Parallelization in Ray Tracing for Graphics

Mihail Gjorgjiev, Computer Science Student[1]

[1]Ss. Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering, North Macedonia
[1]E-mail: mihail.gjorgjiev.1@students.finki.ukim.mk

Abstract— **Ray tracing, a cornerstone in computer graphics, is a rendering technique that simulates the behavior of light to generate visually realistic images. This method, rooted in principles of physics and optics, traces the paths of rays of light as they interact with virtual scenes, accounting for effects such as reflections, refractions, shadows, and complex lighting conditions. Over the years, research in ray tracing has evolved, yielding diverse approaches to enhance efficiency and achieve greater visual fidelity.**

I. INTRODUCTION

Ray tracing stands as a fundamental and powerful technique in the domain of computer graphics, offering a comprehensive approach to simulating the complex interplay of light and surfaces to generate high-fidelity visual representations. Rooted in principles of physics and optics, ray tracing has become a cornerstone for applications requiring photorealistic rendering, spanning fields such as entertainment, scientific visualization, and virtual environments. At its core, ray tracing is an algorithmic method for generating images by simulating the behavior of light rays as they traverse a virtual scene. The foundational concept is derived from the rendering equation, introduced by James T. Kajiya in 1986, which serves as the theoretical framework for understanding the transport of light in a scene. The equation encapsulates the recursive nature of light interactions, encompassing effects such as diffuse reflection, specular reflection, and refraction.

The ray tracing process unfolds in several stages. It commences with the generation of rays emanating from the virtual camera, commonly referred to as primary rays. These rays traverse the scene, and upon encountering surfaces, secondary rays are spawned to account for reflections and refractions. The simulation of light interaction at each point of intersection involves complex computations, considering material properties, surface geometry, and the incoming illumination. One of the primary strengths of ray tracing lies in its ability to authentically simulate phenomena such as shadows, reflections, and global illumination. Shadows are accurately rendered by tracing rays from the point of intersection toward light sources, determining occlusion by other objects. Reflections are achieved by recursively tracing rays in the reflected direction, creating realistic mirrored surfaces. Global illumination, a critical aspect for photorealism, is addressed through techniques like path tracing and photon mapping, which simulate the intricate paths of light in the scene.

II. RELATED WORK

Parallelization in ray tracing for computer graphics plays a pivotal role in addressing the increasing demands for real-time rendering and handling complex scenes. This paper explores the key strategies and challenges associated with parallelizing various stages of the ray tracing pipeline. From parallel ray generation and intersection testing to shading and lighting computations, we examine techniques for efficient workload distribution across multiple processors or GPUs. We discuss the impact of parallelization on rendering performance, scalability, and its role in advancing the capabilities of ray tracing for interactive applications.

Research similar to this topic was conducted by Turner Whitted' in 1980 when he presented "An Improved Illumination Model for Shaded Display,"[1] which introduced ray tracing for realistic image synthesis. This foundational contribution marked the beginning of simulating complex light interactions in computer graphics.

In 1986, James T. Kajiya released his groundbreaking paper - "The Rendering Equation," a fundamental concept shaping the understanding of light transport in computer graphics. This work laid the theoretical foundation for subsequent advancements in global illumination techniques and ray tracing.[2].

Henrik Wann Jensen's 1996 work on "Photon Mapping"[3] revolutionized global illumination in ray tracing by introducing a technique that simulates the behavior of photons. This method, involving tracing photons through a scene, significantly advanced the realism achievable in rendered images.

The introduction of the "Metropolis Light Transport"[4] algorithm by Veach and Guibas is a key advancement in the quest for realistic global illumination. This probabilistic algorithm significantly improved the efficiency of simulating complex lighting interactions, influencing subsequent developments in rendering algorithms.

In 2001, Ingo Wald and colleagues contributed to real-time graphics with "Real-Time Ray Tracing for Interactive Global Illumination,"[5] presenting techniques for accelerating ray tracing on modern graphics hardware. This work played a crucial role in bringing real-time ray tracing closer to practical application.

NVIDIA's 2010 work on "OptiX: A General Purpose Ray Tracing Engine"[6] revolutionized GPU-accelerated ray tracing, allowing developers to harness the parallel processing power of GPUs for real-time applications. OptiX has become a significant tool for researchers and developers exploring real-time ray tracing.

Intel's 2014 contribution, "Embree: A High-Performance Ray Tracing Kernel,"[7] focuses on optimizing ray tracing for modern CPU architectures. Embree provides a framework that enhances the efficiency of ray tracing implementations, contributing to advancements in computational performance.

