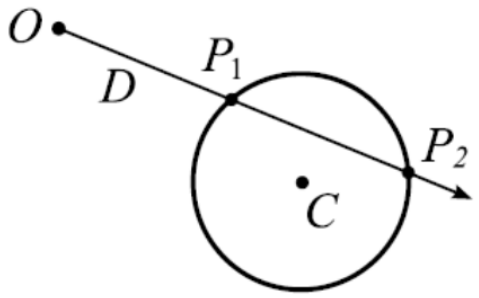

Fig. 8. Ray P intersects circle (C, r), [2]

$$(D \cdot D) \cdot t^2 + 2D \cdot (o - C) \cdot t + (o - C) \cdot (o - C) - r^2 = 0$$

Then we solve this equation for t and get:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Where $a = (D \cdot D)$
$b = 2D \cdot (o - C)$ and
$c = (o - C) \cdot (o - C) - r^2$

H. Möller–Trumbore intersection algorithm

The ray is determined by an origin point **O** and a directional vector $\vec{v}$. Any point along the ray can be represented as $\vec{r}(t) = \mathbf{O} + t\vec{v}$, where t varies from zero to infinity. The triangle is described by three vertices, denoted v_1, v_2, and v_3. The plane containing the triangle, necessary for computing the intersection of the ray with the triangle, is defined by a point on the plane, such as v_1, and a vector orthogonal to all points on that plane, determined by the cross product of the vectors from v_1 to v_2 and from v_1 to v_3:

$$\vec{n} \cdot (P_1 - P_2) = 0,$$

where $\vec{n} = (v_2 - v_1) \times (v_3 - v_1)$, and P_1 and P_2 represent any points on the plane.

- The condition of parallelism of a ray with a triangle Initially, determine whether the ray intersects the plane containing the triangle, and if so, identify the coordinates of the intersection. The ray fails to intersect the plane only when its direction vector aligns parallelly with the plane. This scenario occurs when the dot product between the ray's direction vector and the plane's normal vector equals zero. Conversely, if the dot product is non-zero, the ray intersects the plane at some point, albeit not necessarily within the confines of the triangle.
- Check the intersection of the ray and the plane of the triangle outside it Using barycentric coordinates, any point within the triangle can be expressed as a combination of the triangle's vertices:

$$P = wv_1 + uv_2 + vv_3$$

where w, u, and v are coefficients that are non-negative and sum up to 1. Subsequently, w can be replaced with $1 - u - v$, yielding:

$$P = (1 - u - v)v_1 + u(v_2 - v_1) + v(v_3 - v_1)$$

Observing that $\vec{e_1} = v_2 - v_1$ and $\vec{e_2} = v_3 - v_1$ are vectors along the triangle's edges, they together span a plane. Each point on this plane can be represented as $u\vec{e_1} + v\vec{e_2}$, translated by v_1 onto the triangle's plane.
To determine u and v for a specific intersection, equate the ray expression with the plane expression and isolate the variables and constants:

$$O + tD = v_1 + u(v_2 - v_1) + v(v_3 - v_1)$$

This system of linear equations with three unknowns (t, u, and v) and three equations can be represented as a matrix-vector multiplication:

$$\begin{bmatrix} -D & (v_2 - v_1) & (v_3 - v_1) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - v_1$$

This equation has a solution when the matrix has three linearly independent column vectors in $\mathbb{R}^3$, indicating non-collinear triangle vertices and a non-parallel ray to the plane.

I. Bounding Volume Hierarchies (BVH)

Bounding Volume Hierarchies (BVHs) represent a method to expedite ray intersection computations by dividing primitives into nested sets through subdivision. Unlike spatial subdivision, which primarily divides space into disjoint sets, BVHs focus on partitioning primitives. As depicted in Figure 7, a BVH for a basic scene consists of nodes storing bounding boxes for the primitives beneath them, with the actual primitives stored in the leaves. Consequently, while traversing the tree with a ray, any segment where the ray doesn't intersect a node's bounds permits skipping the subtree beneath that node.

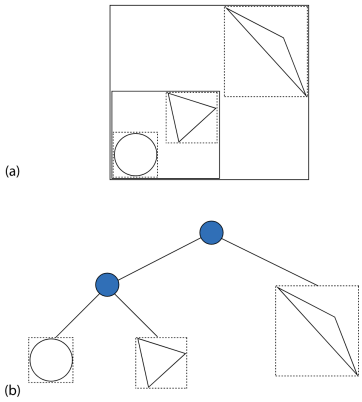

Fig. 9. (a) A small collection of primitives (b) The corresponding bounding volume hierarchy[8]

