A Distributed Path Tracer

Yaojun Huang
hyjun@kaist.ac.kr
KAIST

Abstract

This project introduces a distributed path tracer designed for efficient and scalable graphics rendering. The system employs a client-server architecture where the client computes direct lighting locally while distributing the computationally intensive indirect lighting tasks across multiple servers. Key features include automatic server detection, dynamic load balancing, user input synchronization, and fault-tolerant task redistribution. The system is built on ZeroMQ to ensure robust and asynchronous communication, enabling seamless integration of results from multiple servers into the final rendered image.

Experiments demonstrate the system's ability to efficiently schedule tasks, maintain interactivity, and adapt to server failures while achieving enhanced rendering quality with additional server resources. Even under limited or unstable network conditions, the system ensures basic functionality by supporting fallback to local rendering.

This project showcases the feasibility of distributed rendering for graphics applications, highlighting its potential to improve rendering performance and scalability in real-world scenarios.

Keywords

Distributed System, Computer Graphics, Path Tracing, ZeroMQ

1 Introduction

With advancements in communication technology, online and cloud gaming have become increasingly popular, allowing players to enjoy games anywhere and at any time. This growing market has prompted numerous companies to launch their own cloud gaming services.

In traditional cloud gaming architectures, a game instance operates entirely in the cloud. A thin client sends the user's input to a cloud gaming server, which handles the game logic computations and full-frame rendering. The final rendered images are then streamed back to the thin client for display. This model enables gaming on low-performance devices, is easy to deploy, and ensures compatibility with legacy applications. However, it also introduces high latency and renders the service completely inaccessible without a network connection.

As hardware capabilities continue to improve, thin devices are becoming more powerful and capable of handling tasks beyond simple image display. Traditional cloud gaming servers, often located in remote data centers, face high latency and increased operational costs. To address these challenges, a hybrid solution for scalable and cloud-native gaming has been proposed. In this approach, basic and latency-sensitive computations are performed on local devices, while cloud computing resources are utilized for more demanding tasks. This enables limited access to the application even without a network connection and enhances rendering quality when a connection is available.

In global illumination rendering, the rendering equation can be divided into two parts: emission and direct illumination, which are view-dependent and computationally inexpensive, and indirect illumination, which is view-independent, computationally intensive, mostly smooth, and can tolerate some latency without significant visual impact. These two components can be computed separately and integrated during the final rendering process.

Traditional gaming architectures compute all rendering locally, limiting scalability. CloudLight [3] and Distributed Hybrid Rendering (DHR) [6] address this by offloading computationally intensive tasks, such as indirect illumination, to the cloud. Kahawai[4] employs a unique approach where the server computes both low-quality and high-quality renderings, streaming only the delta between the two to the client. The client combines this delta with locally rendered low-quality frames to produce high-quality visuals, enabling limited offline functionality. However, these models depend heavily on a single server, and performance can degrade significantly if the server experiences overload or disconnection.

In contrast, parallel rendering [1] techniques, widely used in offline media production such as film, leverage rendering farms to generate highly realistic images with substantial computational budgets. While powerful, these methods are unsuitable for real-time applications due to their focus on high-fidelity production rather than interactive performance. Combining the strengths of hybrid and parallel rendering methods offers a promising pathway for scalable and robust distributed rendering systems, especially for interactive applications like augmented reality (AR) and virtual reality (VR).

This term project aims to develop a distributed path tracer that computes direct illumination locally on the user's device while distributing the indirect illumination computation across other devices. The goal is to evaluate the feasibility and scalability of hybrid rendering in gaming applications.

2 Background

2.1 The Rendering Equation and Path Tracing

The Rendering Equation [5], introduced by James Kajiya at SIGGRAPH 1986, is a fundamental framework for modeling light transport in a scene. It determines the color and brightness of visible surfaces using a mathematical model that calculates the color and brightness of each pixel by considering the light emitted and reflected in various directions, including the effects of shadows, reflections, and indirect illumination. The equation is expressed as follows:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n)\, d\omega_i$$

In this equation, $L_o(x, \omega_o)$ represents the radiance leaving a surface point x in the outgoing direction ω_o. The term $L_e(x, \omega_o)$ denotes the emitted radiance at point x in the direction ω_o. The integral accounts for all incoming directions ω_i over the hemisphere Ω above the surface. For each incoming direction, the radiance $L_i(x, \omega_i)$ contributes to the reflected light based on the surface's Bidirectional Reflectance Distribution Function (BRDF) $f_r(x, \omega_i, \omega_o)$, which characterizes how light is scattered at the surface. $(\omega_i \cdot n)$ is the dot product between the incoming light direction ω_i and the surface normal n and $d\omega_i$ represents the differential solid angle element for integration over all possible incoming directions over the hemisphere.

