Concurrent Real-Time Ray Traycer

Andrii Yaroshevych*, Pavlo Kryven*, Oleksandr Shchur*
Ostap Trush* †
* *Faculty of Applied Sciences of Ukrainian Catholic University, L'viv, Ukraine*
† *Our mentor*

***Abstract*—This project aimed to create a real-time ray tracer program on the CPU, utilizing the wide possibilities for parallelization. Blinn-Phong [1] reflection model is used as a backbone of lighting effects. Ray-objects intersection testing is conducted using the Möller–Trumbore algorithm [2] for triangles and simple linear algebra operations for plains, spheres, and ellipses. Global and local view concepts are introduced to implement the 3D object sizing, rotation, and positioning. The camera obscura model is used to reproduce the properties of the camera, such as field of view and perspective. As expected, the overall conclusion is that the CPU is insufficient to perform a high-resolution real-time rendering. However, GPU rendering also had some downsides [3], so the investigated problem is still relevant.**

I. Introduction

Ray tracing is a fundamental technique in computer vision and computer graphics. The key concept of ray tracing is the behavior simulation of light in a virtual 3D environment by tracing rays of light from a virtual camera into the scene. It involves casting rays from the camera's viewpoint and calculating their intersections with objects in the scene to determine how they interact with the objects and contribute to the final image. There are also alternative ways to render images, such as rasterization [4] and ray casting [5], each with its own benefits and downsides. This project focuses on the concurrent real-time ray tracer CPU program. As the programming language, C++ was chosen. The auxiliary libraries used were SDL2 (handling the window creation and management) and GLM (the mathematical backbone of all computations). The primary purpose of the program is to render static scenes; however, with minimal modifications, it is easy to achieve the dynamic.

II. Architecture and implementation details

The program architecture consists of several processing stages. All of them, except the scene setup, are repeated in each frame, as they are required to produce the updated image.

1) **Scene setup.** In this stage, the 3D objects in the scene are defined using linear algebra operations such as matrix transformations, which can be used to position, orient, and scale the objects in the scene. A few object types are implemented: sphere, plane, triangle, and cube. The files with a ".obj" format are supported, allowing meshes to be used. The camera position and orientation are also defined using linear algebra operations.
2) **Ray generation.** For each pixel in the image, a ray is generated from the camera position and direction. The camera obscura model is used, with a screen located some distance from the camera, allowing for setting the field of view.
3) **Intersection testing.** The generated ray is tested for intersection with each object in the scene, using different methods depending on the object type. For spheres, ellipses, and planes, the transformation matrices and elementary mathematical operations are used. Testing mesh and triangle intersections additionally involves the Möller–Trumbore algorithm. If the intersection happens, the result of this step is the intersection point and normal vector of the intersected object.
4) **Lighting and shading.** After obtaining the intersection point and normal vector of an object, the lighting and shading effects are calculated. For this purpose, the Blinn-Phong model is used, involving utilizing linear algebra operations to determine the direction and intensity of the light rays, along with considering the material properties of the object.
5) **Color computation.** Based on the previous stage results, the color of the object at the intersection point is computed.
6) **Image rendering** The computed colors for each object in the scene are combined to generate the final image, which is displayed on the screen using the SDL2 library.

Such program structure allows the utilization of embarrassing parallelism in its functioning. It is implemented using the Threading Building Block function tbb::parallel_for.

For optimization purposes, the project conventions were designed, such as normals are always normalized and light rays are kept in the direction of the light source.

III. Theoretical foundation for used algorithms

A. Vector operations

Vectors are used to represent the position and orientation of objects in the scene, as well as the direction of the rays being traced. Vector operations, such as dot product and cross product, are used to calculate the angles, distances, and other properties needed for intersection tests and lighting calculations.

B. Matrix transformations

Matrices are used to represent transformations, such as rotation, translation, and scaling, that can be applied to 3D objects to change their orientation, position, and size in the scene. They can also be used to define the position and orientation of the camera in the scene. Matrices are also used to transform objects from their local coordinate systems to the global coordinate system of the scene, as well as to transform rays from the camera's coordinate system to the global coordinate system.

