Using State-of-the-art Computer Vision Models to Build Novel Applications on Recent XR Platforms

Candidate Number: 1058971

Word Count: 12701

A thesis submitted for the degree of

MSc in Advanced Computer Science

Trinity 2022

Acknowledgements

I would first like to thank Professor Ivan Martinovic for his supervision and all the convenience and support he provided me to conduct my research. I would also like to thank Doctor Ivo Sluganovic for his incredible guidance on this project. He always had great ideas and gave me valuable insights and helpful resources to finish the project. I am most grateful to my parents who supported my decision to pursue my dream and give me endless love and care. They sacrificed a lot when I was growing up. Without them, I would not be where I am today. Finally, I would like to thank my friends and classmates who helped, supported, inspired, and encouraged me through this wonderful school year.

Abstract

Virtual Reality (VR) and Augmented Reality (AR) technologies have lifted the limitation of distance and opened a new approach for people to perform collaborative tasks by bringing them together into a virtual space. In situations such as construction work and archaeological excavation where the cooperation depends on reality, we need to "teleport" people into a "real" environment instead of the virtual space. It can be done by recording and reconstructing the environment in real time and transmitting it to the collaborators to view in virtual reality. In this thesis, we will build a real-time VR teleportation application with two different computer vision models: 3D point cloud reconstruction and Neural Radiance Fields (NeRF). It will be the first time the 3D point cloud reconstruction method and NeRF model get applied to a real-time VR application. We will further improve upon the state-of-the-art model in the NeRF field by applying application-specific optimization techniques that achieve better training speed and model accuracy. We will also propose a client-server architecture that enables scene reconstruction and rendering to a mobile device with NeRF in real time. Finally, we will demonstrate the reconstruction results and model performance through our experiments and draw conclusions.

Contents

Chapter 1

Introduction

In recent years, Virtual Reality (VR) and Augmented Reality (AR) technologies have developed rapidly thanks to the revolution of mobile hardware. The device has evolved from a bulky head-mounted display powered by a high-performance computer to an all-in-one mobile device such as Meta Quest2 and Microsoft Hololens 2. On the software side, the breakthrough of machine learning and computer vision technologies also unleashed the new potential for VR and AR (XR) applications. Hand tracking and facial recognition can now be achieved with on-device cameras and depth sensors. In this thesis, we will explore using state-of-the-art computer vision models to build a novel VR "teleportation" application between two XR devices. VR "teleportation" means virtually teleporting a client to the same "reality" that the host is currently in. This can be done by recording and reconstructing the environment in real time from the host's XR device and transmitting it to the client's XR device for viewing and becoming immersed. Specifically, we will explore using two different computer vision models for the reconstruction task: 3D reconstruction with point cloud and Neural Radiance Fields (NeRF) [23]. We will use Microsoft Hololens 2 as the teleportation host's device for environment recording and Meta Quest2 as the teleportation client's device for rendering.

1.1 Motivation

XR technologies have lifted the limitation of distance and opened a new approach for people to perform collaborative tasks remotely. The current VR and AR applications mostly bring people together into a virtual space for reality-independent activities such as meetings, online gaming, or virtual collaboration. On the other side, some

activities heavily depend on the environment. For example, archeologists excavating an ancient site or civil engineers working together inside a building under construction. In such situations, We need to teleport people into another "reality" by reconstructing a host environment and sharing it with other collaborators in real time.

Some applications achieve real-time reconstruction by using panoramic images. This approach suffers from view distortions because the view is generated by stitching images together and each image only has one projection center. Others use 3D point clouds to reconstruct a 3D model of the scene. This approach can render natural views from arbitrary viewpoints. However, it renders non-photorealistic views because the reconstructed model and rendering are based on points. Also, this approach has not been applied to a real-time XR application yet. The advent of the Neural Radiance Field (NeRF) has made it possible to render photorealistic views from arbitrary viewpoints by training a deep neural network. But the limitation of such models is that they do not work in real time. Some models have fast training time but are unable to render on a mobile GPU [13, 19, 24, 33, 34]. Other models optimize for mobile rendering but have unacceptable training time [5, 14].