In general, the reflected light can be viewed as a combination of emissions from light sources, direct illumination on the surface, and the accumulation of indirect illumination.

The Rendering Equation serves as a general model for computing light in a virtual environment. One approach to solving this equation is Path Tracing, a Monte Carlo-based algorithm that approximates the solution by stochastically sampling light paths. The algorithm traces rays from the camera, allowing each ray to interact with surfaces according to their material properties. These interactions include reflection, refraction, scattering, and absorption. Rays are traced back to their origins, typically a light source, and the contributions from all sampled paths are combined to compute the final pixel color. To control computation intensity, Russian Roulette (RR) is used to terminate rays probabilistically.

Although Path Tracing produces highly realistic rendered images, it is computationally expensive. A large number of rays must be traced for each pixel to reduce noise and achieve accurate results, which is why it is often limited to offline rendering. However, advancements in hardware, such as the introduction of RT Cores for real-time ray tracing and Deep Learning Super Sampling (DLSS) for denoising and upscaling, have enabled some applications of Path Tracing in real-time rendering.

2.2 CloudLight

CloudLight [3] is a system proposed by NVIDIA that computes indirect lighting asynchronously on an abstracted, computational "cloud". The system is evaluated using three different indirect illumination strategies: path-traced irradiance maps, photon mapping, and cone-traced voxels. The overall architecture of CloudLight is depicted in Figure 1.

In the figure, the blue section on the left represents computations performed on the server, while the black section on the right represents computations performed locally on the client. The arrows indicate the bandwidth requirements between these components. As shown, the rendering pipeline is divided at the point with the lowest bandwidth demand.

For the voxel-based approach, the server employs one GPU (the global illumination GPU) to generate view-independent scene data and another smaller GPU (the final frame GPU) to generate view-dependent frames. The fully rendered frame is encoded on the server side and sent to the client for decoding and display.

In the irradiance map approach, the server gathers indirect lighting data, constructs the irradiance maps, and transmits them to the client. The client calculates direct lighting and performs a texture-space deferred shading pass using the irradiance map for indirect lighting.

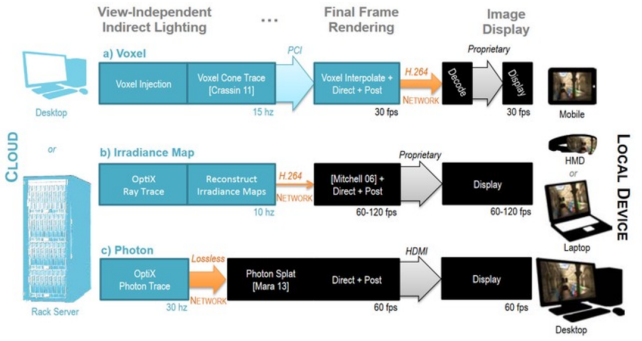

Figure 1: The CloudLight Pipelines of Three Indirect Illumination Strategies.

For photon mapping, the server traces photons using a ray tracer and transmits the photon data to the client. The client computes direct lighting locally and uses the transmitted photons to calculate indirect lighting.

Experiments with CloudLight were conducted on commercial hardware and software, demonstrating its robustness and scalability under realistic workloads. The results highlight its potential for enabling remote illumination in practical applications.

2.3 DHR: Distributed Hybrid Rendering

Distributed Hybrid Rendering (DHR) [6] is a rendering system that integrates local and cloud rendering to enhance the quality of graphics for the Metaverse. The local client generates frames with relatively low graphical settings using rasterization, while the server computes ray-traced shadows and streams the results back to the client to enhance visual fidelity.

Due to network latency, the visibility buffer received from the server may lag several frames behind the locally computed G-buffer. To address this issue, DHR employs motion vectors and G-buffers on the client side to predict the visibility buffer. This approach ensures both the correctness of the rendered frames and the responsiveness required for user interactions.

2.4 ZeroMQ

ZeroMQ [8] is an open-source, high-performance asynchronous messaging library designed for distributed and concurrent applications. Unlike traditional message-oriented middleware, ZeroMQ does not require a dedicated message broker while still offering robust message queuing capabilities. It supports various messaging patterns, including publish/subscribe, request/reply, and router/dealer, among others. Furthermore, ZeroMQ works seamlessly with different transport protocols, making it a flexible and scalable solution for messaging systems.

3 The Distributed Path Tracer

This section introduces the architecture of the Distributed Path Tracer, a distributed rendering system built with ZeroMQ. It leverages client-side ray tracing for direct lighting and server-side path tracing for indirect lighting, with the results seamlessly integrated on the client.