C. Intersection tests

In ray tracing, the algorithm needs to determine which objects in the scene a ray intersects with. This is done by performing intersection tests between the ray and each object in the scene. To perform these tests efficiently, linear algebra techniques such as dot product, cross product, and matrix transformations are used to calculate the position and orientation of the objects and the direction and position of the rays.

The concrete algorithm used in this project is the Möller-Trumbore algorithm [2] for checking intersections with triangles.

This method is based on the idea of representing a triangle as a plane in 3D space and then checking whether the ray intersects that plane. If the ray intersects the plane, the algorithm then checks whether the point of intersection lies within the triangle itself. The mathematical foundation is the following:

Point $T(u, v)$, on a triangle is given by:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2,$$

where (u, v) are the barycentric coordinates which must fulfill $u \geq 0, v \geq 0$ and $u + v \leq 1$. Computing the intersection between the ray, $R(t)$, and the triangle, $T(u, v)$, is equivalent to $R(t) = T(u, v)$, which yields:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2.$$

Rearranging the terms gives:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0.$$

This means the barycentric coordinates (u, v) and the distance, t, from the ray origin to the intersection point can be found by solving the linear system of equations above.

The above can be thought of geometrically as translating the triangle to the origin and transforming it to a unit triangle in $y\&z$ with the ray direction aligned with x, as illustrated in Fig 1 (where $M = [-D, V_1 - V_0, V_2 - V_0]$ is the matrix in the previous equation).

Denoting $E_1 = V_1 - V_0, E_2 = V_2 - V_0$ and $T = O - V_0$, the solution to the equation is obtained by using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D_1, E_1, E_2|} \begin{bmatrix} T, E_1, E_2 \\ -D, T, E_2 \\ -D, E_1, T \end{bmatrix}.$$

We know that $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$. The equation above could, therefore be rewritten as:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix},$$

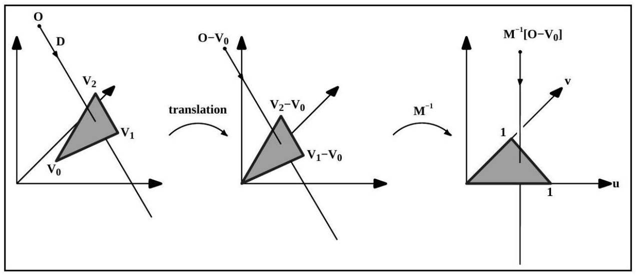

Figure 1: Translation and change of base of the ray origin [2].

where $P = (D \times E_2)$ and $Q = T \times E_1$. To speed up the computations, one can save and reuse these factors.

D. Lighting model

In our project, we used the Blinn–Phong model. This lighting model is a widely used shading model in computer graphics that approximates the interaction of light with surfaces. It calculates the intensity of light reflected from a surface based on its material properties and the position of the viewer.

The model takes into account three components: ambient, diffuse, and specular lighting. Each component contributes to the final color of the surface.

- Ambient Lighting: The ambient component represents the overall ambient light in the scene that uniformly illuminates all objects. It provides a base level of illumination regardless of the surface's orientation or position relative to light sources. The ambient term is typically a constant color value multiplied by the ambient reflection coefficient of the material.
- Diffuse Lighting: The diffuse component accounts for light that is scattered equally in all directions by a surface. It depends on the angle between the surface normal and the direction of the incoming light. Surfaces facing the light source directly receive more light, resulting in a brighter appearance, while surfaces facing away receive less light. The diffuse term is calculated by multiplying the incoming light intensity, the diffuse reflection coefficient of the material, and the dot product between the surface normal and the light direction.
- Specular Lighting: The specular component simulates the reflection of light off shiny or glossy surfaces. It creates highlights or specular reflections that appear as bright spots on the surface. The specular term is determined by the dot product between the reflection vector and the viewer's direction, raised to a power defined by the shininess or specular exponent of the material.

In Phong shading, one must continually recalculate the dot product $R \cdot V$ between a viewer (V) and the beam from a light source (L) reflected (R) on a surface.