III. SYSTEM ARCHITECTURE

The design and implementation of a parallelized ray tracing program require careful consideration of various components, interactions, and optimizations. This solution architecture provides a structured blueprint for creating a performant and interactive ray tracing application. The primary objective is to simulate the interaction of light rays with objects in a scene while leveraging parallel processing for enhanced efficiency.

1) Problem Context: Ray tracing is a widely-used technique in computer graphics to create realistic images by simulating the behavior of light in a scene. Achieving real-time ray tracing for complex scenes can be computationally intensive. This architecture aims to address this challenge by introducing parallelization.
2) Objectives: Develop an efficient ray tracing algorithm capable of rendering scenes interactively. Leverage parallel processing to distribute the computational workload, improving overall performance.
3) Key Components:
    - Ray Tracing: Responsible for calculating ray-object interactions. Utilizes object properties and parallelization to efficiently compute intersections.
    - Parallelization: Manages the distribution of work among multiple threads, dividing the screen into segments for parallel processing. Aims to exploit the full potential of multi-core processors.
    - Rendering: Fills the screen and orchestrates the execution of the parallelized ray tracing algorithm. Ensures a seamless visual representation of the simulated scene.
4) Flexibility and Extensibility: The architecture is designed to accommodate future enhancements or modifications without disrupting existing functionality. Components are modular and can be extended or replaced as needed.

The solution architecture for the parallelized ray tracing program is crafted to guide the development process through carefully delineated phases. Each phase plays a crucial role in achieving the overarching goals of efficiency, modularity, and user interaction within the realm of computer graphics. The following introduction provides a brief overview of the key phases outlined in the architecture:

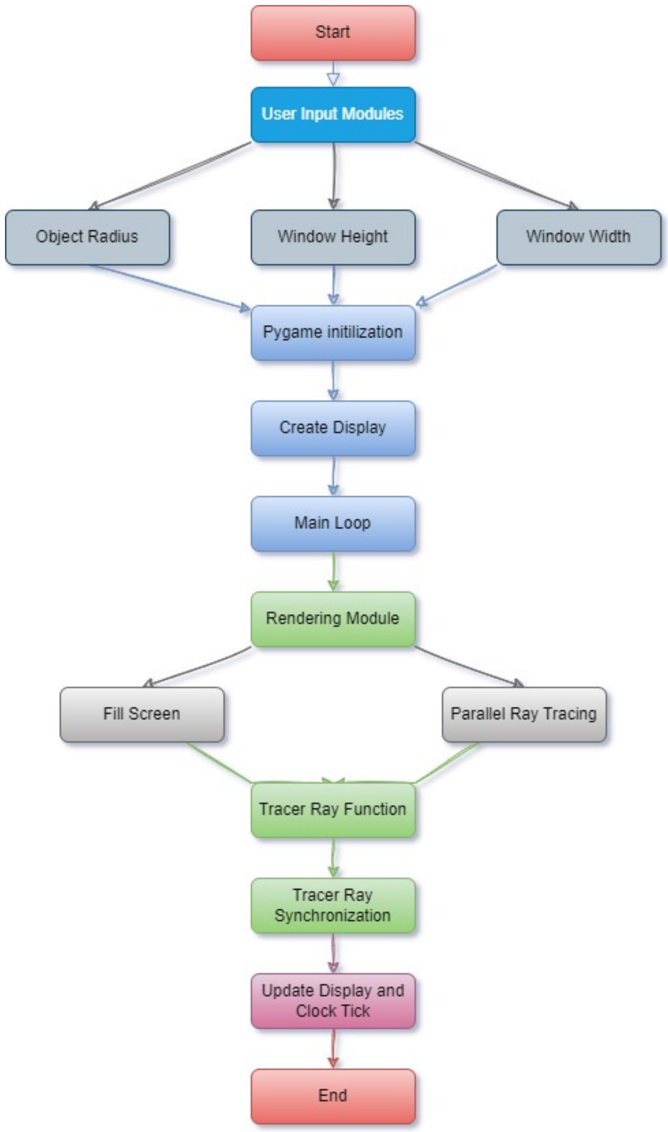
1) Initialization Phase: Lay the groundwork for the program by initializing essential libraries and defining fundamental constants.Modules: pygame-init and constants ensure a stable and standardized starting point for subsequent phases.
2) Object Representation: Then we define the properties of the central object within the scene, namely its position and radius.The module object-properties encapsulates the essential attributes of the object for later use in the ray tracing and rendering processes.
3) Ray Tracing: Implementing the core ray tracing algorithm responsible for simulating the interaction of light rays with the objects in the scene.We have the module ray-tracing focuses on the detailed computation of ray-object interactions, utilizing the object properties and screen segments for parallel processing.
4) Parallelization: Addressing computational bottlenecks by introducing parallel processing, distributing the workload among multiple threads. The module parallelization manages the division of the screen into segments, creating a thread pool to enable concurrent execution of the ray tracing algorithm.
5) Rendering: Orchestrating the execution of parallelized ray tracing to create a visual representaion of the scene on the screen.Also filling the screen with a dynamic display, leveraging the parallelized ray tracing process for optimal efficiency.
6) Performance Measurement: This phase has a goal to evaluate the efficiency gains achieved through parallelization by measuring the script's execution time.