3.1 System Architecture

The system follows a classical Client-Server architecture, where both the client and server share a similar rendering algorithm and environment setup. The server executes the traditional path-tracing algorithm, whose pseudo code is illustrated in Figure 2.

```
shade(p, wo)
    # Contribution from the light source.
    L_dir = 0.0
    Uniformly sample the light at x' (pdf_light = 1 / A)
    Shoot a ray from p to x'
    If the ray is not blocked in the middle
        L_dir = L_i * f_r * cos θ * cos θ' / |x' - p|^2 / pdf_light

    # Contribution from other reflectors.
    L_indir = 0.0
    Test Russian Roulette with probability P_RR
    Uniformly sample the hemisphere toward wi (pdf_hemi = 1 / 2pi)
    Trace a ray r(p, wi)
    If ray r hit a non-emitting object at q
        L_indir = shade(q, -wi) * f_r * cos θ / pdf_hemi / P_RR

    Return L_dir + L_indir
```

Figure 2: The pseudo code of path tracing [7].

The glossy reflection model is implemented using the Frostbite 3 standard material BRDF, which includes diffuse and specular components [2]. Additionally, a joint bilateral filter is applied for final image de-noising. The key difference in computation between the client and the server is that the client only calculates the contribution from the light source, as indicated in the first code block of the pseudo code in Figure 2.

The system is designed with a single client, acting as the system initiator, and multiple servers to facilitate load balancing. An overview of the system architecture is shown in Figure 3.

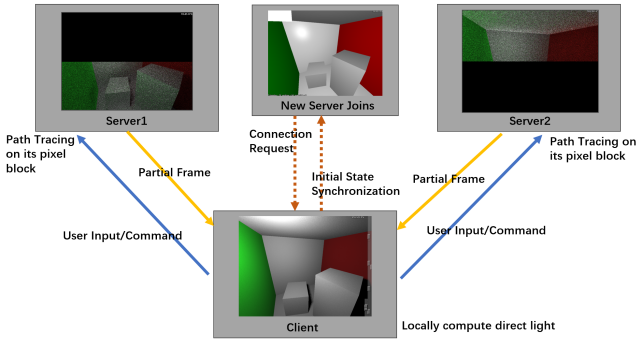

Figure 3: The System Architecture Overview.

The client computes the entire frame of direct lighting using ray-tracing techniques, while the servers handle path-tracing computations for their assigned pixel blocks. The servers transmit their partially rendered results to the client, which integrates the indirect lighting information with its locally computed direct lighting image to produce the final output.

The client broadcasts all user inputs—such as camera movements, field-of-view (FoV) changes, light placements, window size adjustments, tracing commands, and algorithm parameter updates—to all servers. When a new server connects, the client detects the event, initializes the new server with its current state, and schedules pixel block task allocation. Similarly, if a server disconnects due to hardware or network failures, the client detects the issue and redistributes the pixel block tasks accordingly.

This system supports fallback to purely local rendering, ensuring basic interaction and functionality even in unstable network environments.

3.2 Automatic New Server Detection and Initial State Synchronization

This feature leverages ZeroMQ's Router-Dealer communication pattern, a robust messaging model designed for building scalable, distributed systems. Router-Dealer supports asynchronous, non-blocking communication, making it ideal for scenarios where multiple clients or workers interact with multiple servers or request handlers in a load-balanced or message-routing environment.

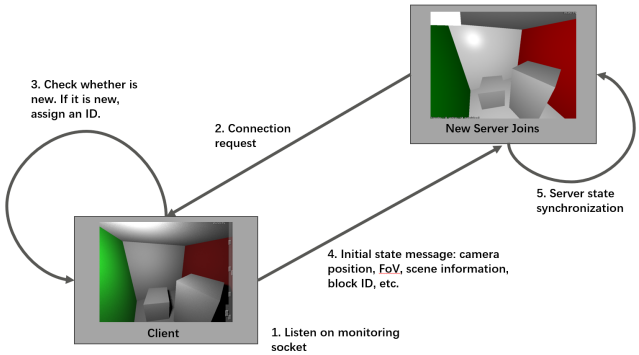

Figure 4: The Process of Server Detection and State Synchronization.

The process for server detection and initial state synchronization is illustrated in Figure 4. The Distributed Path Tracer client monitors a dedicated port for new server connections. When a server connects, it sends a "hello" message to the client and waits for a reply in a blocking manner. The client periodically checks the monitoring port for incoming messages. Upon receiving a message, the client verifies whether the sender's identity is new. If the server identity is unrecognized, it indicates a new server connection. The client records the server's identity in its active server array and assigns it an ID for subsequent pixel block scheduling.