The Blinn-Phong model is similar but approaches the specular model slightly differently. Here, we use a so-called halfway vector instead of a reflection vector, calculated as a unit vector in the middle between the view direction and the light direction. The less difference between this halfway

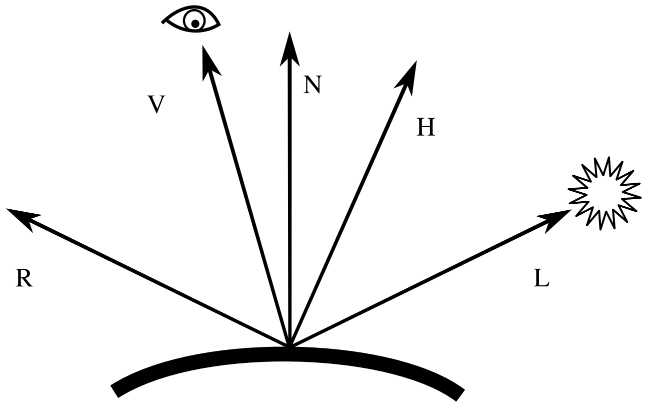

Figure 2: Vectors for calculating Phong and Blinn–Phong shading [6].

vector and the surface's normal vector, the higher the specular contribution.

One calculates a halfway vector between the viewer and light-source vectors as:

$$H = \frac{L+V}{\|L+V\|}.$$

$R \cdot V$ can be replaced with $N \cdot H$, where N is the normalized surface normal. In the above equation, L and V are both normalized vectors, and H is a solution to the equation $V = P_H(-L)$, where P_H is the Householder matrix that reflects a point in the hyperplane that contains the origin and has the normal H.

The Phong model always gives round reflections for a flat surface, but the Blinn–Phong reflections are more like an ellipse when the view angle is large. This can be compared to the case where the sun is reflected in the sea close to the horizon.

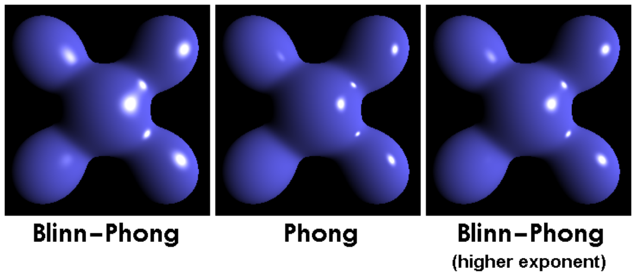

Figure 3: Blinn-Phong comparison. [6]

IV. Performance Analysis and Results

While working on the project, many tests were conducted to find the fastest and most suitable solutions for problems that arose. Some methods were rejected before implementation, and others were removed as such, which proved to be ineffective for our applications.

A. Parallelization

While parallelizing our application, the various tools were tested. The first implementation used the custom thread pool, which, as expected, gave the worst performance. It also introduced the problems with vertical synchronization (aka VSync). The next solution tested was the boost::asio::thread_pool provided by the Boost library, and it proved to be a more effective tool. However, the best results were achieved with tbb::parallel_for by TBB. You can see our results for a test scene in Table I. Note that the program pipeline generally does not depend on the scene conditions, so we consider this single case sufficient to analyze and compare various tools' performance.

Table I: Tests results: 16 logical threads.

Method	FPS	Frame Time, ms
sequential	19.2	52
naive thread pool	55.2	18.1
boost thread pool	60.9	16.4
tbb::parallel_for	62.0	16.1

One can now apply the acceleration coefficient formula, where $L(x)$ is the execution time on x threads:

$$S(n) = \frac{L(1)}{L(n)}.$$

Then, the parallelization efficiency coefficient is calculated by:

$$E(n) = \frac{S(n)}{n}.$$

The results of applying those results on test data can be seen in Table II ($L(1) = 52ms$).

Table II: Parallelization efficiency coefficient: 16 logical threads

Method	S(16)	E(16)
naive thread pool	2.87	0.180
boost thread pool	3.17	0.198
tbb::parallel_for	3.23	0.202

B. Implementation details

Initially, the shared pointers were used for object manipulation. Switching to raw pointers resulted in an x1.5 performance boost.

