B.Comp. Dissertation

Hardware Ray-Tracing Assisted Hybrid Rendering Pipeline for Games

By

Kim-Chan Tze Hui, Louiz

Department of Computer Science

School of Computing

National University of Singapore

2020/2021

B.Comp. Dissertation

Hardware Ray-Tracing Assisted Hybrid Rendering Pipeline for Games

By

Kim-Chan Tze Hui, Louiz

Department of Computer Science

School of Computing

National University of Singapore

2020/2021

Project No: H1351640

Advisor: Anand Bhojan

Deliverables:

Report: 1 Volume

Abstract

This project creates a distributed rendering algorithm using two devices across the network that combines rasterization and raytracing techniques. This is done to enable the better visual quality that raytracing brings compared to traditional rasterization techniques, to devices that cannot support hardware accelerated raytracing, by offloading the raytracing steps of the real-time rendering algorithm to a server. We describe the architecture of our implementation of a direct-lighting shading algorithm that makes use of distributed rendering and raytracing, as well as directions for augmenting our system for more advanced shading algorithms that support global illumination. Currently, our implementation supports static scenes that allows user input to move the camera. The current implementation's visual quality has need of improvement, requiring the addition of the suggested shading algorithm augmentations before it is ready for production, and the framerate is low at 7-8 fps, and requires the addition of H.264 compression and possibly changing from TCP to UDP to overcome the current network bottleneck. Finally, we discuss the areas of improvement and future work.

Subject Descriptors:

Computer Methodologies--Computer Graphics--Rendering
Computer Methodologies--Computer Graphics--Rendering--Ray tracing
Computer Methodologies--Computer Graphics--Rendering--Rasterization
Computer systems organization--Architectures--Distributed architectures--Client-server architectures

Keywords:

Visual Computing, Computer Graphics, Computer Games, Distributed Rendering

Implementation Software and Hardware:

Windows Version 10.0.19041 Build 19041, DirectX 12, DirectX Raytracing, NVIDIA GeForce RTX 2060

Acknowledgment

I would like to thank my supervisor, Dr. Anand Bhojan, for his guidance and direction.

I would also like to thank Tan Yu Wei and Anthony Halim from the Hybrid Rendering team, Tan Yu Wei for her advice and guidance, along with sourcing for relevant research papers, and Anthony Halim for working on the project with me.

Table of Contents

List of Figures

1. Introduction

1.1. Summary

Our project creates a prototype for a novel distributed rendering algorithm, where an instance of a 3D scene exists on both the client and the server, and inputs from the client are sent to the server to ensure the scenes are synchronized. Rasterization and post-processing passes are performed on the client computer, and only raytracing passes are performed on a server, and the client receives raytraced data from over the network to render the scene. This way, a client computer can have graphics in real-time applications achieve the higher visual quality of raytracing without the need for hardware that supports hardware accelerated raytracing.

1.2. Motivation

Rasterization is the traditionally used technique for the generation of computer graphics in real-time applications, while raytracing can generate more photorealistic images, but at a greater processing cost, and is thus often reserved for offline rendering use (Sabino, Andrade, Gonzales Clua, Montenegro, Pagliosa, 2012). In recent years however, NVIDIA's Turing GPU architecture, which sport RT Cores that accelerate bounding volume hierarchy traversal and ray-triangle intersection testing, has made it viable to utilize raytracing in real-time applications (Champagne, 2020).

This new GPU architecture and advances in technology have enabled the use of raytracing in real-time applications, but traditional rasterization techniques are generally still faster. Additionally, rasterization is available to general devices, even mobile phones, whereas raytracing support is not yet ubiquitous. Even though the hardware to support hardware accelerated raytracing is available to consumers (e.g. NVIDIA's RTX series), the typical consumer may not have the latest technology. Especially when it comes to mobile devices with much lower computing power, raytracing support for real-time applications is not yet possible.

Even with supported hardware however, current commercial real-time engines (e.g. Unity's HDRP, Unreal Engine) that use raytracing do not purely use raytracing, but a combination of the faster rasterization techniques and raytracing, along with post-processing denoisers (Anis, 2019, 2020). This "hybrid rendering" approach allows for an improvement of visual quality without the computational cost of full-on raytracing as is used in offline rendering. We would like to introduce a cloud aspect to this hybrid rendering – our project aims to make use of hardware available on a cloud for raytracing, while performing only rasterization/post-processing on the client's computer, to achieve an improvement in visual quality. Although

this may have not been optimal due to latency from sending data across large distances, with the rise of edge computing and 5G networks, this will become more feasible in the near future, as cloud computing power will become available to users at lower latency (Gyarmathy, 2019).

This project's major goal was to produce a novel algorithm for hybrid rasterized/raytraced rendering over the cloud, which also involved building the framework that will be used to implement a rendering pipeline that makes use of various rasterization and raytracing techniques. In the future, we would like to integrate the rest of NUS's novel raytraced post-processing effects (including hybrid motion blur and depth-of-field effects) as well.

1.3. Cloud-based Rendering

Our novel rendering technique involves "distributed rendering", which is the distribution of rendering workload to multiple devices. The main motivation of this is the rise of 5G and edge computing. With 5G networks achieving very low latency and high bandwidth, having high amounts of data flow will become much more viable, and edge computing can reduce the latency compared to currently available cloud gaming services. Already, we have services that allow users to play games on the cloud. An example is Google Stadia, which takes the user's inputs and streams games to consumers' computers/phones/tablets. Another is Playstation Now, which allows you to play PS3 and PS2 games streamed to your PC. In these services, games are completely run on the cloud, and the user simply sees the output streamed like how videos are live streamed. The games are not unplayable, but latency is noticeable (Nelius, 2019).

Our project explores the possibility of using the cloud not to completely render a scene (whereby the client's computer does not render graphics and only streams the rendered game that is run on the server), but we wish to split the work between the server and the client's computer. Instead of the entire game running on the server as in the examples mentioned above, the game would run on the client's hardware (although the scene must be synchronized across the client and server), with only rendering partially offloaded to the cloud, not game logic. This would make the use of cloud rendering much more general, and possible with much lighter server load.

We will further explore the techniques currently used for distributed rendering in the related works section.

1.4. General Overview of Rendering Algorithms

It is useful to briefly describe the difference between rasterization and raytracing. The following algorithms described are simplified versions of typical rasterization and raytracing

algorithms (omitting major details such as culling), just to illustrate the difference in computing power required. We assume a 3D scene is made up of triangles.

In the rasterization algorithm (Algorithm 2), for each frame, we loop through all triangles in the scene, and render them to the screen at their respective coordinates, overwriting any previously drawn triangles if a later triangle's pixel coordinate is closer to the camera. This typically only involves matrix multiplications and float comparisons to choose which triangles appear in the final render. In the raytracing algorithm (Algorithm 1) however, we can see that there is a much more computationally intensive step – finding the closest intersection of a ray with a triangle for all triangles in the scene, which is done per pixel. Step 1.2 of algorithm 1 is typically not done in an O(n) fashion by testing every triangle with the ray, but is done using acceleration structures (e.g. bounding volume hierarchies (BVH)) which can reduce the time complexity to O(lg(n)), but traversal of BVH structures and the reconstruction of BVH's per frame is still much more computationally expensive. NVIDIA's Turing architecture dedicates hardware to performing these intersection computations, which significantly speeds up raytracing and makes it viable for real-time applications (Champagne, 2020).

Algorithm 1: Simplified Raytracing	Algorithm 2: Simplified Rasterization

1. For every pixel
  1. Construct a ray from the camera to that pixel
  2. For every triangle in the scene
  1. Find intersection with the ray
  2. Keep intersection data if closest
  3. Shade pixel with the closest intersected triangle, depending on light and normal vector.
	1. For every triangle in the scene
	  1. Compute vertex coordinates in camera space (the camera's perspective frame)
	  2. Perform rasterization - determining which pixels the triangle occupies (filling up the triangle)
	  3. For every pixel the triangle occupies
	  1. If it is closer compared to any triangle previously occupying this pixel, shade the pixel depending on the light and normal vector

Currently, hybrid rendering algorithms exist that make use of a combination of both rasterization and raytracing. More details will be mentioned in the related works section. The main idea to be highlighted here is that if a pass "makes use of rasterization", we mean that we perform rendering per triangle in the scene, whereas if a pass "uses raytracing", it means that the expensive ray-triangle intersections with the scene are computed per pixel.