Following this, the client sends its complete state to the newly connected server, including parameters such as camera position and orientation, field of view (FoV), light source positions, the current

scene configuration, kernel size of the joint bilateral filter, and samples-per-pixel settings. The server uses this data to synchronize its state, which includes loading the relevant scene and initializing path tracer parameters.

3.3 User Input Synchronization

This feature utilizes ZeroMQ's Pub-Sub pattern, a messaging model designed for efficient broadcast communication. The Pub-Sub pattern allows a Publisher (Pub) to send messages to multiple Subscribers (Sub), enabling one-to-many communication.

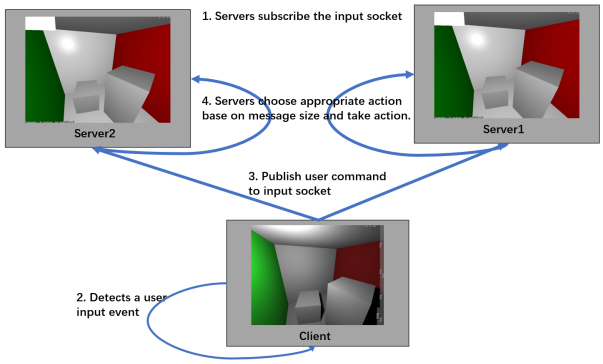

Figure 5: The Process of User Input Synchronization.

The process is depicted in Figure 5. The client initializes an input broadcast socket to which servers subscribe. Whenever the client detects a user input event (e.g., mouse movement, keyboard input, or window refresh), it packages the relevant parameters into a message and broadcasts it to the servers. Since the parameters for different event types vary, the size of the event message also varies. Servers handle these messages based on their size, which allows them to determine the appropriate action to take for each event type.

3.4 Server Failure Detection and Pixel Block Reallocation

Server failure detection and pixel block reallocation are also implemented using the Router-Dealer pattern on the monitoring port. The process is outlined in Figure 6. The client periodically sends heartbeat messages to all servers listed in its active server array, with a six-second interval between checks. Each server responds to the heartbeat message with a "hello" reply. The client tracks responses, recording the IDs of servers that reply.

At the next heartbeat cycle, the client checks whether all servers have responded. If one or more servers fail to reply, they are marked as unresponsive, and the client initiates task reallocation. The client assigns new IDs to the responsive servers and sends the updated IDs and the current count of active servers to all functioning servers. Upon receiving this update, each server adjusts its parameters accordingly, and new pixel blocks are assigned.

3.5 Final Frame Integration

Final frame integration is also handled using the Router-Dealer pattern, but through the frame socket. The process is detailed in

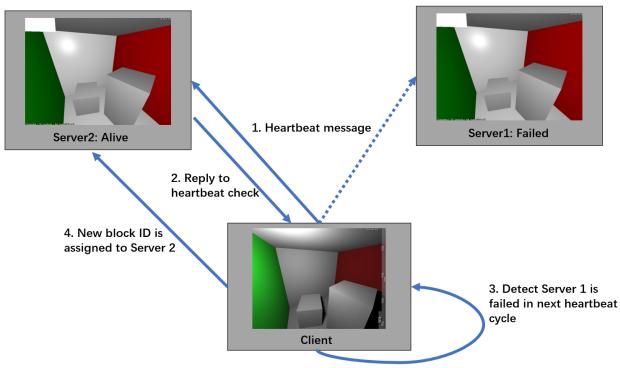

Figure 6: The Process of Failure Detection and Task Reallocation.

Figure 7. When the client receives the "start tracing" command, it broadcasts all relevant path tracing parameters through the input broadcast socket and begins computing direct lighting locally. Once the local computation is complete, the client starts monitoring the frame socket.

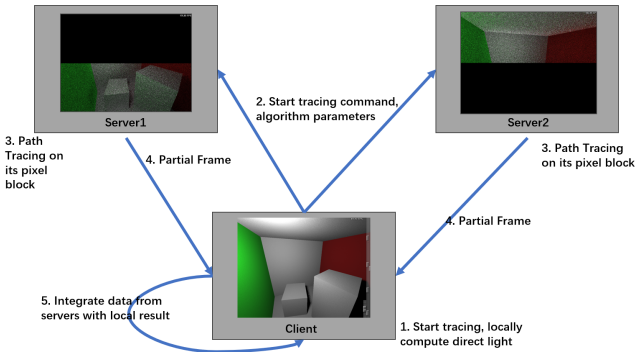

Figure 7: The Process of Final Frame Integration.