This structured approach to development ensures a systematic and comprehensive exploration of each phase, fostering a clear understanding of the solution architecture and facilitating a successful implementation. In conclusion, the solution architecture guides the project from its initial phases through ongoing development, testing, and potential dissemination. By addressing each aspect systematically, the architecture ensures a thorough and well-documented development process, setting the stage for the parallelized ray tracing program's success and future evolution.

1) Start: At the beginning of the script execution, the interpreter starts interpreting the Python code.
2) User Input Module: The program foregoes interactive user input and directly sets specific values for the object radius and window size within the code. These predetermined values are crucial for optimizing the performance of the ray-tracing simulation. By setting these parameters programmatically, we can fine-tune the visual characteristics of the rendered scene and tailor the simulation to specific requirements without incurring the runtime overhead associated with interactive input. This approach is particularly useful when aiming for consistent and optimized performance in scenarios where dynamic user input is not a priority.
3) PyGame Initialization: Pygame is initialized to set up the underlying framework for graphics and game-related functionalities. This involves preparing Pygame modules to handle display windows, user input, and other gaming-related features.
4) Create Display: Using the Pygame library, a display window is created based on the user's specified win-

dow size. The window dimensions are determined by the user-input values for window width and window height. Additionally, the window title is set to "Simple Ray Tracing" to identify the application.

5) Main Loop: The program enters the main game loop, a fundamental structure in game development. This loop continually executes, processing user input and updating the display to create an interactive experience.
6) Renderer Module: The render-scene function from the renderer module is called. This function is responsible for initiating the rendering process, setting the stage for creating the visual elements of the scene. The rendering module contains the core logic for creating the visual scene. It involves multiple steps, including filling the screen with a background color and initiating parallel ray tracing.
7) Trace Ray Function: Within the renderer module, the trace-ray function is called for each pixel in the specified screen segment. This function calculates the distance of each pixel from the object center and determines if it falls within the object's radius. If so, the pixel color is set to black, contributing to the rendering of the object.
8) Thread Synchronization: To ensure a synchronized and coherent rendering process, the main thread waits for all spawned threads to complete their tasks. This synchronization prevents race conditions and ensures that the entire screen is processed before moving to the next step.
9) Update Display and Clock Tick: Once the ray tracing and rendering are complete, the display is updated to reflect the newly rendered scene. This step involves flipping the display to showcase the results of the rendering process.To control the frame rate and maintain a consistent animation speed, the program utilizes Pygame's clock mechanism. The clock ticks regulate the timing of each iteration of the main loop, ensuring a smooth and visually appealing experience.The program returns to the main loop after completing the rendering and display update. This loop continuation allows for continuous monitoring of user input and ongoing updates to the displayed scene.
10) End: Program execution concludes.
The main game loop eventually exits when the user closes the window or performs some other termination action. Pygame resources are cleaned up, and the program gracefully terminates. This step marks the conclusion of the program's execution. The main loop continuously checks for user input and updates the display until the user decides to exit. Upon termination, any allocated resources are released, and the program exits. This is the final phase of the program's lifecycle.



IV. RESULTS

In this section, we present the initial results of our study. The results are depicted in tables that describe our performance improvement achieved through the implementation of multithreading in our Ray Tracing Application

V. RESULTS

Ray tracing, a rendering technique in computer graphics, simulates the interaction of light with objects to create realistic images. As computational demands rise, optimizing ray tracing performance becomes crucial. This analysis compares the execution times of a non-parallel ray tracing application with its parallelized version. By evaluating the efficiency gains of parallelization, we aim to provide insights into the potential benefits for graphics-intensive applications.