2. Study of Related Works

2.1 Improving Pure Cloud Gaming

There are several distributed rendering approaches. Seong and Ngai (2014) describe a mobile cloud gaming architecture called “Layered Coding”, whereby most of the game logic is performed on the server, inputs are sent from the client to the server, and both the server and the client perform rendering. The server renders the game once at full quality, and once for a low-quality version, and the client renders the low-quality version as well, and the server sends the difference after H.264/AVC encoding, and the client combines the base layer with the difference to obtain a high-quality render.

Chan, Ichikawa, Watashiba, Putchong and Iida (2017) describe instruction-based streaming, whereby the game is primarily run on the server, but graphics commands are streamed to the client to render a lower quality image, and partial rendering is performed on the server and streamed to the client, overlaid on the client’s output. Another work (Jurgelionis et al, 2009) has a server execute games but send graphics commands to the client, and the client does the rendering.

A common theme in these described architectures is that they are focused on cloud-based gaming – these methods are done to reduce the required network bandwidth of pure cloud gaming (whereby the game is run on the server and the result is streamed to a client as a video) by making use of the client’s hardware, but this is not the same angle we are looking at, as we are looking more at using a server to offload general rendering that would originally be done on a client’s computer.

2.2. Distributed Rendering Algorithms

Eduardo et al (2015) describes “delta encoding”, which is similar to the work by Seong et al (2014), but has three instances of the game running, one low-resolution version on both the client and server, as well as an additional high-resolution version on the server. Inputs will be sent from the client to the server, all three games will be rendered in sync, and the difference between the server’s high and low-quality renders will be sent to the client to combine with the client’s low-quality render to obtain a high-quality render on the client device. Eduardo et al (2015) also describes using H.264 encoding for compression.

For our approach, we will have two instances of the scene (we are not running full games for now), one on the server and one on the client. Instead of rendering a low/high quality version and sending the difference, we will send information derived from raytracing and send that to

the client for subsequent shading. Making use of H.264/AVC instead of sending whole frames will help to decrease the latency, but at the moment, we are still at an early stage and have not yet included this optimization.

Eduardo et al (2015) also has another technique, which is to have the client and server each render a subset of the frames, which is not immediately relevant to us, as we assume that the client's hardware cannot make use of raytracing.

Chen and El-Zarki (2018) describe trade-offs in collaborative rendering in layered coding, in that having the client render at a lower level of detail requires greater bandwidth for delta data transfer, and if it can render at a higher level of detail, it will have a lower bandwidth requirement. It also offers methods for dynamically changing the level of detail depending on bandwidth. These can be further explored in the future.

2.3. Hybrid Rendering Algorithms

Hybrid rendering algorithms that combine raytracing and rasterization have implementations available in commercial game engines. An example is Unity's High-Definition Rendering Pipeline (Anis, 2019, 2020), which creates a rasterized GBuffer and performs a rasterized shadow pass, and uses raytracing for area shadows, directional shadows, ambient occlusion, reflections, and indirect diffuse. It then performs a deferred lighting pass, which takes the aforementioned passes and combines them to create an output frame, followed by further effects on top of that.

Our current pipeline is much more primitive, and only has a rasterized GBuffer pass, a raytraced visibility pass for shadows (described in Section 3.2), and a deferred lighting pass that combines the GBuffer and the visibility pass result, to form a simple direct-lighting render. In the future, we suggest making use of a similarly more advanced rendering algorithm, but for our current prototype stage, we are using the method just described, which we detail later in Section 3 Algorithm 3B, which was adapted from Algorithm 3A that was implemented in the tutorial framework by Wyman (2019).

3. System Design

3.1. Raytraced Direct Lighting Shading Algorithm

For our implementation, as it is still an early stage, we perform the most basic of raytraced shading algorithms, simply performing a raytraced shadows algorithm with Lambertian shading – this is direct lighting with no global illumination (color bleeding, reflections etc.). In the future, we can look at adding specular highlights, global illumination and denoising to the pipeline, which we discuss in Section 3.3. The base raytracing algorithm is as shown in Algorithm 3A, which had an implementation available from Wyman (2019).

Algorithm 3A: Original Raytraced Shadows

1. Rasterize the scene to create a GBuffer (Detailed in Figure 1)
2. For each pixel in the GBuffer
    1. Initialize the color *c* to (0, 0, 0)
    2. For each light, *l*, in the room
        1. Trace a ray from the pixel's world position, *wPos* (from the GBuffer), to the light's world position
        2. If the ray is not obstructed by any object
            1. Get the (world space) normal vector of this pixel, *n*, and the diffuse color, *m*, from the GBuffer
            2. *c* ← *c* + Lambertian(*n, m, wPos, l*)
    3. Set the pixel color to *c*

Lambertian(*normal*, *matColor*, *worldPos*, *light*):

1. *toLight* ← normalize(*light.position – worldPos*)
2. *lDotN* ← max(dot(*normal*, *toLight*), 0) *// Clamp to [0, 1]*
3. return *lDotN* * intensity(*light*) * *matColor / π // Light intensity involves distance falloff and light color.*

The first step involves the creation of a GBuffer with classic rasterization as described in Algorithm 2, where in step 1.3.1, we do not output a shaded pixel, but separately output the attributes of the visible point. The GBuffer contents can be seen in Figure 1, which include the world positions of the surface to be drawn at each pixel, the surface normal of the surfaces at each pixel, and the surface's material diffuse color at that pixel (more details in Section 4.4.1).

We then use the GBuffer in conjunction with raytracing per pixel for a sort of deferred shading, with Lambertian shading. For Lambertian shading, as in (Ikeuchi, 2014) with slight modification, the color of a surface rendered by a pixel is given as

$$E = \sum_i \frac{I_i}{\pi} * \rho * \max(dot(\boldsymbol{n}, \boldsymbol{L}_i), 0) * k_{shadow,i}$$

We use the material's diffuse color as ρ for the surface albedo, the light's intensity is given by I (which may be attenuated by falloff), the surface normal is **n**, and the vector from the point

to the light (normalized) is given by **L**; these values come from the GBuffer or the scene's global values. k_{shadow} is obtained via raytracing, with a value of 0 if the light is obstructed and 1 otherwise, so obstructed lights (where an object blocks the surface that is to be shaded from the light) do not contribute to the shaded pixel. The contribution is summed for all lights i.

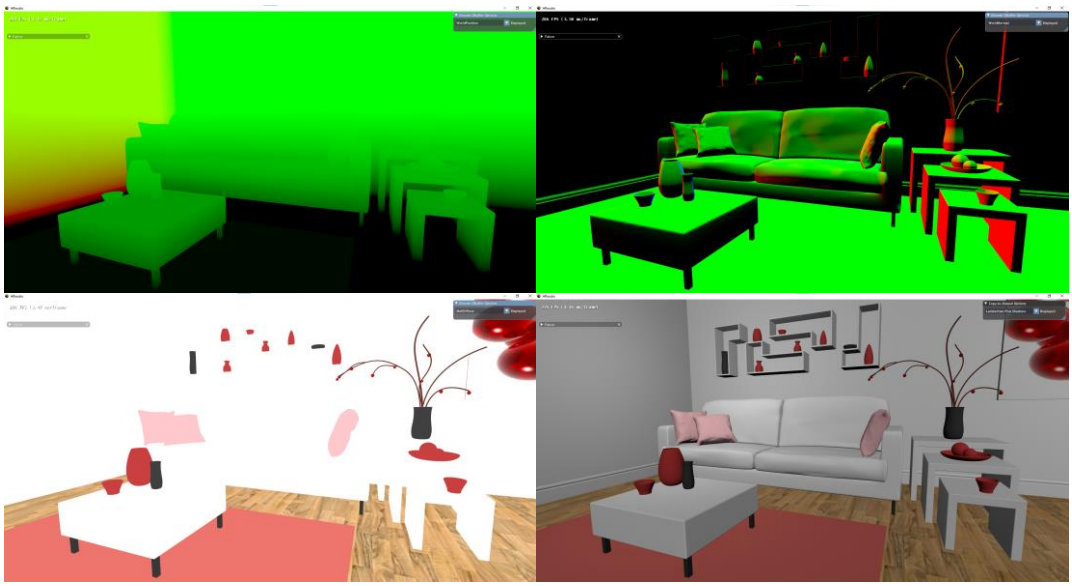

Figure 1: Top-left, top-right and bottom-left show the GBuffer contents. Top-left: position coordinates of the pixel in world space. Top-right: normal vector of the pixel. Bottom left: diffuse color of the pixel. Bottom-right shows the result of shading (without shadows, i.e. ignoring the k_{shadow} term) with Lambertian shading.
3D Scene: "The Modern Living Room" by Wig42 / CC BY

Step 1 of Algorithm 3A is a rasterized step, and we would like to perform this on the client PC, but the raytracing step happens per pixel in step 2.2.1 of Algorithm 3A, where raytracing determines any intersections between the ray with the scene. For our pipeline, we wish to isolate the raytracing step so that we can distribute the rendering between a device that does not support raytracing, and a server that does support raytracing.