Upon receiving the command, servers commence path tracing computations for their assigned pixel blocks. After completing indirect lighting computations, servers optionally apply denoising techniques (depending on the parameter settings) and send the processed partial frames to the client via the frame socket. These messages contain the server's identity, allowing the client to associate the received data with the corresponding pixel block. The client then blends the indirect lighting results from the servers with its locally computed direct lighting results.

When all partial results are received and integrated, the client completes the image blending process, resulting in the final fully rendered path-traced image.

4 Implementation and Experiment

This section details the experimental setup and discusses the results, demonstrating the feasibility of distributed path tracing.

4.1 Implementation and Environment Setup

The path tracer is implemented on the CPU, utilizing OpenMP (OMP) for multi-threaded computation to accelerate rendering. Both the client and servers operate on the same machine, which is equipped with an AMD Ryzen 9 7945HX processor and 32GB of RAM. Communication between the client and servers is established through the TCP protocol.

Port 5555 is designated for synchronizing user input, while port 5556 is responsible for detecting server connections or disconnections and transmitting the initial state. Frame data is transmitted via port 5557.

4.2 Experiment Result

The reference image is shown in Figure 8. It was rendered using a single-machine path tracer with 256 samples per pixel to achieve a noise-free image. This process is highly computation-intensive.

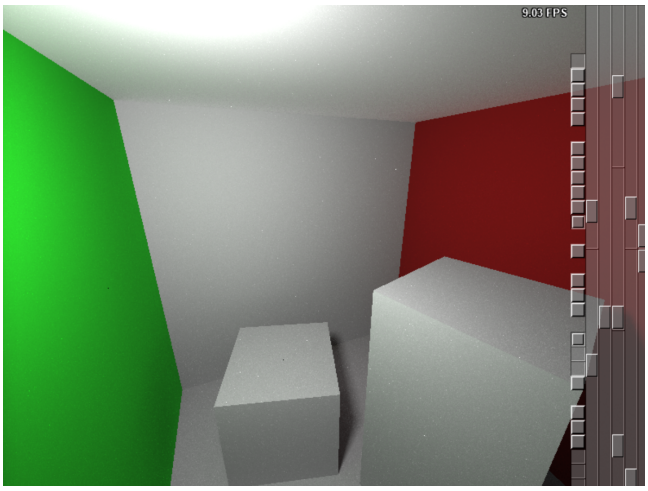

Figure 8: The Reference Image.

Figure 9 shows the result produced by the distributed path tracer client without any server connection. This image includes only the direct lighting contribution.

When a server is connected, the client and server execute their computations independently. Figure 10 illustrates the intermediate rendering process of the client and server working together.

After the server completes its rendering tasks, the client's rendered image is enhanced with indirect illumination results from the server, as shown in Figure 11.

With more servers connected, the client is capable of broadcasting user input commands to all servers in a synchronized manner. This synchronization is demonstrated in Figure 12.

Once the client completes its local computation of direct lighting, it waits for the servers to send their results. This intermediate state is shown in Figure 13.

When the servers complete their computations, the final image is integrated by the client. Figure 14 illustrates the result, where a joint bilateral filter applied on the server side's indirect illumination results effectively reduces image noise with low sample per pixel.

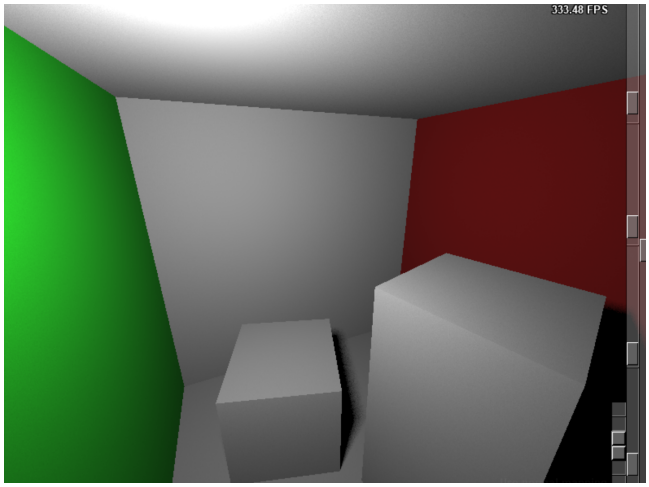

Figure 9: The Distributed Path Tracer Client's Direct Lighting Result.