Primitive subdivision guarantees that each primitive appears just once in the hierarchy, while spatial subdivision might lead to primitive intersecting multiple regions, causing multiple intersection tests as the ray moves through. This also minimizes memory needs for hierarchy representation. In a binary BVH with one primitive per leaf, the total node count is $2n - 1$, with n primitives leading to n leaf nodes and $n - 1$ interior nodes. Fewer nodes are required if leaves hold multiple primitives. BVH construction is generally faster than kd-trees, though kd-trees often offer slightly quicker ray intersection tests but require significantly longer build times. BVHs are usually more numerically robust and less prone to missed intersections due to rounding errors than kd-trees.
We adopt a top-down approach, dividing the input set into subsets, enclosing them in chosen bounding volumes, and recursively repeating until each subset contains only one primitive (reaching leaf nodes). Although top-down methods are easy to implement and quick to construct, they typically don't yield the most optimal trees overall.

- BVH Construction BVH construction in this implementation comprises three stages. Initially, bounding details for each primitive are calculated and stored in an array for subsequent tree construction. Then, the tree is erected using the specified algorithm, resulting in a binary tree with interior nodes pointing to their children and leaf nodes referencing one or more primitives. Lastly, the tree is transformed into a more streamlined, pointerless representation for optimized rendering. For interior nodes, partitioning the collection of primitives between the two

children subtrees is necessary. With n primitives, there are typically $2(n-1)-2$ potential ways to partition them into two nonempty groups. In practical BVH construction, partitions are usually considered along a coordinate axis, resulting in approximately $3n$ candidate partitions. (Each primitive can be placed in either the first or second partition along each axis.) Here, we opt for one of the three coordinate axes to partition the primitives. We choose the axis associated with the largest extent when projecting the centroid of the bounding box for the current set of primitives. (Although an alternative would be to try all three axes and select the most optimal result, in practice, this method is effective.) This approach often yields satisfactory partitions in various scenes, as illustrated in Figure 8. The primary aim of partitioning

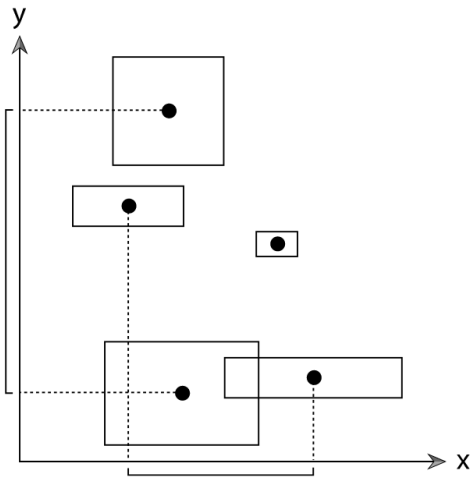

Fig. 10. Choosing the Axis along Which to Partition Primitives, [8]

here is to choose a partition of primitives with minimal overlap in the bounding boxes of the resulting sets. Excessive overlap necessitates traversing both children subtrees more frequently during tree traversal, leading to increased computational overhead compared to efficiently pruning away collections of primitives. This concept of identifying optimal primitive partitions will be further developed shortly in the discussion of the surface area heuristic. If all centroid points coincide (i.e., the centroid bounds have zero volume), recursion halts, and a leaf node containing the primitives is generated, none of the splitting methods provided are effective in this uncommon scenario.

In cases where primitives exhibit significant overlap in bounding boxes, this splitting method might struggle to divide the primitives into two distinct groups.

- Partition primitives into equally sized subsets The method divides the primitives into two equal-sized subsets. The first subset comprises the $n/2$ primitives with the smallest centroid coordinate values along the chosen axis, while the second subset consists of the remaining primitives with the largest centroid coordinate values. Although this approach can yield favorable outcomes on occasion, it performs inadequately in scenarios like the one illustrated in Figure 9. It takes a start, middle, and ending pointer as