The information that is derived from raytracing is a single boolean per light, which is whether that light has unobstructed line-of-sight to the point to be shaded, which in turn is used to determine whether that light contributes to the shading (k_{shadow} in the Lambertian shading equation). In Algorithm 3A, we do this raytracing step as well as shading on the machine, but if we have the visibility boolean of each light per pixel computed by another machine, that information will still be sufficient to perform shading. As such, we modify Algorithm 3A to simply retrieve information from a server regarding the obstruction status in step 2.2.2, to obtain a distributed rendering algorithm.

3.2. Distributed Raytraced Direct Lighting Shading Algorithm

We required a compact and sufficient way to check for each pixel, whether each light is unobstructed, so we use a texture which we call the visibility bitmap, where for each pixel, we represent the visibility of all lights as a 32-bit bitmask. For example, if a pixel's visibility bitmask is given as '...0110', this means that lights 1 and 2 (from the right) are unobstructed, while lights 0 and 3 are obstructed and will not contribute to the shading. In other words, in this example, in the Lambertian shading equation, $k_{shadow,0} = k_{shadow,3} = 0$, while $k_{shadow,1} = k_{shadow,2} = 1$. An illustration of the visibility bitmaps can be seen in Figure 2.

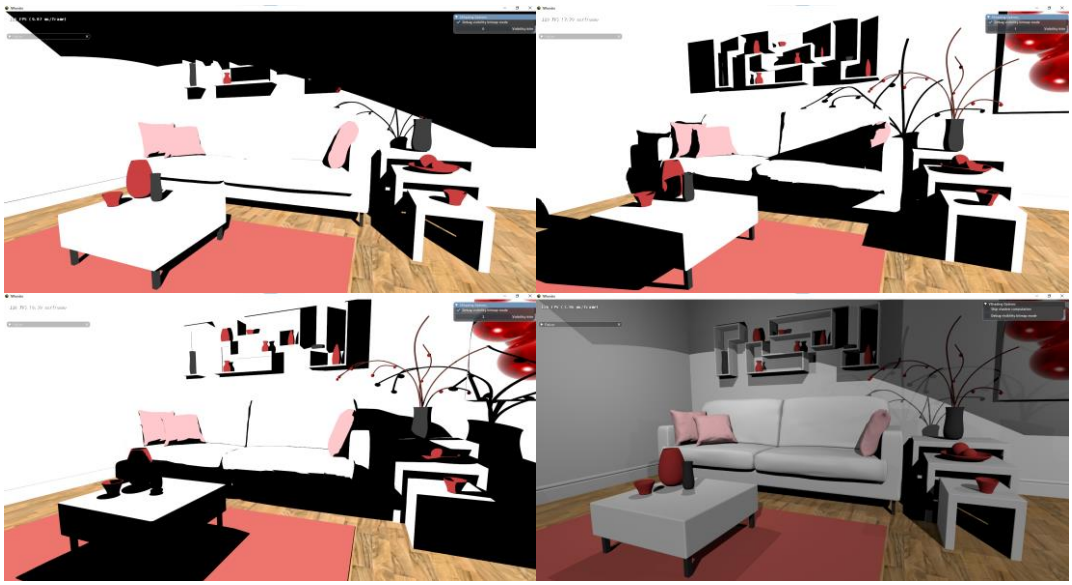

Figure 2: Illustration of visibility bitmap. The scene has three lights, the upper/left images illustrate the visibility bit per pixel. In the upper-left image, if light 0 is unobstructed from the surface shown by a pixel, the material color is shown, otherwise the pixel is black. Similarly for light 1 with the upper-right image, and light 2 with the bottom-left image. Visibility for all three lights are stored as a bitmask per pixel in a single 32 bit unsigned integer. Bottom-right image shows the result of Lambertian shading with all three lights.

Thus, we can split up Algorithm 3A to have different parts done on different devices. The architecture for our distributed shading system is detailed in Figure 3, and the details of per-frame rendering are explained in Algorithm 3B. We require the both the device and the server to have a copy of the scene that is being rendered that should be synchronized, hence inputs/scene changes should be sent before a frame is rendered (step 1 for both the client and server). Both will render the rasterized GBuffer (step 2 for both the client and server). After that, the server will perform raytracing to compute the visibility bitmap (steps 3-4 for the server), which it will send to the client (step 3 for the client algorithm, step 5 for the server). The client device will then make use of the visibility bitmap, along with the GBuffer it had computed, and produce the shaded screen image (step 4 for the client) in a "post-processing"

pass, i.e. there is no rasterization occurring; the GPU simply applies operations on the input textures and fills the output texture. The algorithm is detailed in Algorithm 3B.

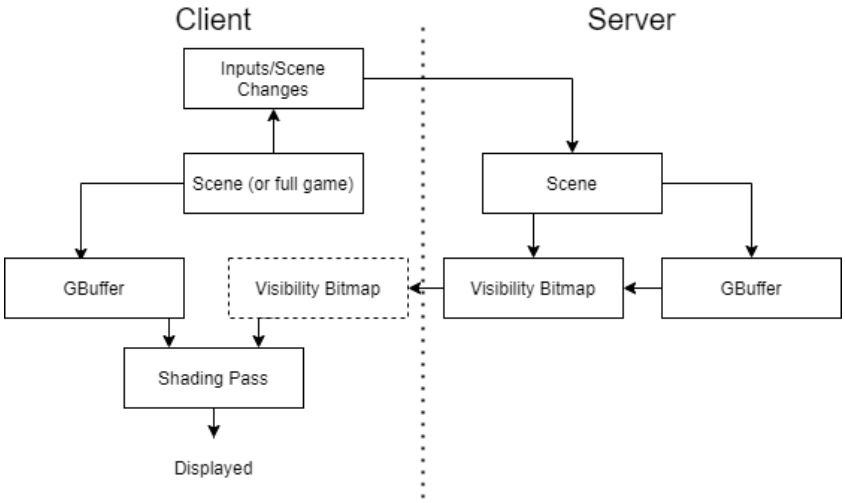

Figure 3: Distributed Raytraced Shading Rendering Architecture Diagram

Algorithm 3B-1: Client side raytraced Shadows

1. Send inputs/scene changes to server to synchronize scene
2. Rasterize the scene to create a GBuffer
3. Receive VisibilityBitmap from server
4. For each pixel in the GBuffer
    1. Initialize the color *c* to (0, 0, 0)
    2. For each light, l_i, in the room
        1. If the boolean for the light in VisibilityBitmap is true (for visibility bitmask *v* for this pixel in the VisibilityBitmap, check *v* & (1 << *i*))
            1. Get the world space position of this pixel, *wPos*, the normal vector of this pixel, *n*, and the diffuse color, *m*, from the GBuffer
            2. c ← c + Lambertian(n, m, wPos, l)
            3. Set the pixel color to *c*

Algorithm 3B-2: Server side raytraced Shadows

1. Receive inputs/scene changes from client to synchronize scene
2. Rasterize the scene to create a GBuffer
3. Initialize the VisibilityBitmap texture to 0's
4. For each pixel in the GBuffer
    1. Initialize visibility bitmask *v* to 0
    2. For each light, l_i, in the room
        1. Trace a ray from the pixel's world position, *wPos* (from the GBuffer), to the light's world position
        2. If the ray is not obstructed by any object, *v* ← *v* | (1 << *i*)
    3. Set the pixel value in VisibilityBitmap to *v*
5. Send the VisibilityBitmap to the client

This architecture design does include a redundant rendering of the GBuffer (two copies, one on the server and one on the client device). An alternative design for distributed rendering

would be to have the client render the GBuffer and send it to the server for the server's rendering passes, or to have the server compute the GBuffer and the visibility bitmap and send both to the client for its rendering. The reason for our current implementation is that for our experiments, the sending of textures over the network was the performance bottle neck, and not the rendering of the GBuffer, so we have chosen this design although there is redundant computation, since the GBuffer is rather large to send over the network (the full GBuffer for a 1920x1080 render would be around 83MB. The visibility bitmap itself is around 8MB).