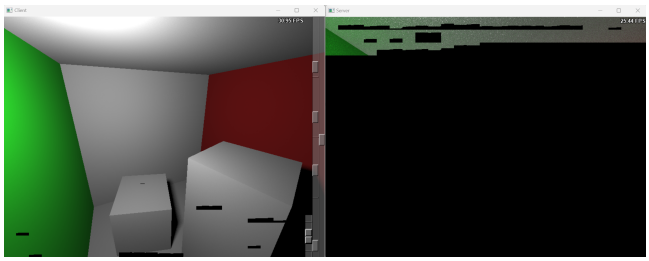

Figure 10: The Intermediate Rendering Process of the Client and Server.

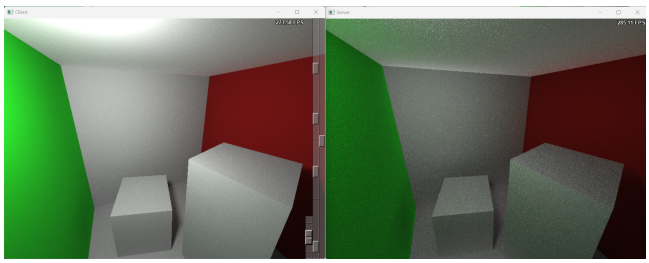

Figure 11: The Rendering Result of the Client and One Server.

If a server becomes disconnected, the client detects the failure and redistributes the task to the remaining servers. This rescheduling process is depicted in Figure 15.

This task distribution strategy, which involves dividing an image into blocks, is applicable to various scene workloads. Figures 16, 17, and 18 illustrate the rendering results for different scenes using varying numbers of servers.

Figure 16 presents the rendering result of the Sponza scene with one server. Figure 17 shows the rendering result of the Crytek-Sponza scene with two servers, while Figure 18 depicts the rendering result of the Conference scene with three servers.

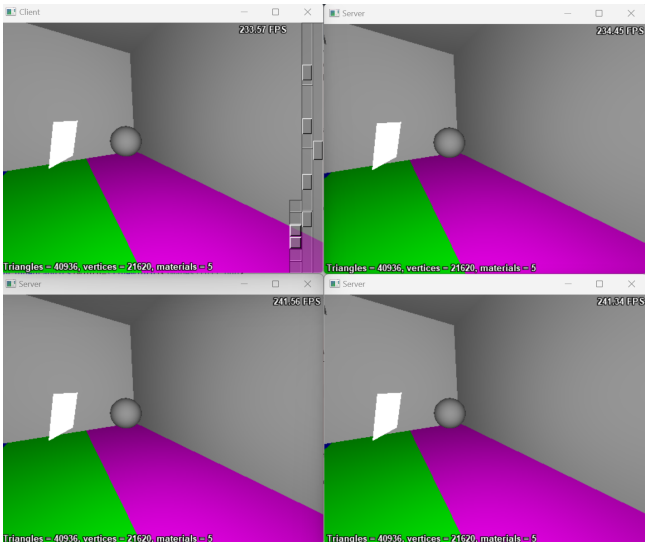

Figure 12: The Synchronized Control between Client and Three Servers.

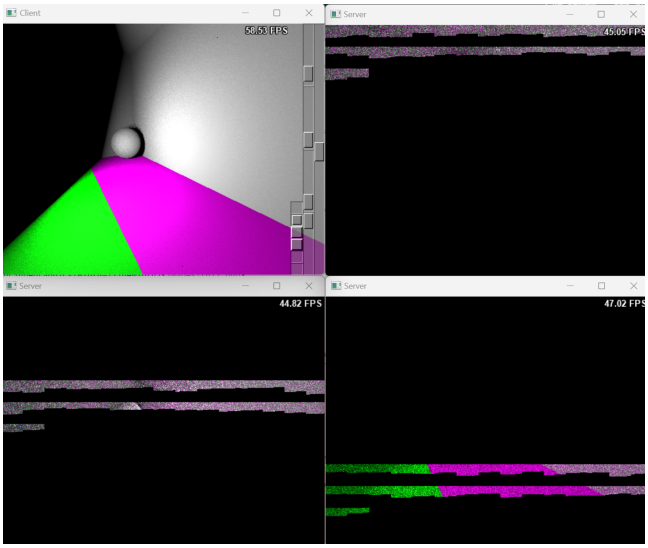

Figure 13: The Client Waits for the Results from Three Servers.