C. Rejected methods

- While researching the topic, a few different lighting models were discussed. Eventually, we decided to discard all probabilistic methods. Performance reasons mainly conditioned this decision, as the real-time CPU ray tracer cannot afford to send sufficient rays to prevent random grain effects.
- Gamma correction was implemented and tested. However, it significantly decreased performance up to three times, so the team decided to turn off this feature.

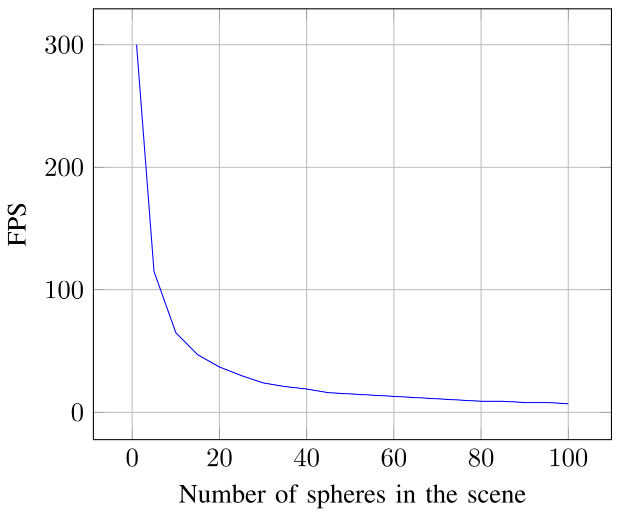

Figure 4: Plots of FPS depending on the number of spheres on the scene.

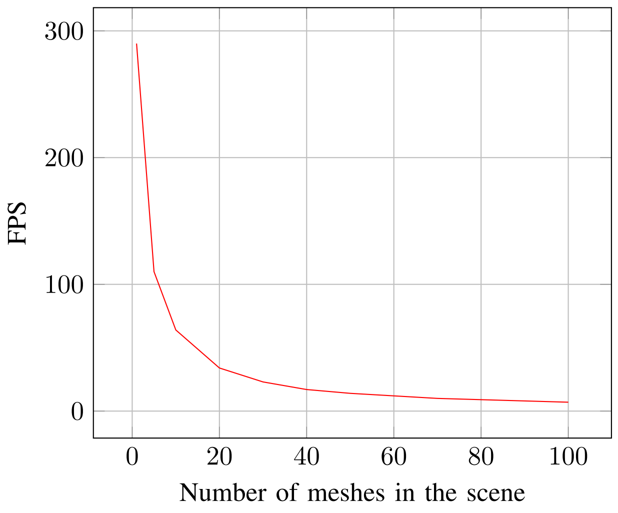

Figure 5: Plots of FPS depending on the number of meshes on the scene.

D. Results

Figure 4 shows the results of performance tests with spheres on the scene.

Essentially, the same can be seen in Fig. 5, but for meshes, where each mesh is just a single triangle.

Important to note that all tests were conducted in 640x480 resolution.

V. Conclusion

In this project, we explored the common approaches in ray tracing and its implementation on the CPU. The conclusion is that it is possible to use CPU for ray tracing, using further optimizations in this field. However, it's still much slower than any graphics, computed on GPU, especially in recent years.

The code of the project is available at https://github.com/codefloww/Ray-Tracer.

References

[1] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, p. 192–198, jul 1977. [Online]. Available: https://doi.org/10.1145/965141.563893

[2] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 7–es. [Online]. Available: https://doi.org/10.1145/1198555.1198746

[3] I. Wald, T. Purcell, J. Schmittler, C. Benthin, and P. Slusallek, "Realtime ray tracing and its use for interactive global illumination," *Eurographics State of the Art Reports*, p. 12, 01 2003.

[4] M. Worboys and M. Duckham, *GIS, a Computing Perspective*. CRC Press, 01 2004.

[5] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, p. 37–45. [Online]. Available: https://doi.org/10.1145/1468075.1468082

[6] "Blinn-Phong reflectance model," https://knowww.eu/nodes/59b8e93cd54a862e9d7e415a, [Accessed 16-May-2023].