In this thesis, we will develop a real-time VR teleportation application with 3D point cloud reconstruction and NeRF separately. It will be the first time that the 3D point cloud reconstruction model gets deployed to an XR application that runs both reconstruction and rendering in real time. For NeRF, we will apply application-specific optimization techniques to improve both the training speed and accuracy upon the state-of-the-art model [24] in this field. We will also introduce a client-server architecture to enable real-time training and remote rendering to a mobile device for the first time.

1.2 Thesis Objectives

For each approach, We will introduce the methods we use to reconstruct the scene. We will also introduce optimization techniques that make the application run in real time. For the 3D point cloud reconstruction approach, we will

- Introduce a method to color the point clouds using RGB frames
- Develop a method to merge the colored point clouds which generates the final 3D point model of the scene

- Propose a downsampling approach to improve data processing and rendering speed while not impairing visualization quality
- Build a client-server architecture for real-time data processing and rendering

For the NeRF approach, we will

- Introduce a data pre-processing method to improve the quality of training data
- Review the mathematical foundations of NeRF where our improvement method is based upon
- Review a state-of-the-art method that enables faster training and rendering speed of NeRF which is required by our application
- Propose our own improvement method that builds on top of the state-of-the-art approach by using depth information. It further speeds up the training process and improves model accuracy.
- Develop a client-server architecture that enables training on a GPU server and remote rendering to a mobile device in real time.

1.3 Thesis Outline

In Chapter 2, we will introduce the background by reviewing related work in panoramic image, 3D scene reconstruction and NeRF fields. We will also show the VR and AR devices we choose for our application and the reason about our choice. In Chapter 3, we will explain the 3D point cloud reconstruction approach by showing the methods to color and merge the point clouds. In Chapter 4, we will first introduce the data pre-processing step. Then we will review the basic NeRF model and the state-of-the-art method that improves upon it. Next, we will bring up our own application-specific optimization techniques that improve the training speed and accuracy of the state-of-the-art method. Finally, we will demonstrate the client-server architecture that enables real-time training and rendering for our application. In Chapter 5, we will showcase our experiments, followed by the reconstruction results for both approaches and parameter tunings. In Chapter 6, we will present the conclusions and compare the two approaches. In the end, we will discuss possible improvements and future work.

Chapter 2

Background

There are three approaches for scene reconstruction: panoramic image, 3D scene reconstruction and Neural Radiance Fields (NeRF). In this chapter, we will show their advantages and disadvantages and review related work in each field. After that, we will introduce the two XR devices we choose for the teleportation host and client. Then, we will show their hardware specs and introduce the sensors we use for data collection.

2.1 Related Work

2.1.1 Panoramic Image

The basic approach to reconstructing a 3D scene is panorama-based 360° image with 6 degrees of freedom (DOF). To generate such an image, one needs to align and stitch multiple images together. It is done by finding and matching distinctive image features to establish correspondences between image pairs and applying appropriate transformations to blend the images [38]. During the blending process, one needs to fix lens distortion, parallax, moving objects and exposure difference.

The method has already been widely adopted in VR applications. For example, one can view a 360° panoramic image and movie on any standard VR device for an immersive experience. Apple introduced Quicktime VR [4] in 1995 which uses panoramic images instead of 3D computer graphics to model and render virtual environments. It lets users navigate inside a VR environment by hopping to different panoramic points. A more recent version of such application is Google and Apple Maps' street view features.

While this method has the advantage of speed and simplicity, there exist disadvantages. Since the panoramic image is generated by stitching multiple images together and each image has only one projection center, viewing a novel scene from arbitrary camera angles would result in image reprojection and unnatural views. Also, fixing moving objects, parallax and exposure discrepancy may be challenging for a scene with large differences. [11, 40].

2.1.2 3D Scene Reconstruction

The 3D scene reconstruction approach estimates a most likely 3D object model from a collection of images with known camera poses. The advantage of this approach over panoramic image is that it enables generating natural novel views from any camera angle because of the 3D model.

In general, the class of methods that perform 3D reconstruction without depth knowledge is called Multi-View Stereo (MVS). They infer depth information from 2D images and convert it to geometry information. Such geometry information can be represented by voxels, level-sets, polygon meshes or depth maps. The methods usually need to run a process that finds correlated features between views based on some visibility models or criteria. It then runs a reconstruction algorithm based on the correlated features and their respective camera poses to reconstruct a 3D model. The algorithm may use a good initialization or prior knowledge of geometry to guide the reconstruction process. The reader can refer to the MVS comparison and evaluation paper [37] for a detailed explanation of the method and a comprehensive list of related literature in this area.

