Foveated Ray Tracing

A cloud-based parallel ray-tracing system

FRANK QIN and STUDENT TWO, University of Washington

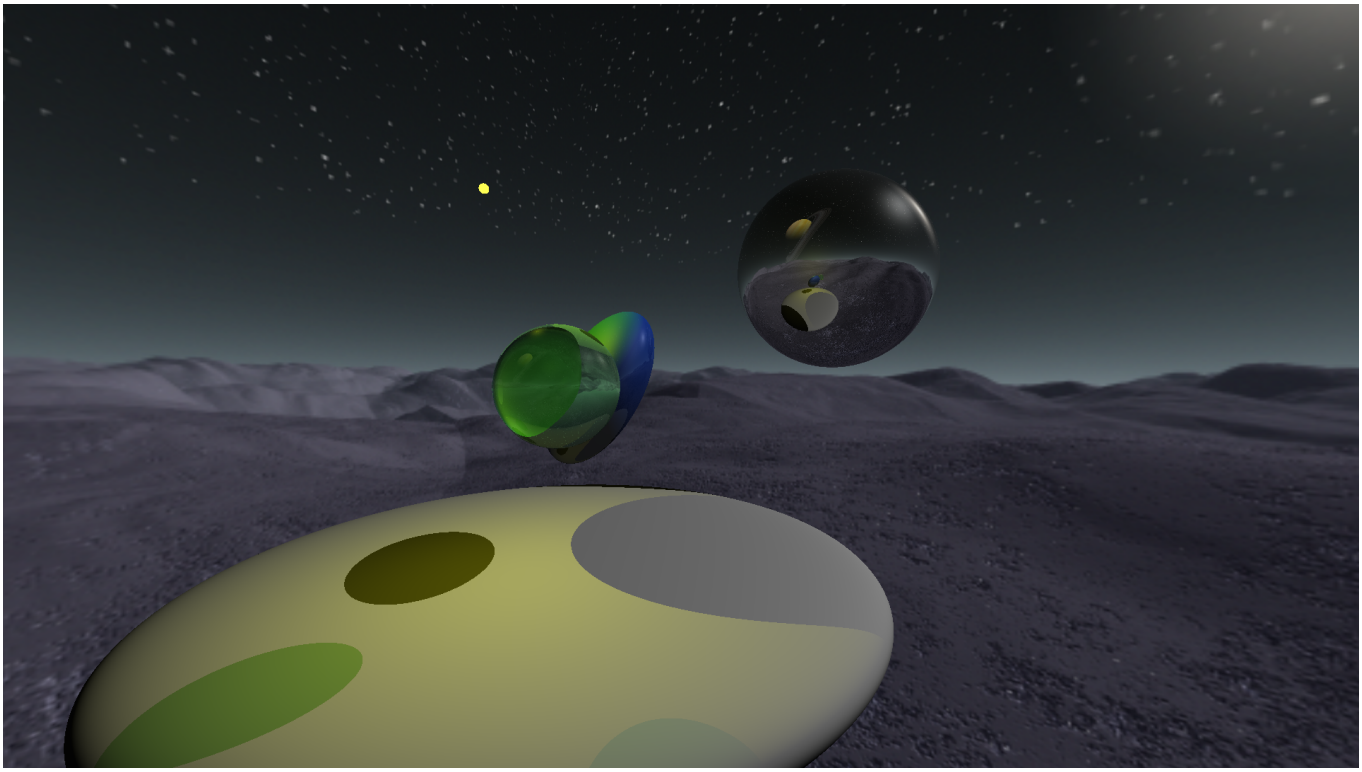


Fig. 1. Our ray tracing based rendering approach produces higher quality frames than traditional approach used by CSE 490V homework projects. The scene includes a yellow point light source and a white directional light source from the upper-right corner. Our approach can render features that are hard to render with the traditional approach, such as reflective surfaces (the upper-right sphere), translucent objects (the middle green sphere), opaque shadows (two farther shadows on the bottom ellipsoid, one from the directional light and one from the point light), and translucent shadows (two nearer shadows on the bottom ellipsoid, one from the directional light and one from the point light).