We will discuss our implementation of this algorithm in section 4.

3.3. Extensibility of Architecture

The system design used in this project is not the final product, as the rendering algorithm is rather primitive. For this rendering pipeline to be useful in games, we would want to incorporate other effects that are commonly used, such as specular highlighting, global illumination (reflections and diffuse global illumination), and post-processing effects such as bloom.

For adding specular highlighting (using the GGX equation as mentioned in (Walter, Marschner, Li and Torrance, 2007) for example, or using the Phong Illumination equation by Phong (1975)), we do not need any further raytracing information, and we simply need to modify the shading equation we use in Algorithm 3B-1 step 4.2.1.2, with no modification to the system architecture necessary.

For global illumination, a lot more information needs to be sent from raytracing than just a single boolean. For diffuse global illumination, we trace a ray (or multiple rays) in a random direction(s) and contribute the (shaded) color of the closest surface that this ray intersects to the shaded point – this results in color bleeding, e.g. a white wall with a red object next to it should be shaded as slightly red. We can also obtain specular global illumination by doing something similar, but we trace a ray with a random distribution given by the GGX lobe as in Walter et al (2007), which gives us a higher likelihood (depending on the reflectivity and roughness of the surface) to sample rays that are reflections of the ray from the eye to the surface, which gives us reflections. For this, we must send the result of raytracing as a full color bitmap, as the color contribution from global illumination requires a full color vector per pixel, and not just a Boolean bitmask. We would then add the result of global illumination to the result of direct lighting which is covered by algorithm 3B (which still requires sending the visibility bitmap as well).

If we are sending colored bitmaps already, it may seem like there is no point in performing distributed rendering, and we might as well perform the full rendering on the server. However, the client device will still need to compute the direct shading result, as well as perform denoising. Images resulting from raytraced global illumination are typically not immediately usable. When rendering an image via raytracing, we typically require dozens to hundreds of samples per pixel to achieve noise-free results. To achieve framerates of 30Hz with a 1920×1080 viewport however, we are limited to just a few rays per pixel due to current hardware limitations (Schied et al, 2017). As such, reconstruction filters which can denoise low samples-per-pixel frames are necessary to achieve non-noisy ray-traced renders with passable visual-quality in real-time. Denoising can be done with Spatiotemporal Variance-Guided Filtering as mentioned in (Schied et al, 2017), which involves convolutions with a bilateral filter, and this can be done on hardware without raytracing capabilities, but it is relatively computationally intensive. As such, there are still benefits of distributing the rendering and not just streaming a fully rendered scene. By only performing global illumination raytracing and sending the visibility bitmap, the server will not need to perform the direct lighting computation, as well as denoising. This would allow such a server to have the computational capacity to serve more concurrent render jobs to different clients.

The global illumination discussion thus far can also be applied to recursive raytracing, which contributes the color of surfaces from more than one bounce.

For post-processing effects, it is simply a matter of adding an additional pass after the shading passes, as this is post-processing. This does not require assistance of the server, and mobile hardware can perform these operations, as they do not require raytracing.

Compared to Figure 3, Figure 4 shows the proposed extended architecture with the changes discussed here. For our implementation discussed in the next section, we implement the simplified version of Figure 3, but this is suggested for future work.

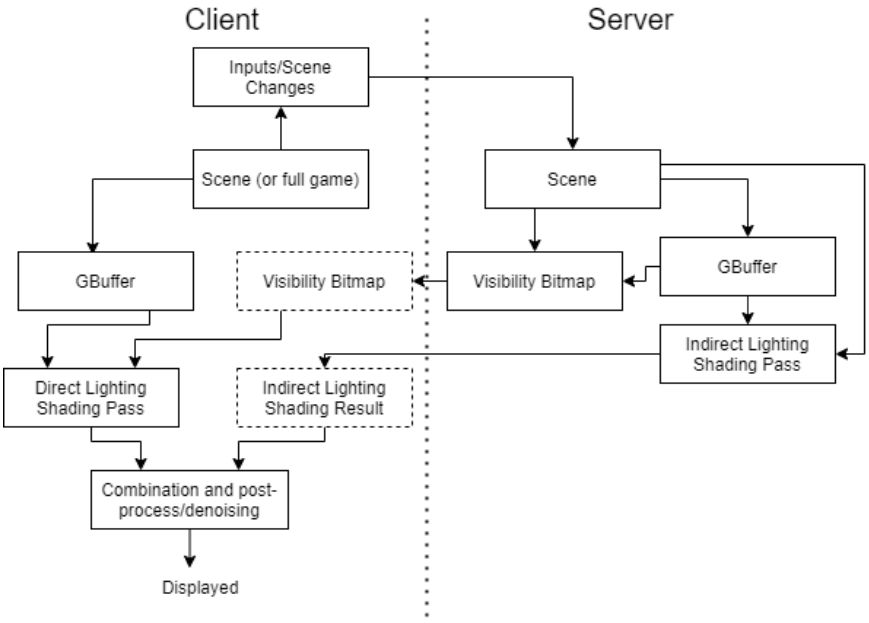

Figure 4: Proposed Extended Rendering Architecture Diagram. "Combination and post-process/denoising" can be implemented as multiple passes, and we can have multiple "indirect lighting shading passes", but they are illustrated as single boxes for brevity

4. Implementation

Now we discuss the implementation of the distributed rendering algorithm, Algorithm 3B.

4.1. Hardware and Framework

The premise of the project relies on hardware-accelerated raytracing (on the server). We are making use of NVIDIA's Turing hardware (RTX series) which supports this. We are also performing distributed rendering, which requires two devices connected over a network.

The implementation is built on NVIDIA's Falcor framework, which is an abstraction layer on top of DirectX 12 and DirectX Raytracing (DXR). As DirectX is a low-level API, the use of Falcor greatly speeds up prototyping implementation. It provides numerous features, the most relevant of which include:

- DirectX context creation and management
- Window and input handling
- Shader program compilation and management
- 3D asset management
  - 3D scene file input/output
  - Construction of DXR acceleration structures
  - Convenient interface for toggling animation, camera movement, light/material properties
- Easy to use GUI framework

Building a modular rendering pipeline, even on top of Falcor, would still take a very long time. As such, our implementation takes its base code from the DirectX Raytracing tutorial by Wyman (2019). The tutorial makes use of the provided RenderingPipeline framework which was built on top of Falcor to provide a further abstraction layer. The provided framework was written in Falcor 3.1.0, and had to be migrated to Falcor 4.3. Performing rendering steps is abstracted into RenderPasses, which have a C++ portion to interact with other passes, acquire resources, and prep the GPU for the shader portion of the RenderPass that performs shading algorithms on the GPU (written in High-Level Shading Language).

An alternative design choice would be to build on top of Falcor's native rendering pipeline framework, Mogwai. However, this pipeline would involve compiling each render pass as a separate C++ library, and pipelines are constructed using a Render Graph Editor which builds Python scripts to store the pipeline information. I felt that this would be more unwieldy and was not as flexible. It also does not provide the same level of abstraction as the

RenderingPipeline framework, which would mean that we either must work with more code repetition, or re-implement that abstraction. With the RenderingPipeline framework, it is also more straightforward to pass resources (e.g. textures, vectors) between render passes, modify the system (e.g. adding network functionality and threading operations), and make quick changes in the interfaces of the code (native Falcor requires the use of the Render Graph Editor).

4.2. High Level Details

4.2.1. Data Flow

Figure 5 gives an understanding of the data flow in the program. There are four memory locations, the client's RAM (accessed by the CPU) and VRAM (accessed by the GPU), and the server's RAM and VRAM. The client and server must be connected via a network. Refer to section 3.2 for the system architecture and details of what is computed on each device.

The information transfer flow (as in Figure 5) is as follows. Before rendering a frame, inputs are handled, and the 3D scene on the client's graphics card (VRAM) is updated. Sending of network data happens during the rendering of a frame, so when a frame begins rendering on the client, we send the changes in the scene to the server. In our implementation, we have yet to include animated (or dynamic) scenes, so these updates only include the camera movement.