The advent of RGB-D cameras makes 3D scene reconstruction with depth information possible. It brings better accuracy and performance improvement over the traditional MVS method. Curless et al. [6] proposed a volumetric method that lays the foundation for many of the modern approaches. It represents each range image by a set of signed distance functions (SDF) and uses discrete voxel grids to store them. The SDFs are then combined to reconstruct the final model. With this model, surface reconstruction can be easily done by taking the points with zero distance on the SDF. The SDF and voxel-based representation idea was later adopted by KinectFusion [26] for a real-time 3d reconstruction system. Another class of real-time reconstruction systems is based on points [16, 35]. The point-based system has the advantage of speed and memory efficiency by directly working with points from the depth sensor

instead of converting them into the voxel representation. We will adopt the point-based approach in our 3D reconstruction method for its simplicity and efficiency. A better review of the classic 3D scene reconstruction with RGB-D data methods can be found in [48].

More recent work trains deep neural networks to represent 3D shapes and perform scene reconstruction. Given limited input such as partial point cloud and RGB images, the network is able to learn an SDF [29, 25] or occupancy field [22, 15] representation of the scene. The other neural network methods do not learn a consistent 3d shape; instead, they directly generate novel views based on the input images. It is called view synthesis and image-based rendering. Example work can be found in [12, 23, 31, 47]. Among those, Neural Radiance Fields (NeRF) [23] is the state-of-the-art approach.

2.1.3 Neural Radiance Fields (NeRF)

NeRF uses a partial set of 2D posed images to train a fully-connected Multi-Layer Perceptron (MLP) network that learns the correspondence between views of a 3D model and camera poses. Once trained, the neural network can generate novel views for any camera poses. Thus, the train MLP represents a 3D model implicitly. The advantage of NeRF over the traditional 3D scene reconstruction methods is that it can generate photorealistic views based on just 2D image input. Also, compared to the polygon meshes and voxels used by the classic methods to store the 3D model, NeRF requires less storage space.

Since the publishing of the original NeRF paper, there have been subsequent works that improve upon the method. One field is to improve the training and rendering speed. The original method takes 1-2 days to train a good quality model on a single NVIDIA V100 GPU and about 30 seconds to render an 800×800 image [23]. The first cause is that it trains a single large fully-connected MLP, and each backward propagation step updates every parameter of it. The second cause is that the volumetric rendering approach adopted by this paper requires querying the MLP hundreds of times for each input pixel. To speed up the training and evaluation process, Nvidia introduced a new input encoding mechanism in its Instant Neural Graphics Primitives (Instant-NGP) paper [24] that allows the use of a much smaller MLP augmented by multiresolution hash tables. It reduces the training time from days to minutes. Another class of methods improve the performance by factoring the MLP into sub-networks [13, 33, 34] or using a 3D voxel grid system [13, 14, 19, 34]

to store spacial information. For example, KiloNeRF [34] factorizes the MLP network into thousands of tiny networks. Each network represents part of the scene that falls into a cell of a 3D grid so that they can be updated separately. It also utilizes empty space skipping (ESS) and early ray termination (ERT) techniques to speed up the point sampling and pixel rendering process. FastNeRF [13] factorizes the MLP network into two smaller networks: one depends only on the point 3d coordinates and the other depends only on the viewing direction. It then uses caching to trade memory for performance. SNeRG [14] introduces a sparse grid representation and precomputes and stores the trained NeRF into the grid to speed up rendering.

Another field to improve the model is by utilizing more information. For example, DS-Nerf [7] uses depth information to supervise the basic NeRF model that leads to 2-3x speed improvement with fewer training views. PixelNerf [44] uses a fully-convolutional image encoder to learn image features before training and feed NeRF with both spacial coordinates and corresponding feature vectors. Taking the idea from learning-based MVS, MVSNeRF [3] first constructs a cost volume by warping 2D image features onto a plane sweep and then converts it to a neural encoding volume using 3D Convolutional Neural Network (CNN). The output of neural encoding volume instead of the 3D position will be fed into the MLP to generate a view. Both PixelNerf and MVSNeRF can take as few as three input images for training and still yield decent results.