Ray tracing workload is highly parallelizable. Since current ray tracing implementations are CPU-based, we made the hypothesis that ray tracing performance can greatly benefit from having access to multiple CPU cores. The goal of this project is to implement a high performance ray tracer for real-time VR rendering by utilizing all cores in a system and separating the ray tracer from the end user, so that the ray tracer can be put on a dedicated server with high computational power. After testing, our fully parallel ray tracer performance is significantly better than our single-threaded version, but worsened when we put it on a server with larger number of cores. We have a few hypothesis about why the performance is lower on the server, and future work is needed to verify and resolve these issues.

1 INTRODUCTION

Ray tracing, an image synthesis technique, is a way of simulating realistic light interacting with objects in a scene. In general, ray tracing produces much higher quality rendering than traditional graphics pipeline used in CSE 490V homework projects. However, ray tracing is rarely used as a real-time rendering method because of its high computational workload per frame. The goal of this project is to implement a simple but high performance ray tracer to be acceptable in a real-time VR system.

In ray tracing, the color of each pixel is sampled by casting a rays from the viewer to intersect objects in the scene, and potentially casting rays from intersected objects to reflective/refractive directions recursively. Until recently with the new Nvidia RTX GPUs, these calculations are typically done by CPUs. Since the color of each ray is calculated only based on the scene and independent from other rays, this process is highly parallelizable. From past experience with the CSE 457 project ray tracer, we know that ray tracing process can be significantly speed-ed up by multi-threading. Since more cores yielded better performance, we made the hypothesis that

Authors' address: Frank Qin, qzh@cs.washington.edu; Student Two, student2@cs.washington.edu, University of Washington.

we can achieve even better performance by running the ray tracer on a server with larger number of cores. To connect the ray tracer on the dedicated server and the end user VR headset, we created a simple protocol based on WebSocket to transfer pose data from the headset to the ray tracer, and rendered data from the ray tracer to the headset.

We tested the performance of our ray tracer on my personal computer and the UW CSE attu server. We chose the attu server because it has a lot of cores, is available to all CSE students, and has relatively low ping to us. Our test results agreed with our prior observation of CSE 457 ray tracers that the performance greatly benefited from multi-threading. However, our test results showed the opposite of our hypothesis that performance should be higher on a server. We made some hypothesis about why the performance is lower on attu, and future work is needed to find the exact cause of slow-down and resolve the issue.

1.1 Contributions

Our primary contributions are:

- We implemented a simple high performance ray tracer. The ray tracing algorithm comes from our CSE 457 ray tracing project, but the ray tracer in this project is written from scratch to optimize for performance and simplicity.
- We created a VR headset to ray tracing server protocol based on WebSocket. This protocol allowed us to run the ray tracer on a server and still produce results on the end user's headset.
- We found that simply running a ray tracer on a server with large number of cores may produce lower performance than running it on a laptop. Future work must be done before fully utilizing the computational power of a server.

2 RELATED WORK

Ray tracing can be used both in static images or real-time scenes. As demonstrated by a video game developed by the ray-tracing team at Intel - Enemy Territory: Quake Wars [4], real-time ray tracing can offer amazing graphical renderings. For example, to display shadow effects, as rasterization, which is used in a lot of games, requires complicated calculations and approximations, ray tracing solves the problem with just checking if the path from the light to the surface is blocked or not (checking a shadow ray). On the other hand, performance was discovered as the main reason why this technique is not yet used in mainstream games, especially when we have ten or more surfaces in a row, such as a tree, rendering costs increased drastically.

As a possible solution for ray tracing performance issue, a GPU-driven technique using foveated rendering coupled with eye-tracking is discussed in paper [1] which can accelerate interactive 3D graphics with minimal loss of perceptual details. Based on the experiments mentioned in the paper, a 2.8x to 3.2x speedup is possible. As a possible future goal, we may consider apply this technique to optimize the performance on VR headset with eye-tracking systems.