The server then inputs these changes into the scene and updates the scene that is stored on the graphics card. The server can then render the GBuffer and the visibility bitmap (detailed in section 3.2), which occurs concurrently with the client computing the GBuffer. The server must transfer the data from its graphics card to its RAM, and then it sends it to the client's RAM, which it can then upload to its VRAM for use by its GPU. The client can then use the GBuffer it had computed, and the Visibility Bitmap received from the server, to perform shading.

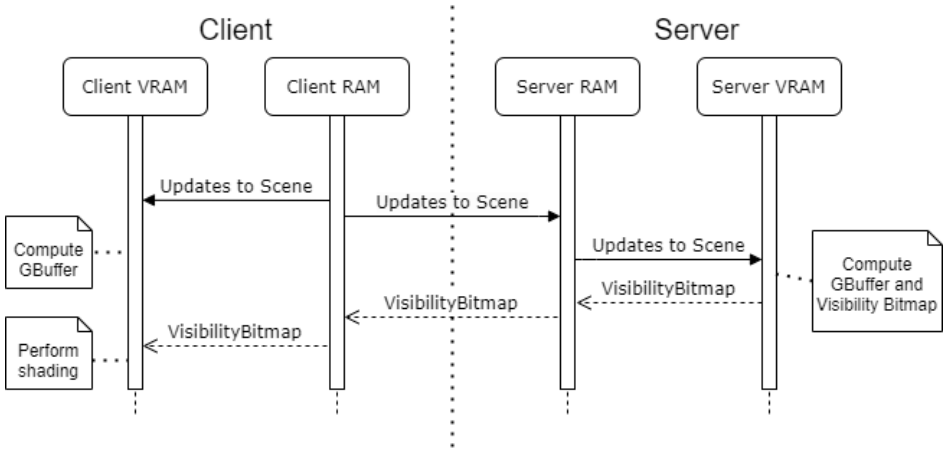

Figure 5: Data Flow Diagram Between Client and Server Memory

4.2.2. System Architecture

We mentioned the architecture rendering architecture in Section 3.2 and Figure 3, which gave an abstract idea of how the rendering is carried out. Here, we detail the program's system architecture. The architecture, detailed in Figure 6, is based on the RenderingPipeline framework mentioned in Section 4.1.

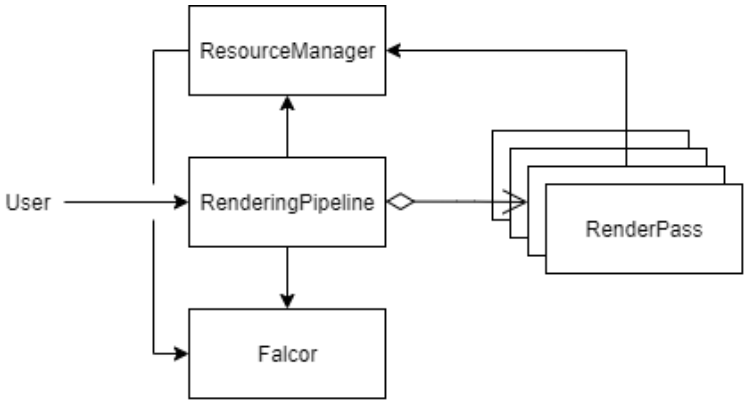

Figure 6: System Architecture of Implementation Program.

The ResourceManager interacts with Falcor to handle the initialization and access of GPU resources such as textures (bitmaps in GPU memory), and the scene mesh information, so that other components do not need to directly interact with Falcor.

The RenderPasses will get GPU resources via the ResourceManager. The RenderPasses each perform a rendering step, where the C++ portion typically involves the retrieval and setting of values in GPU memory, while the shader portion performs shading algorithms and outputs values to a texture (the texture can then be blit to the screen for display by the final pass).

The RenderingPipeline handles the main running of the program by implementing methods for Falcor to call and letting Falcor run. RenderingPipeline is initialized with RenderPasses specified in order, which we detail in Section 4.3 and Section 4.4. Falcor handles things such as window creation, receiving input, and the main "event loop", which is an infinite while loop that continually handles inputs, updates the scene, renders the scene, along with other processes. Falcor handles these by calling the methods in the IRenderer interface which the RenderingPipeline implements. As such, we have a lot of control over what happens on input events, on each frame's scene render, as well as GUI renders, by specifying these in the RenderingPipeline's methods.

Essentially, for each of these events, the RenderingPipeline will run a for loop through all the RenderPasses it was initialized with, calling their respective methods (e.g.

onKeyEvent/onMouseEvent, onExecute, onGuiRender etc.) in order. So, for each frame, each selected pass's onExecute (which runs its C++ code and shader code) method is run in the order the RenderingPipeline was specified with, and this creates the final image.

The render passes in our system, which we will go into detail in the next section, are as follows:

Client	Server
1. NetworkPass (Send inputs) 2. GBufferPass 3. NetworkPass (Receive VisibilityBitmap) 4. MemoryTransferPass (CPU to GPU) 5. ShadingPass 6. OutputPass	1. NetworkPass (Receive inputs) 2. GBufferPass 3. VisibilityPass 4. MemoryTransferPass (GPU to CPU) 5. NetworkPass (Send VisibilityBitmap) Concurrent with rendering passes: 1. "Network Thread"

The control flow of the program per frame is as in Figure 7 (output pass omitted, indicated as a "Display Frame" to prevent clutter). Note that this is the implementation of Algorithm 3B in Section 3.2, although it may look more complicated due to the number of parts. We will first discuss the implementation of the server network thread and the network passes in Section 4.3, as these involve multiple threaded/waiting parts, and then we will discuss the individual rendering passes in Section 4.4. Section 4.5 will discuss some other miscellaneous details.

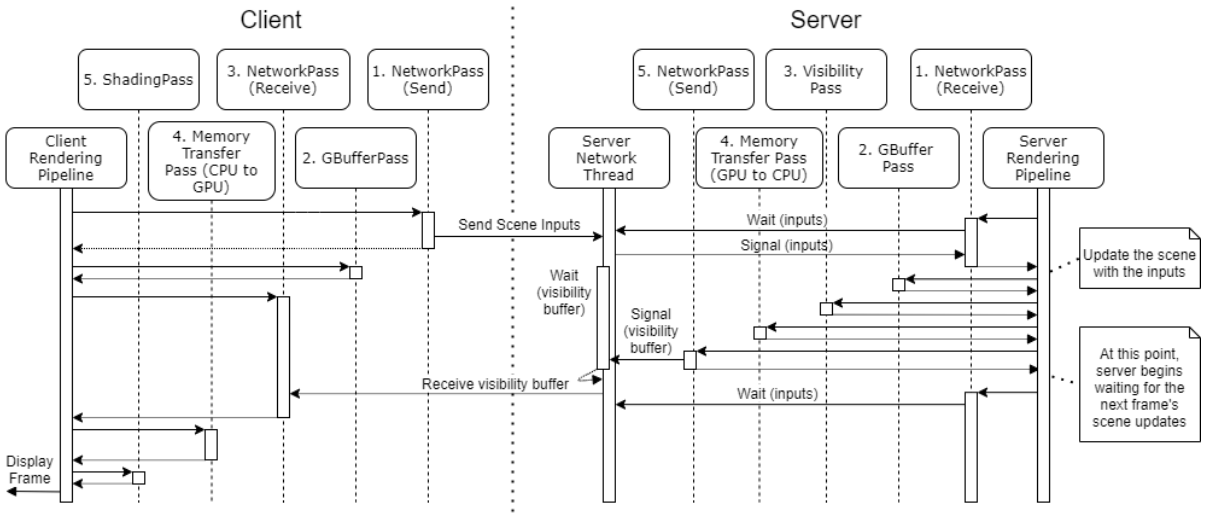

Figure 7: Control Flow Diagram of Render Passes.