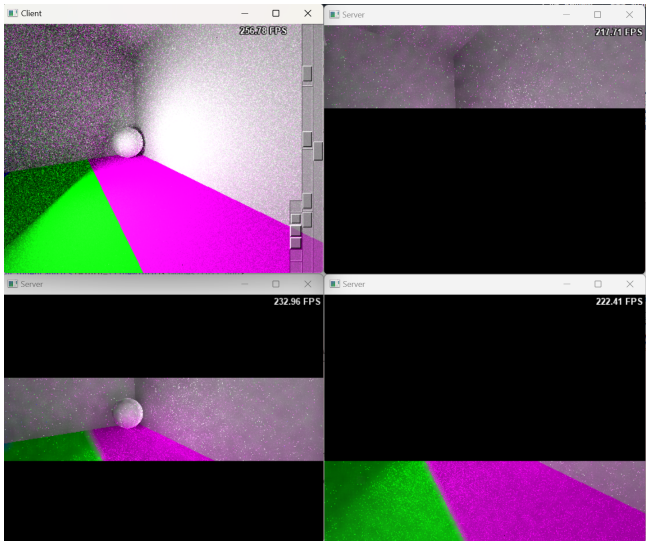

Figure 14: The Client Integrates the Results from Three Servers.

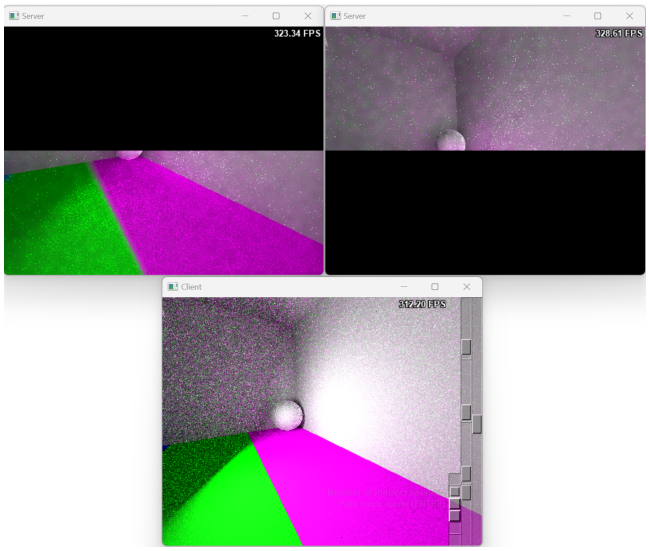

Figure 15: The Client Detects One Server Failure and Redistribute the Task.

5 Discussion

The current implementation of the distributed path tracer was designed to operate on Windows and was tested on a single machine due to device limitations. This setup inherently constrained the number of rays available for rendering computation. Nevertheless, the experiment successfully validated the feasibility and scalability of the distributed path tracer.

The client system demonstrated resilience by leveraging local computational capabilities, and maintaining interactivity even without server connections or network connectivity. This capability underscores the importance of fallback mechanisms in ensuring a seamless user experience, particularly in interactive media applications.

5.1 Scalability Analysis

As demonstrated in Figures 8 and 9, incorporating indirect illumination significantly enhances image realism but incurs substantial computational costs. Notably, later bounces contribute marginally to the final image's appearance, suggesting that combining direct illumination with a limited degree of global illumination suffices for most practical applications.

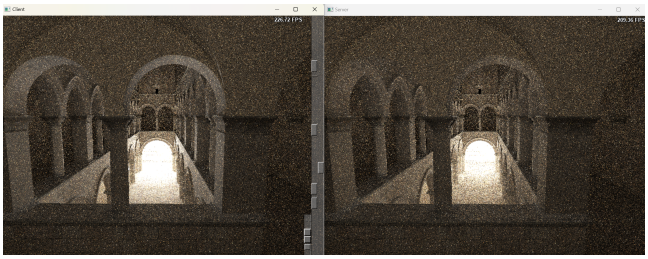

Figure 16: The Rendering Result of Scene Sponza with One Server.

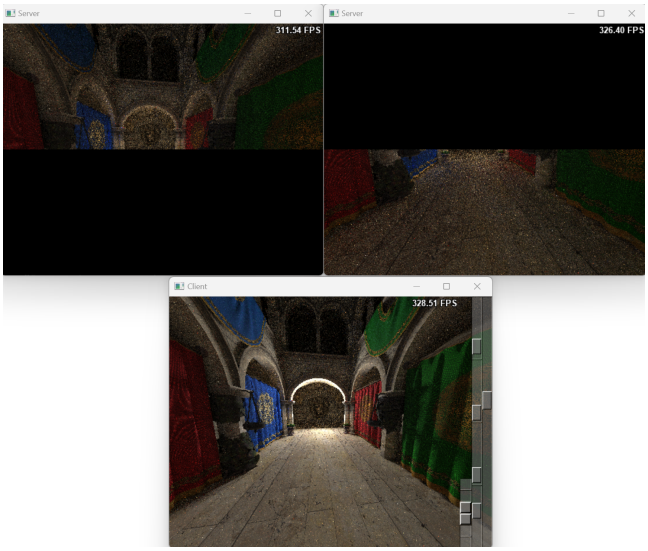

Figure 17: The Rendering Result of Scene Crytek-Sponza with Two Server.