3 METHOD

We decided to separate the VR system into a front-end on the user's display and a ray tracer back-end running on a server. The front-end and the back-end will communicate through a custom protocol designed for this project. The back-end should hold all data necessary to complete the rendering, such as the scene and textures. The front-end should display rendered image to the user headset, and transmit user options such as rendering quality, and real-time data such as the viewer pose.

The protocol is based on WebSocket. The protocol has three types of WebSocket packets: trace option update, pose update, and rendered data.

- A trace option update packet is used for the front-end to transmit user settings, such as the resolution, to the ray tracer. It should be sent by the front-end when it first connects to the back-end, and can be used to update ray tracing parameters whenever the user changes the settings. A trace option update packet contains the trace depth and a number of rays. The front-end is responsible for creating these rays according to user options.
- A pose update packet is used for the front-end to notify the most recent pose of the headset. The pose update will contain a timestamp of when the pose is generated. Ideally for every pose update, the back-end will perform ray tracing on the new viewer pose. If the back-end cannot keep up with the rate of the pose update, which means the previous rendering has been not finished, the back-end will ignore the pose update.
- A rendered data packet is used for the back-end to transmit ray tracing result back to the front-end. The packet will be sent by the back-end whenever a ray tracing has finished. The packet will contain the same timestamp data from the pose update packet. The front-end will know which pose this rendering is generated from the timestamp data, and can possibly do time wrapping with the latest pose.

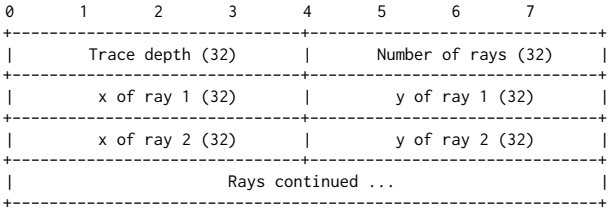
The back-end will organize the scene in a hierarchical way similar to CSE 457 projects rather than as a list of objects similar to CSE 490V homework. This allows the builder of the scene to group objects and transform them together. This also allows the ray tracer to check for intersections more efficiently by using a Binary Space Partitioning tree. For each ray, the ray tracer walks through the hierarchical tree of objects and first check whether the ray intersects a node, and then recursively check whether the ray intersects its children. When an intersection is passed back up to the parent, the intersection position will be transformed according to the child transform matrix, and the intersection normal vector will be transformed using the normal transformation equation.

We did not implement the real front-end, but we implemented a test front-end to test our back-end.

4 IMPLEMENTATION DETAILS

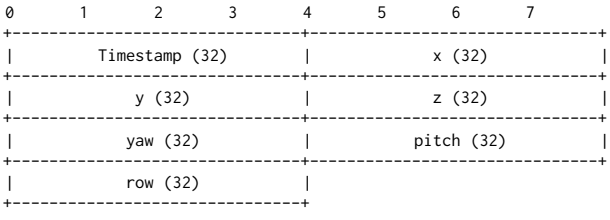
The protocol between the front-end and the back-end is defined as follows:

The trace option update packet contains the depth limit of recursive ray tracing and the rays the front-end needs to trace. The exact layout is described here:



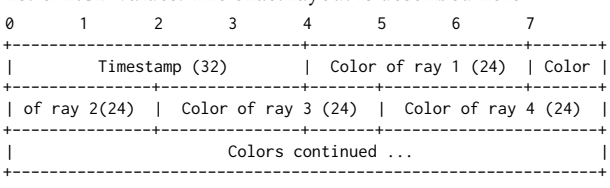
The depth is a 32-bit unsigned integer in little endian. The number of rays is a 32-bit unsigned integer in little endian, describing the number of rays after this field. Each ray is described by two 32-bit floating point in little endian, x and y. Each pair of x and y represents a ray from the origin of the view space, to the $[x, y, -1]$ direction in the view space. The direction will be normalized by the back-end.