4.3. Network Passes and Threading

There are four network passes and one network thread altogether in Figure 7. On the client, there are two network passes called in sequence with the rest of the rendering passes. This is a direct translation of Algorithm 3B-1, with one extra step (pre-4). We give the details with emphasis on the network passes here:

- 1. First network pass - send the inputs to the server (3B-1 step 1)
    - In our implementation thus far, we only deal with static scenes. As such, the only scene updates will be from the camera movement. We thus just send the camera's position, target, and up vectors
- 2. Rasterize the GBuffer (3B-1 step 2)
- 3. Second network pass - receive the VisibilityBitmap from the server (3B-1 step 3)
    - This step must occur before we can execute the next steps, hence as illustrated by the longer bar in Figure 7, this pass will wait until it receives the VisibilityBitmap from the server before the rendering can progress.
- 4. Memory Transfer pass - VisibilityBitmap from RAM to VRAM (3B-1 pre-step 4)
    - After receiving the bitmap, it must be uploaded to the graphics card's memory for use by the shading pass
- 5/6. Perform the Shading pass and the output pass (output to screen) (3B-1 step 4)

As for the server, it is not as direct a translation from Algorithm 3B-2, because we make use of a second thread to constantly listen on the network. This second thread is created during the first frame's rendering with Falcor's thread management, and exists for the rest of the program execution. As such, all actual network receiving/sending on the network is done by the network thread, and the network passes simply wait/notify that thread with conditional variables (synchronizing variables similar to semaphores, whereby calling cv.wait() will lock the thread until another thread calls cv.notify()). The details with emphasis on the network passes are as follows:

- Network thread – handles receiving of scene inputs and sending of the visibility bitmap
    - When the thread receives the scene inputs from the client, it notifies the conditional variable for cameraDataReceived
    - The thread then waits for the conditional variable for visTexureComplete to be notified
        - It must not attempt to send the visibility texture before its rendering is completed and uploaded from the GPU to CPU memory
    - It then sends the visibility texture to the client
- 1. First network pass – receive inputs from the client (3B-2 step 1)
    - The network pass thread does not actually perform the network interaction, unlike for the client's network pass. This pass simply waits on the cameraDataReceived

conditional variable until the network thread notifies it. This is for synchronization, to prevent rendering from occurring until the scene can be updated.
    - It then uploads the camera updates to the VRAM.
- 2. Rasterize the GBuffer (3B-2 step 2)
- 3. Visibility Bitmap pass – render the visibility bitmap (3B-2 step 3, 4)
- 4. Memory Transfer pass - VisibilityBitmap from VRAM to RAM (3B-2 pre-step 5)
    - After receiving the bitmap, it must be retrieved from the graphics card's memory to the RAM for the network thread to access it and send it to the client
- 5. Second network pass – sends visibility bitmap to client (3B-2 step 5)
    - Like the first network pass, it does not actually perform the network interaction. It simply notifies the conditional variable for visTextureComplete to allow the network thread to send the data to the client

This gives a big picture of the network interaction. Next, we zoom in to the network process.

4.3.1. Network Protocol Details

For our implementation, for ease of implementation, we use TCP instead of UDP. Although this does increase overhead and latency, this project is still in its early stages, and this way we could avoid having to implement error checking. In the future we should move to UDP.

For our C++ implementation, we made use of the Winsock client and server code available from Microsoft Docs (2018) as a base and modified it according to our purpose.

For now, the network protocol is not very robust, and is very sequential, and requires the server to be started before the client. Prior to the server rendering its first frame, the server establishes a connection with the client, and waits for the client to send its window width and height (which happens during the client's first frame). The server then initializes its textures with the specified width and height. For now, our implementation does not support resizing of the client window. The server then enters an infinite while loop, where it receives the camera data and sends the visibility bitmap. For receiving the camera data, it simply receives data from the socket until enough data to fill an std::array<float3, 3> is received, and the data is cast from a char* to that type. To send the visibility buffer, it simply sends the result extracted from the VRAM, and on the client's side, it just awaits until (width * height * 4) bytes are received, as each pixel of the visibility bitmap occupies 32 bits, and uploads that to the VRAM.

4.4. Render Pass Details

Now that we have gone through the network passes/thread where there is a lot of interaction, we will go through the implementation details of render passes that perform individualistically.

4.4.1. GBuffer

The GBuffer pass is a rasterized pass that occurs on both the client and the server. It performs rasterization as in Algorithm 2 in Section 1.4, but instead of outputting a shaded pixel output in step 1.3.1, it outputs multiple textures, of which some can be seen in Figure 1. The result of rasterization of each triangle in the scene results in three textures:

- World positions
  - Each pixel occupies 16 bytes – 4 bytes per dimension (x, y, z, w)
    - w is usually set to 1, but if no geometry was rendered at that pixel, it will be 0.
- World normal
  - Each pixel occupies 8 bytes – 2 bytes per dimension (x, y, z, w)
    - w is used to store the distance of the camera to the point (not necessary for our purposes for now)
- Material information
  - Each pixel occupies 16 bytes – 1 byte per value (these values take values of [0, 255])
    - Each original float is converted to a uint8, and four such uint8's are stored in each 32-bit float of the float4 texture. We pack 16 numbers – diffuse color (RGB), specular color (RGB), emissive color (RGB), opacity, linear roughness, and a few other properties (that are not relevant for our purposes) – into the float4 texture.

Note that the material information has been compacted, and when used, requires decoding the values. E.g. RGB diffuse colors and opacity are stored in a single 4 byte value per pixel, but for use, these need to be extracted to 4 floats (each occupying 4 bytes). These compression/extraction methods can be found in the packingUtils.hlsli. The main purpose for this was to use as few textures as possible, to reduce memory cost and improve performance. Another practical issue is the limited maximum number of render targets allowed in a single pass (for RTX 2060, this is 8), which becomes a limiting factor when information such as motion vectors, normals, and derivatives etc. are needed for spatiotemporal variance guided denoising support as in (Schied et al, 2017). For our implementation now, we do not require so many render targets, but in the future, these may become necessary.

These three textures are sufficient to shade the scene as seen in the bottom-right image of Figure 1 (without shadows, which requires another pass to determine whether the lights in the scene reach each point). The textures are all accessible to other render passes via the ResourceManager (see Section 4.2.2).

4.4.2. Visibility Pass

The visibility pass is a raytraced pass that occurs only on the server. It performs the raytracing steps, steps 3-4 of Algorithm 3B-2. Via the ResourceManager, it makes use of the world position texture of the GBuffer, and traces a ray to each light for each pixel from the position specified in the GBuffer, and outputs the visibility bitmask for that pixel, as specified in Section 3.2 and illustrated in Figure 2.

4.4.3. Memory Transfer Pass

The memory transfer pass makes use of modified methods available in the Falcor library that were originally used to upload image files to the VRAM as textures, and to take screenshots.

For the client side, uploading the data received from the server to the VRAM occurs, then the shading pass can execute, and this is synchronous, as the client thread will wait for upload completion before continuing. On the server side, retrieving the data from the VRAM to the RAM is asynchronous with the network thread. The network thread simply waits until its conditional variable is notified, and when the retrieval from the VRAM is complete, the memory transfer pass notifies the conditional variable, allowing the network thread to continue.

4.4.4. Shading Pass

The shading pass is a post-processing pass that technically does not perform rasterization nor raytracing, it simply performs per-pixel operations on textures. It makes use of the VisibilityBitmap, and all three textures of the GBuffer, and executes step 4 of Algorithm 3B-1 directly, outputting the result to a texture.

4.4.5. Output Pass

This pass simply copies the result of the shading pass to the output buffer to be shown on the window.

4.5. Miscellaneous Details

4.5.1. Running Modes

Figure 7 gives an idea of separate passes being run on the server and client programs, but in actuality there is only one compiled program. To run it as a client, simply add the command

line argument “client” (to run Algorithm 3B-1), and similarly with “server” to run it as a server (and running Algorithm 3B-2). Running with neither argument will run it as a standalone with no network/memory passes, and simply performing algorithm 3A.

4.5.2. First Frame

During the execution of the program, for our current prototype, initialization occurs partially during the rendering of the first frame, rather than before. This means that the first rendered frame of the client takes particularly long.

The connection with the server socket occurs before the client’s first frame begins rendering. During the first frame, the client sends the height and width of the window, and then begins rendering its first frame. The server only initializes its textures after the client sends its height and width, so it begins rendering its first frame only while the client has already began rendering its first frame. After the first frame, the control flow in Figure 7 takes over.

4.5.3. Compression Mode

For experimental purposes, we have added a primitive compression mode. In the future, we wish to use H.264 encoding, but this was just for experimentation. This mode makes use of Lempel-Ziv-Oberhumer compression (Oberhumer, 2017), which is a lossless data compression algorithm. The main reason for picking this compression algorithm was the ease of implementing it in code, since it is available as a header only C library. By default, this compression method is used, but can be turned off using the command line argument “no-compression”.