The ray tracing application is designed to render a scene using two different approaches: a non-parallel version and a parallel version utilizing multi-threading. The primary goal is to analyze the performance difference between these two implementations.

- Non-Parallel (Sequential) Version:
  In the non-parallel version, the ray tracing algorithm is implemented with a sequential approach, where each pixel in the scene is evaluated one after the other in a nested loop. The application measures the time it takes to complete the rendering process in this non-parallel mode.

Height	Width	Radius	Time
800	600	20	0,0612
800	600	90	0,0605
1440	900	20	0,0700
1440	900	90	0,0620
3840	2160	20	0,110
3840	2160	90	0,104

TABLE I
RESULTS FROM THE NON-PARALLEL VERSION

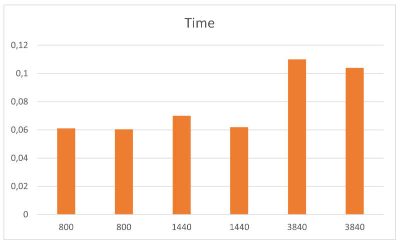


Fig. 1. Non Parallel

- Parallel Version:
  The parallel version of the application introduces multi-threading to enhance performance. The screen is divided into segments, and each segment is processed concurrently by a separate thread. The number of threads is set to four in the provided code (num-threads = 4), allowing for parallel processing of distinct portions of the scene. The application measures the time taken to render the scene in this parallelized mode.

Height	Width	Radius	Time
800	600	20	0,0550
800	600	90	0,0535
1440	900	20	0,0615
1440	900	90	0,0611
3840	2160	20	0,0982
3840	2160	90	0,0883

TABLE II
RESULTS FROM THE PARALLEL VERSION

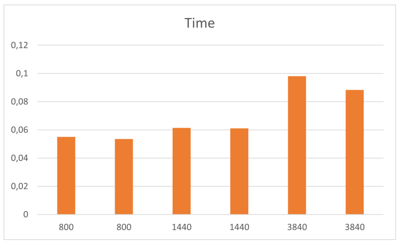


Fig. 2. Parallel

The execution times of the non-parallel and parallel versions are compared to assess the impact of parallelization on the overall rendering speed. The efficiency gain achieved through multi-threading is quantified by comparing the time taken for the two implementations. The experimental results demonstrate that the parallelized version of the ray tracing application surpasses its sequential counterpart, showcasing enhanced rendering speed. Notably, an increase in the window size correlates with extended processing times, highlighting the impact of canvas dimensions on computational workload. Additionally, it is observed that larger object radii contribute to faster processing times, albeit with a marginal difference, suggesting a subtle influence of object size on overall rendering efficiency.

1) Conclusion: The culmination of the ray tracing experiments provides valuable insights into the performance dynamics of the implemented application. The parallelized version, leveraging multi-threading for concurrent processing, emerges as the more efficient rendering solution, showcasing its potential to handle computational workloads more swiftly.

The observed correlation between window size and processing time indicates the sensitivity of the application's performance to canvas dimensions. As the window size increases, the computational demands escalate, emphasizing the need for careful consideration of resolution when aiming for optimal rendering speed.

An intriguing finding surfaces in the relationship between object radius and processing time. Despite a generally small difference, larger object radius contribute to faster processing times. This nuanced effect suggests that the size of objects within the scene can impact the rendering efficiency, albeit modestly.

REFERENCES

[1] Whitted, T. (1980). An Improved Illumination Model for Shaded Display. Communications of the ACM, 23(6), 343–349.
[2] Kajiya, J. T. (1986). The Rendering Equation. ACM SIGGRAPH Computer Graphics, 20(4), 143–150.
[3] Jensen, H. W. (1996). Global Illumination using Photon Maps. Rendering Techniques '96, 21–30.
[4] Veach, E., Guibas, L. J. (1997). Metropolis Light Transport. Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, 65–76.
[5] Wald, I., Mark, W. R., et al. (2001). Real-Time Ray Tracing for Interactive Global Illumination. ACM SIGGRAPH 2001 Proceedings, 434–446./
[6] NVIDIA Corporation. (2010). NVIDIA OptiX Ray Tracing Engine. Retrieved from https://developer.nvidia.com/optix
[7] Intel Corporation. (2014). Embree: A High Performance Ray Tracing Kernel. Retrieved from https://www.embree.org/