The pose update packet contains a timestamp of this pose, the translation of the viewer from the origin, and the orientation of the viewer. The packet is fixed to be 28 bytes. The exact layout is described here:



The timestamp is a 32-bit value that the back-end will send back to the front-end as is. In our test front-end, we use the Unix time as an unsigned 32-bit integer as the timestamp. The x, y, and z are 32-bit floating point values in little endian. They describes the position of the viewer in world space. The *yaw*, *pitch*, and *row* are 32-bit floating point values in little endian, They describes the orientation of the viewer in world space, encoded as Euler's angles. We chose to use Euler's angle because it is the smallest representation. The back-end will convert them into the transformation matrix as soon as it receives the pose.

The rendered data packet contains the echoed timestamp and a list of RGB values. The exact layout is described here:



The timestamp is copied from the pose update packet as is. In the list of colors, each color is described by a 24-bit RGB value. The order of the rays will be exactly the same as the order specified in the trace option update packet.

The goal of the back-end is to be a high performance ray tracer, so we decided to implement the back-end in C++. Even though the ray tracer uses generally the same algorithm as the CSE 457 ray tracer which is also in C++, it is built from scratch to avoid unnecessary functionalities and dependencies of the CSE 457 ray tracer and to optimize for performance. Specifically, the CSE 457 ray tracer is very closely coupled with a Qt graphics UI, and re-implementing the ray tracer from scratch is the only way to remove its Qt UI dependency.

It has the following external dependencies:

- Boost: asio and beast
- OpenGL Mathematics (GLM)
- Simple OpenGL Image Library (SOIL)

The Boost asio library provides a thread pool implementation for multi-threading, as well as network sockets. The Boost beast library provides a WebSocket server implementation. By using their asynchronous APIs, all network IO are non-blocking so all threads can be fully used by the ray tracing process.

Even though the back-end does not depends on any OpenGL rendering functionality, it uses the GLM library for its vector and matrix calculations. And the back-end also uses the SOIL library to load textures.

The color of each ray is modeled as three components: direct, reflection, and refraction.

- The direct component uses the phong lighting model. The ray tracer cast a ray from a point on the object to each light source. If the ray successfully reaches the light source, the ray tracer will calculate the light contribution from this light source using the phone lighting model. If the ray intersects an object first, the light contribution from this light source will be reduced according to the object's transmissivity.
- The reflection component is calculated by recursively casting a ray to the reflection direction. The reflection direction reflecting the viewer direction about the surface normal direction. After the light of the reflected ray is calculated, the light is reduced according to the object's surface reflectivity. The reduced light value is the reflection component.
- The refraction component is calculated by recursively casting a ray into the object along the refraction direction. The refracted angle is calculated by using Snell's law

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}.$$

  After the light of the refracted ray is calculated, the light is reduced according to the object's transmissivity. The reduced light value is the refracted component.

If a ray does not intersect an object, it goes to infinity and the corresponding direction on the environment map texture will be assigned to this ray.

We did not implement the real front-end, but we implemented a test front-end to test our back-end. The test front-end is used to generate the teaser image of this report. The test front-end is not connected to any IMU, so the pose of the viewer is controlled solely by keyboard. The test front-end does not have any foveation feature. It simply creates a 1920x1080 image and create a ray for each pixel.

5 EVALUATION OF RESULTS

The ray tracer back-end was tested both on my personal laptop and on attu4. We checked that no other users are performing CPU intensive tasks on attu4 at the time of testing.

The test is use the testing front-end to render the teaser scene. We intentionally made the scene to highlight our ray tracing benefit over traditional graphics rendering, including opaque/translucent shadows, reflections, and refractions. From the teaser image, we can

	laptop	attu4
# of cores	8	48
Max MHz	4000	3100
L1d cache	128K	32K
L1i cache	128K	32K
L2 cache	1024K	256K
L3 cache	8192K	30720K
RAM	7.9G	128G
render time	1.5 s/frame	3.8 s/frame

Table 1. Spec and render time of personal laptop and attu4

clearly see that ray traced images are far more realistic than the CSE 490V homework rendering approach.