This is not an optimal encoding algorithm for video streaming, as it does not make use of previous frames and sending deltas, which is used by H.264, rather it compresses each frame’s visibility bitmap individually before sending it over, and it is decompressed before being used.

Adding this compression algorithm actually slowed down our overall framerate, as the time taken for compressing the texture, sending it over the network, and decompressing ended up taking up more time than just sending the full texture over the network (slowed down by ~4.3% in our early tests). As such, we should not use it, and recommend using another (H.264) algorithm in the future.

5. Analysis

The implementation was done on two devices both sporting NVIDIA RTX graphics cards which support hardware accelerated raytracing. Although the client does not technically need this, we did this for ease of setup, as the framework we used uses DirectX Raytracing, so setting up compilation without hardware support could become unnecessarily difficult. The two devices were connected with wired ethernet connections to a home router.

5.1. Visual Quality

The visual quality is generally acceptable, but this is not a final state, as the rendering algorithm requires more steps to obtain the level of polish expected in games. Figure 8 shows a comparison of our algorithm with the global illumination algorithm described in Section 3.3 (not run as a distributed rendering – it is performed on a single machine). With that global illumination algorithm, we make a comparison to what a single bounce of raytracing can add compared to our current implementation.

The main differences were highlighted in Section 3.3. The first is the lack of specular highlighting, e.g. the shiny spot on the chair leg in the bottom image of Figure 8, the brighter regions of the flower pots in the top image. There also are no reflections, e.g., in the chair leg in the bottom image. Since there is no global illumination to result in light bouncing around the room, we also have perfectly dark shadows when no light source reaches a point. This is visible under the tables and sofa in the top image of Figure 8, and in the whole shadow of the middle image, where the shadows are perfectly black in our distributed algorithm, which is not realistic, as when the rest of the scene is bright, some light will bounce to the shadowed regions.

The perfectly dark shadows can be somewhat remedied by altering our Lambertian shading algorithm specified in Section 3.1. Again, the original shading equation is given as:

$$E = \sum_i \frac{I_i}{\pi} * \rho * \max(dot(\boldsymbol{n}, \boldsymbol{L}_i), 0) * k_{shadow,i}$$

In our current algorithm, k_{shadow} (obtained via raytracing) is assigned value of 0 if light i is obstructed and 1 otherwise, so obstructed lights do not contribute to the shaded pixel. This results in perfectly dark shadows. We can modify this by adding an ambient lighting term, e.g. adding $\boldsymbol{L}_{ambient} * \rho$ to the final color of the pixel, where $\boldsymbol{L}_{ambient}$ is tuned for the scene, either unconditionally or if all k_{shadow} values were 0. Figure 9, bottom, shows how this can work when adding this term when all $k_{shadow} = 0$, such that our shadows are no longer perfectly

dark, but it does not look very good, as the coloring is flat. The extensions mentioned in Section 3.3 should be implemented to overcome these issues in a more realistic manner.

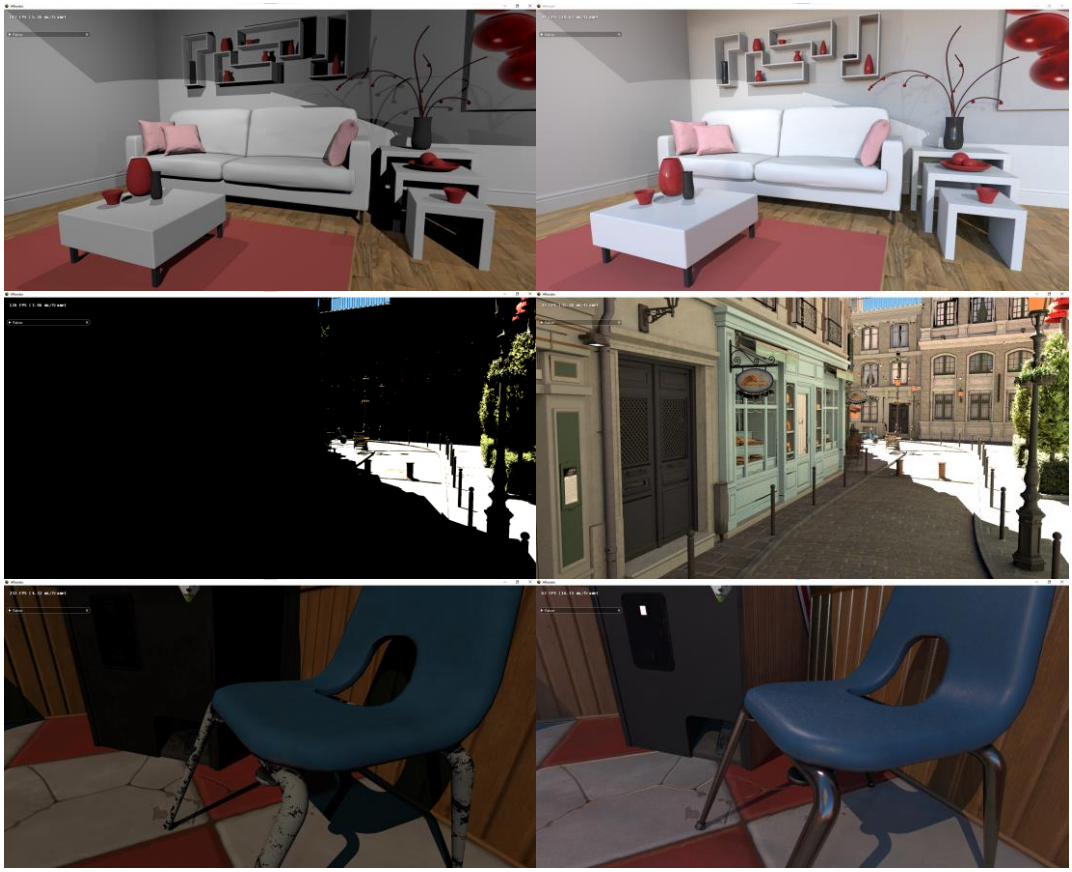

Figure 8: Comparison of our direct lighting shading algorithm with a simple 1-bounce global illumination shading algorithm. Left: Our direct lighting algorithm result. Right: The result of shading with one bounce global illumination. Top: "The Modern Living Room" by Wig42 / CC BY. Middle: "Amazon Lumberyard Bistro"/ CC BY. Bottom: "Arcade sample scene" by Nicholas Hull / CC BY

Figure 9: Result of adding an ambient lighting term. Left: No ambient lighting term. Right: Addition of a tuned ambient lighting term, $L_{ambient}$, for each scene, which adds $L_{ambient} * \rho$ when all $k_{shadow} = 0$ for that pixel.

5.2. Framerate

Our current implementation achieves a framerate of ~7.7fps (130ms/frame) for the pink room scene (as in Figure 9), which is a rather simple scene with three lights. The following table (Figure 10) details the time breakdown of our frame. As some passes occur concurrently (specifically the client's GBuffer pass which occurs concurrently with the server's GBuffer and visibility pass) and are not the time-limiting passes, only the bottleneck passes are shown. Some of these timings will be slightly off, since CPU and GPU tasks can occur concurrently and are timed separately by the Falcor profiler, so these are estimations. Another detail is that our two server network passes were two instances of the same C++ class, and were hence profiled under the same Falcor timer, so we do not have the split between the server's receive and send timings, but the server's network receive pass should be negligible compared to the server's network send pass, since the camera data is only ~0.00043% the size of the visibility bitmap. The time breakdown of the time-limiting passes is as follows:

Pass	Time (ms)	%
GBuffer (Server)	15.38	11.86
Visibility Pass (Server)	8.18	6.31
Memory Transfer (Server GPU to CPU)	13.24	10.21
Network Pass (Server) (both send/receive times are included)	87.33	67.36
Memory Transfer (Client CPU to GPU)	4.14	3.19
Visibility Shading Pass (Client)	1.14	0.88
Output Pass	0.24	0.19

Figure 10: Time Profile of Memory Passes.

Of course, this will vary from device to device, scene to scene, and even from run to run on the same device, but these should give a good ballpark of the relative durations of each pass.

The overall framerate is rather low at the current state. Typically, mobile and console games run with at least 30 fps (60 fps is recommended) (GameBench, 2019), so 7.7fps is not good enough for these applications. Our program is still interactive (ability to move the camera), but the framerate is low.