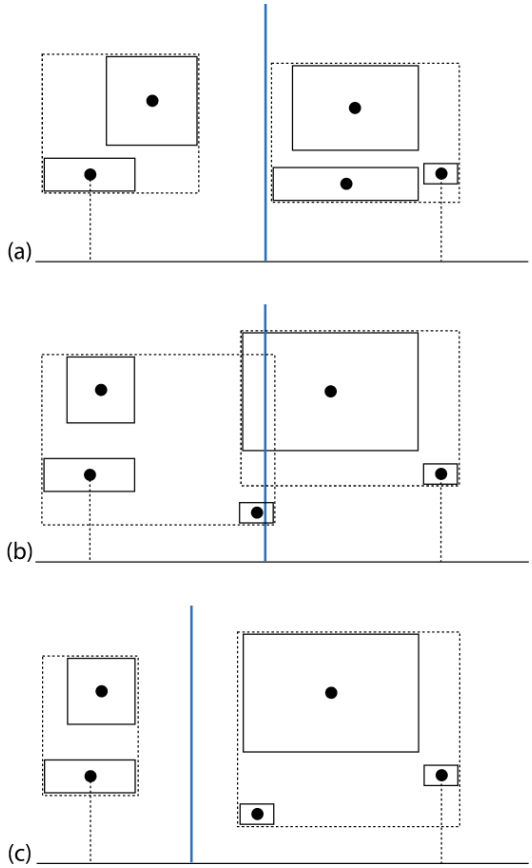

Fig. 11. (a) Splitting based on the centroid midpoint along the chosen axis (thick blue line) is effective for certain primitive distributions, as depicted. The dashed lines outline the bounding boxes of the resulting groups. (b) In distributions like this, selecting the midpoint is suboptimal, resulting in substantial bounding box overlap. (c) Splitting the same primitives along the indicated line yields smaller, non-overlapping bounding boxes, enhancing rendering performance. [8]

well as a comparison function. It orders the array so that the element at the middle pointer is the one that would be there if the array was fully sorted, and such that all of the elements before the middle one compare to less than the middle element and all of the elements after it compare to greater than it. This ordering can be done in $O(n)$ time, with n the number of elements, which is more efficient than the $O(n \log n)$ of completely sorting the array. Using all written above, we got this (Fig.10 and Fig. 11):

J. All together

- Image
  Set image width, height, image array etc.
- Camera
  Set image position, direction vector and view angles.

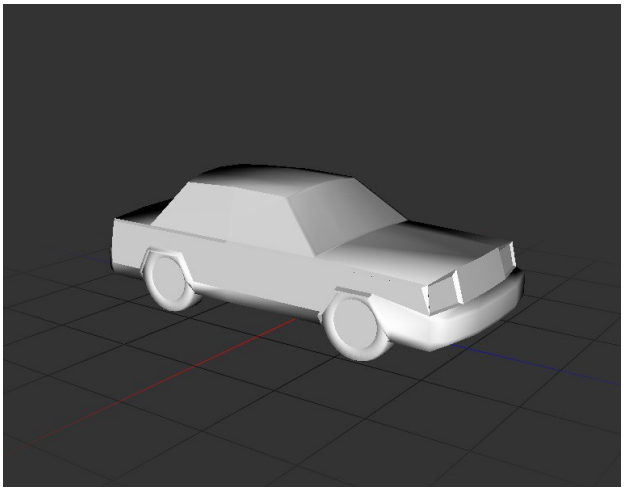

Fig. 12. Car model

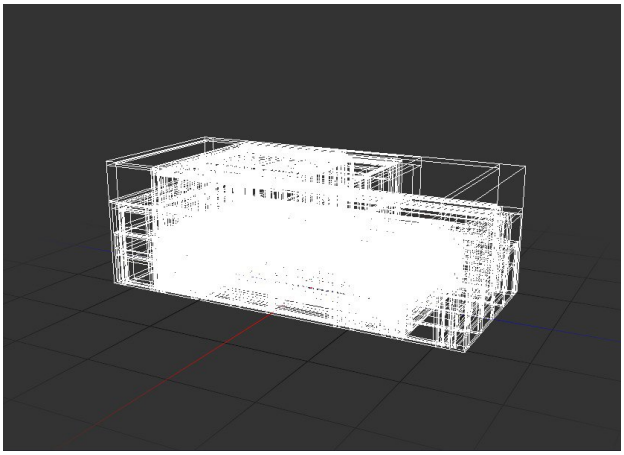

Fig. 13. Our result of BVH for car model

- Build pixels
  For each pixel, we do 2x2 subpixels. The subpixel colors will be averaged. Calculate the array index for pixel(x,y). Also, we use a Tent filter for determining the location of sample rays within the pixel.
- Radiance
  We use our radiance function to estimate radiance and add the gamma-corrected subpixel color estimate to the pixel color. Obviously, it depends on the properties of the surface that the ray intersects or bounces from.
- Recursion handling
  We stop the recursion randomly based on the surface reflectivity. We use the maximum component of the surface color and don't do stop until after depth 5.
- Mirror reflection handling
  As we know, the angle of incidence equals the angle of reflection. It's visualisation you can see at fig.7.
- Refraction handling
  When light travels from one substance or medium into another, the light waves may undergo a phenomenon known as refraction. In our case, handling means finding a solution to the next equation:

$$T = \frac{n_a(D + N(D \cdot N))}{n_b} + N\sqrt{1 - \frac{n_a^2(1 - (D \cdot N)^2)}{n_b^2}}$$

Where n_a and n_b are refractive indexes, that give us the speed of light within a medium compared to the speed of light within a vacuum. Also, we implemented Fresnel