In purely cloud-based rendering systems, the computational burden shifts entirely to the cloud, achieving high-quality visuals at the cost of completely losing application access during network failures. This trade-off poses challenges for interactive media, where innovation depends on continuous responsiveness. Hybrid approaches that balance computation between local and cloud systems, coupled with robust fallback mechanisms, represent a promising area for future research.

From a computational perspective, the time complexity of the path-tracing algorithm can be expressed as follows:

- For a scene with T triangles and a ray with an average depth of D bounces, the time complexity per ray is $O(D \cdot T)$.
- Utilizing an acceleration structure such as a Bounding Volume Hierarchy (BVH) reduces this to $O(D \cdot \log T)$.
- For an image with resolution $W \cdot H$, the total time complexity for one ray per pixel is $O(W \cdot H \cdot D \cdot \log T)$.

Given a machine with computational capacity C, the maximum samples per pixel (SPP) is determined by:

$$SPP = \frac{C}{O(W \cdot H \cdot D \cdot \log T)}$$

Figure 18: The Rendering Result of Scene Conference with Three Server.

On the client side, where $D = 1$ (direct illumination only), the computational complexity is significantly lower compared to the server, which must process global illumination with D bounces. As a result, the server's computational workload is D times greater than that of the client. When additional servers are introduced, the rendering workload is distributed across N servers, with each server processing a proportional fraction of the image. Consequently, the samples per pixel (SPP) for each server can be expressed as:

$$SPP = \frac{C}{O(W \cdot H \cdot D \cdot \log T)} \cdot N$$

The workload distribution strategy allows servers to allocate more resources to improving rendering quality by increasing the sampling rate. Alternatively, if the SPP is fixed, by allocating $N = D$ servers, the computation required for each server is effectively reduced to $\frac{1}{N}$ of the original workload.

Alternatively, if the SPP is fixed, allocating $N = SPP \cdot D$ servers effectively reduces the computation per server to $\frac{1}{N}$ of the original workload. This adjustment aligns each server's computational time with that of the client processing direct illumination, assuming the client and server have similar computational capabilities. As a result, the servers can complete rendering almost simultaneously with the client, enabling immediate result previews. In practice, servers are typically more powerful than clients, reducing the required number of servers. This system improves the overall fidelity of the image and ensuring scalability as more servers joins.

5.2 Future Improvement

The current implementation is limited to offline rendering scenarios due to the computational complexity of the path tracing algorithm. For real-time rendering, it requires significant improvements. These include the integration of sophisticated global illumination algorithms and advanced computer graphics techniques, such as up-sampling and frame generation, to attain a minimum frame

rate of 30 FPS. Transitioning to GPU-based computation and separating computation from display logic are also essential for more efficient task offloading. Furthermore, predictive techniques will be necessary to reduce the impact of network latency and enhance system responsiveness. Accurate device capability estimation is essential for balanced task distribution and consistent frame rendering. These improvements would facilitate the transition from offline to real-time rendering, expanding the system's applicability in interactive and dynamic environments.

References

[1] A. Chalmers, E. Reinhard, and T. Davis. 2002. *Practical Parallel Rendering*. CRC Press. https://books.google.co.kr/books?id=loxhtzzG1FYC

[2] Sébastian Lagarde Charles de Rousiers. 2014. Moving Frostbite to Physically Based Rendering 2.0. https://www.ea.com/frostbite/news/moving-frostbite-to-pb

[3] Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter-Pike Sloan, and Chris Wyman. 2015. CloudLight: A System for Amortizing Indirect Lighting in Real-Time Rendering. 4, 4 (October 2015), 1–27. https://jcgt.org/published/0004/04/01/

[4] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. , 15 pages. https://doi.org/10.1145/2742647.2742657

[5] James T. Kajiya. 1986. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. Association for Computing Machinery, New York, NY, USA, 143–150. https://doi.org/10.1145/15922.15902

[6] Yu Wei Tan, Alden Tan, Nicholas Nge, and Anand Bhojan. 2022. DHR: Distributed Hybrid Rendering for Metaverse Experiences. In *Proceedings of the 1st Workshop on Interactive eXtended Reality (MM '22)*. ACM, 51–59. https://doi.org/10.1145/3552483.3556455

[7] Lingqi Yan. 2020. GAMES101. https://sites.cs.ucsb.edu/~lingqi/teaching/games101.html

[8] ZeroMQ. 2024. ZeroMQ: An open-source universal messaging library. https://zeromq.org/