The main bottleneck right now, taking up 67% of the total frame time, is the network pass. The visibility bitmap being sent over the network per frame is 8.3MB (1920 * 1080 * (4 bytes per pixel)), which is rather large, even over a single hop home router with latency of ~5ms. For future work, H.264 compression so that we do not need to send the full visibility bitmap each frame should help reduce the data sent per frame.

Another issue to note is that we are using TCP instead of UDP so that we would not need to deal with network issues (dropped packets etc.) and implement our own error checking. Swapping to UDP will greatly reduce overhead and improve the speed, which we recommend for future work.

6 Conclusion

6.1 Summary

In summary, we have implemented a novel distributed rendering algorithm for a simple raytraced direct-lighting shading algorithm. In this distributed rendering system, the client device performs rasterized shading passes and passes that process input textures ("post-processing passes") to produce an output frame, and a server device performs raytracing with GPU's that support hardware accelerated raytracing. The details of the shading algorithm are explained in Algorithm 3B in Section 3.2, and the control flow of our system implementation can be found in Section 4.2. The server device creates a GBuffer and produces the visibility bitmap via raytracing, and sends the visibility bitmap to the client. The client device performs its own rasterized GBuffer pass, and a shading pass that makes use of the GBuffer and the visibility bitmap it receives from the server, which creates the output frame.

Currently, the visual quality is useable but not polished, and there is much room for augmenting the current shading algorithm, as suggested in Section 3.3. The framerate is also not good right now, and is limited by the network pass, so working in compression algorithms to the current framework is an essential future work to making this program viable.

6.2 Limitations

Many limitations have already been mentioned in Section 3.3 and Section 5, but there are several other limitations that have yet to be addressed.

Our visibility pass for direct lighting uses a 32-bit visibility bitmask, and this means that the scene can have at most 32 lights. This is a limitation, but not too much of a concern, as typically in games, we would not want to have too many lights in a scene, as this will hit the performance. In games, if a scene requires so many lights, most lights should be baked into the scene, rather than dynamic (i.e. having shadows that update every frame - the way that we handle all lights currently). However, this is still a limitation, but it can be overcome by sending multiple 32-bit visibility bitmasks. Additionally, instead of using a texture that has 32-bits per pixel, we can extend this to up to 128-bits per pixel if necessary. This can be dynamically implemented as well.

As mentioned in Section 4.3.1, the server textures are initialized with the client's window size, and currently no resizing of the screen allowed. This is a limitation, and to overcome this, we would need to make more robust network pass. Instead of just sending the texture, the network

pass should be able to receive messages that e.g., indicate a change in window size, so that these can be changed on the fly.

Currently, we work with a static scene, and the only inputs to the scene for now are camera movements. We also performed our tests with the 3D scene file already available on each device. This should be extended to allow the client to actually send the scene to the server, and we also need to design a pipeline to send changes of e.g., object locations in the scene, to the server, to have this system usable for games.

In Section 5.2 Figure 10, the slowest pass after the GBuffer and network pass is the memory transfer pass. We have discussed that the network pass may be improved. The GBuffer pass is a rendering pass, so we may not be able to speed this up significantly. This leads us to the memory transfer pass. Currently a total of ~11% of the frame time is spent on the memory transfer (~41% of the frame time excluding the network pass). In the future, it would be good to investigate optimizing this.

6.3 Recommendations for Further Work

We have mentioned several areas for improvement in the previous section, and here are some additional recommendations for further work.

As mentioned in Section 2 and Section 5.2, making use of H.264/AVC as in Seong et al (2014) which does not require the sending of entire frames will be helpful in improving the latency and throughput. Additionally, swapping our network protocol from TCP to UDP will also help to reduce the network pass overhead.

As mentioned in Section 3.3 and 5.1, we should augment the rendering algorithm. We can add specular highlighting and global illumination, for which we already have code for a single device, but have yet to implement it in a distributed algorithm. These will require the addition of a denoising pass for real-time application as well.

Finally, when we are satisfied with the quality of the distributed rendering algorithm, looking to test this on mobile devices can be a next step.

References

Anis Benyoub. 2019. Leveraging Real-Time Ray Tracing To Build A Hybrid Game Engine. Retrieved October 30, 2020 from http://advances.realtimerendering.com/s2019/Benyoub-DXR%20Ray%20tracing-%20SIGGRAPH2019-final.pdf

Anis Benyoub. 2020. High Definition Render Pipeline real-time ray tracing is now in Preview. Retrieved October 30, 2020 from https://blogs.unity3d.com/2020/03/06/high-definition-render-pipeline-real-time-ray-tracing-is-now-in-preview/

Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. (2007). Microfacet models for refraction through rough surfaces. In Proceedings of the 18th Eurographics conference on Rendering Techniques (EGSR'07). Eurographics Association, Goslar, DEU, 195–206.

Champagne, Rick. (2020). Real-Time Ray Tracing Realized: RTX Brings the Future of Graphics to Millions. Retrieved October 30, 2020 from https://blogs.nvidia.com/blog/2020/08/25/rtx-real-time-ray-tracing/

Chan, K. L., Ichikawa, K., Watashiba, Y., Putchong, U., and Iida, H. (2017). A Hybrid-Streaming Method for Cloud Gaming: To Improve the Graphics Quality delivered on Highly Accessible Game Contents. International Journal of Serious Games, 4(2). https://doi.org/10.17083/ijsg.v4i2.163

Chen, D., and El-Zarki, M. (2018). Improving the quality of 3d immersive interactive cloud based services over unreliable network. Proceedings of the 10th International Workshop on Immersive Mixed and Virtual Environment Systems. doi:10.1145/3210438.3210440

Chris Wyman. 2019. A Gentle Introduction to DirectX Raytracing. Retrieved October 30, 2020 from http://cwyman.org/code/dxrTutors/dxr_tutors.md.html

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. In Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15). Association for Computing Machinery, New York, NY, USA, 121–135. DOI:https://doi.org/10.1145/2742647.2742657

GameBench Staff (2019). Measuring frame rates in Android and ios games. Retrieved April 06, 2021, from https://blog.gamebench.net/measuring-frame-rates-in-android-and-ios-games

Gyarmathy, K. (n.d.). Reducing latency with edge computing and network optimization. Retrieved April 07, 2021, from https://www.vxchnge.com/blog/how-data-center-reduces-latency

Ikeuchi, Katsushi (2014). "Lambertian Reflectance". Encyclopedia of Computer Vision. Springer. pp. 441–443. doi:10.1007/978-0-387-31439-6_534

Jurgelionis, A., Fechteler, P., Eisert, P., Bellotti, F., David, H., Laulajainen, J. P., Carmichael, R., Poulopoulos, V., Laikari, A., Perälä, P., De Gloria, A., and Bouras, C. (2009). Platform for distributed 3d gaming. International Journal of Computer Games Technology, 2009, 1-15. doi:10.1155/2009/231863

Microsoft Docs. 2018. Running the winsock client and server code sample - win32 apps. Retrieved April 06, 2021, from https://docs.microsoft.com/en-us/windows/win32/winsock/finished-server-and-client-code

Nelius, J. (2019, November 21). Here's how Stadia's input lag compares to native PC gaming. Retrieved April 06, 2021, from https://www.pcgamer.com/heres-how-stadias-input-lag-compares-to-native-pc-gaming/

Oberhumer. (2017). Lzo real-time data compression library. Retrieved April 06, 2021, from http://www.oberhumer.com/opensource/lzo/

Phong, Bui Tuong. (1975). Illumination for computer generated pictures. Commun. ACM 18, 6 (June 1975), 311–317. DOI:https://doi.org/10.1145/360825.360839

Sabino T.L., Andrade P., Gonzales Clua E.W., Montenegro A., Pagliosa P. (2012) A Hybrid GPU Rasterized and Ray Traced Rendering Pipeline for Real Time Rendering of Per Pixel Effects. In: Herrlich M., Malaka R., Masuch M. (eds) Entertainment Computing - ICEC 2012. ICEC 2012. Lecture Notes in Computer Science, vol 7522. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33542-6_25

Schied, C., Salvi, M., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C., Burgess, J., & Liu, S., Dachsbacher, C., and Lefohn, A. (2017). Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. 1-12. 10.1145/3105762.3105770.

Seong-Ping Chuah and Ngai-Man Cheung. (2014). Layered Coding for Mobile Cloud Gaming. In Proceedings of International Workshop on Massively Multiuser Virtual Environments (MMVE '14). Association for Computing Machinery, New York, NY, USA, 1–6. DOI:https://doi.org/10.1145/2577387.2577395