We also measured the time for the ray tracer to finish rendering each frame. The results and relevant spec of my personal laptop and attu4 are specified in Table 1.

The result is very surprising to us. We expect the render time to be lower on attu4 than on my personal laptop, since it has much more cores and is more computationally powerful. After some experiments, we have a few hypothesis why the ray tracer performance is lower on attu4:

- Cache size: If we compare the hardware specification of two machines, attu4 has significantly smaller L1 and L2 caches. Since in ray tracing, we need to compute color over a large image buffer, the personal laptop may benefit from larger cache sizes and fewer cache misses.
- NUMA: The attu4 server has two NUMA nodes, each containing 24 cores. Since the tasks of computing ray colors are distributed arbitrarily among the CPU cores, there might be a lot of cross-node memory reference, which has a significant impact on performance.
- Locality: Since the tasks of computing the color of each ray is distributed arbitrarily among threads in the thread pool, multiple cores may frequently computerays whose location in the image buffer lies in the same cache line. This will cause multiple cores to synchronize their caches frequently, which also has a great impact on the performance. This hypothesis is supported by the observation that setting a smaller thread limits yields a slightly better results on attu4. To mitigate this issue, we can block the rays into cache lines and distributes computational tasks in the unit of blocks.

We have suspected that sharing the thread pool among ray tracing tasks and WebSocket I/O tasks might caused the performance drop, but later experiments found that the performance drop stays the same without the WebSocket I/O tasks. We have also suspected that too much contention among threads caused the performance issue, but the performance drop stays the same after we optimized the code to eliminate all uses of locks.

Even with the better result of 1.5 seconds per frame result, the ray tracer is still far less preferment to be used as a real-time rendering method. Since the image for the left eye and the image for the right eye are rendered separately, each eye can only get 0.3 fps, which is 20x lower than the refresh rate of most displays.

6 FUTURE WORK

The problem with the attu4 performance being lower than the laptop performance indicates that a naive implementation of a multi-threaded ray tracer can easily have unexpected performance issues. Further experiments are needed to verify our hypotheses and identify the exact causes of the performance problem. Some potential solutions include separating I/O tasks from ray tracing CPU tasks into a different thread pool, blocking rays together into cache lines, and running the ray tracer on a server with larger cache sizes. However, how much those solutions will increase the performance, or even whether those solutions will help with the performance at all, is still yet to be tested.

Additionally, with newer hardware such as ray tracing capable GPUs, we use GPU-based ray tracing on a GPU equipped render server to achieve higher performance.

7 CONCLUSION

For this project, we attempted to implement a high performance ray tracer for real-time VR rendering by running the ray tracer on a dedicated server with high computational power. Contrary to our expectation, the performance worsened when the ray tracer is put on a server with larger number of cores. Future investigation is needed to identify the exact cause of this issue.

REFERENCES

[1] Augmentarium, X. M., Meng, X., Augmentarium, R. D., Du, R., Augmentarium, M. Z., Zwicker, M., & Amitabh Varshney Augmentarium. (2018, July 1). Kernel Foveated Rendering.
Retrieved March 19, 2020, from https://dl.acm.org/doi/pdf/10.1145/3203199?download=true
[2] Project 3: Trace. (n.d.).
Retrieved from https://courses.cs.washington.edu/courses/cse457/19sp/src/trace/trace.php
[3] Ray Tracing. (n.d.).
Retrieved March 19, 2020, from https://courses.cs.washington.edu/courses/cse457/handouts/marschner-shirley.pdf
[4] Sponsored Feature: Light It Up! Quake Wars Gets Ray Traced. (n.d.).
Retrieved from https://www.gamasutra.com/view/feature/132320/sponsored_feature_light_it_up_.php