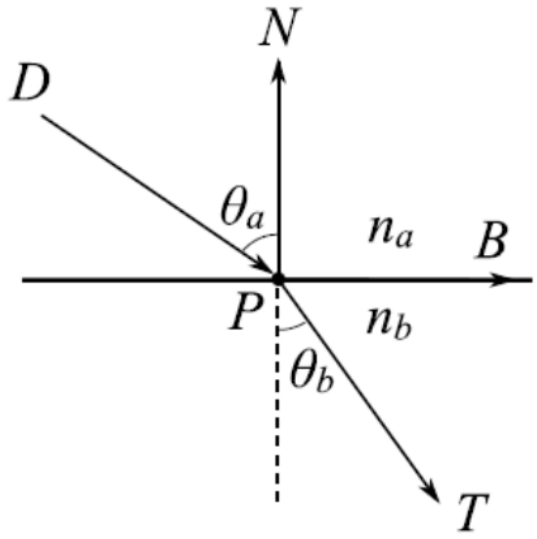

Fig. 14. Visualisation of refraction

reflectance, which handles some cases of rays behavior at the intersection of environments, like the percentage of reflected/refracted light from a glass surface based on incident angle (θ_a), reflectance at normal incidence,where $n = n_a/n_b$

$$F_0 = \frac{(n-1)^2}{(n+1)^2}$$

and reflectance at other angles:

$$Fr(\theta) = F_0 + (1 - F_0)(1 - \cos\theta)^5$$

Rules described above, we use in the following way: 2 more recursive steps if the current depth is ≤ 2 and 1 otherwise.

IV. Results

We used all the written above gave us all the necessary knowledge to get the result:

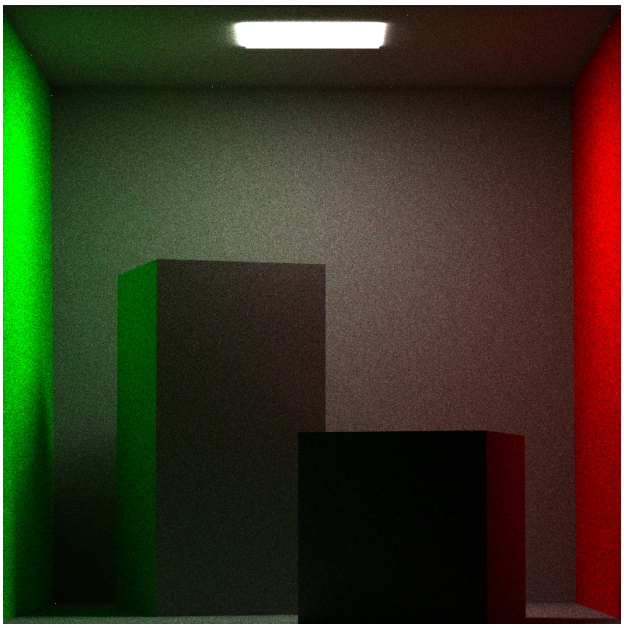

Fig. 15. Our result for Cornell box

The code of the project is available at https://github.com/mvysotskyi/path-tracing.

References

[1] Peter Shirley, R. Keith Morley, "Realistic Ray Tracing" *IEEE Transactions on Image Processing*, vol. 13, no. 9, 2004. Available online: https://www.irisa.fr/vista/Papers/2004_ip_criminisi.pdf.

[2] Chris Wynn, "An Introduction to BRDF-Based Lighting,". Available online: https://www.cs.princeton.edu/courses/archive/fall06/cos526/tmp/wynn.pdf.

[3] Tomas Moller and Ben Trumbore, "Fast minimum storage ray-triangle intersection," *Proceedings of SPIE - The International Society for Optical Engineering*, 9443, 2015. Available online: http://www.graphics.cornell.edu/pubs/1997/MT97.pdf.

[4] Wolfgang Lucht, Crystal Barker Schaaf,Member, IEEE, and Alan H. Strahler, Member, IEEE, "An Algorithm for the Retrieval of Albedo from Space Using Semiempirical BRDF Models," https://web.gps.caltech.edu/~vijay/Papers/BRDF/lucht-etal-00.pdf.

[5] Steve Marschner, Monte Carlo Solutions to Rendering Equations, Feb 2005, https://www.cs.cornell.edu/courses/cs667/2005sp/notes/10cai.pdf

[6] Rendering techniques, Suriya, Medium, https://medium.com/everythingcg/rendering-techniques-rasterization-vs-ray-tracing-vs-path-tracing-f9969b60417e

[7] Pathsy: a Simple Path Tracer on an FPGA, Tayfun Kayhan,https://tayfunkayhan.wordpress.com/2021/08/26/pathsy-a-simple-path-tracer-on-an-fpga/

[8] Physically Based Rendering, Matt Pharr, Wenzel Jakob, and Greg Humphreys, 2018 ,https://www.pbr-book.org/3ed-2018/