For good user experiences, our application needs to train the model and render views in real time. Therefore, we will adopt Nvidia's Instant-NGP NeRF implementation and improve upon it. It enables us to train a model in a few minutes and render each frame in milliseconds magnitude. Since the XR device we use has a depth sensor, we will borrow the idea of depth supervision from DS-Nerf [7] but implement our own cost function to further improve the Instant-NGP framework.

2.1.4 Industrial Efforts on VR Teleportation

There is no dedicated literature work on VR teleportation yet. However, some industrial companies have made efforts to develop applications in this area. Varjo released a real-time reality teleportation service on their XR-3 headset using low-latency video pass-through technology and on-device lidar hardware [17]. Google and Microsoft both introduced novel video conference systems that "teleport" attendants into a conference room through VR and make them feel as if they are in the same physical space [18, 46]. Microsoft also presented DreamWalker [43], which "teleports" users to

a virtual world with the same walkable paths and obstacles in real time while walking outdoors.

2.2 Devices

2.2.1 Microsoft Hololens 2

In our research, we use Microsoft Hololens 2 as the teleportation host's AR device for environment recording and data collection. We choose an AR device because it can let the host see the environment and virtual recording information at the same time during the scanning process. Also, Hololens 2 is equipped with an RGB camera, a depth camera and a built-in inertial measurement unit (IMU) that provides high-quality data for our 3D point cloud reconstruction and NeRF training tasks. Figure 2.1 shows the available sensors on the Hololens 2 headset.

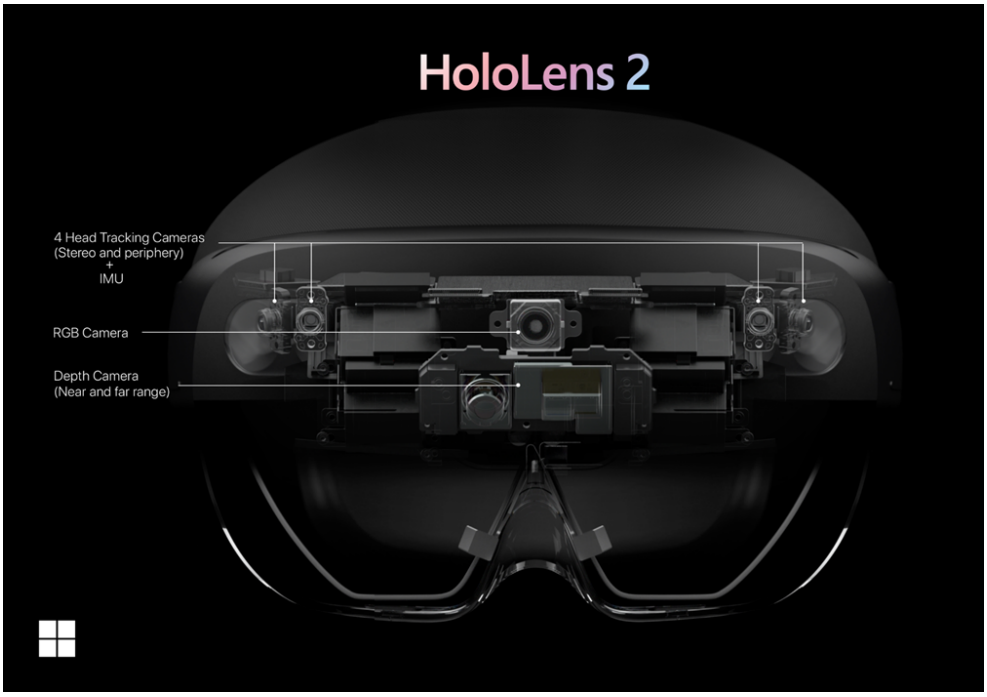


Figure 2.1: Hololens 2 sensors

We enable the research mode [39] on Hololens 2 which provides us access to the raw data from the sensors. The RGB camera generates 8-megapixel RGB images at the frame rate of 30 frames per second (fps). The depth camera can operate in two modes, near-depth sensing with high frame rate (45 fps) and far-depth sensing with low frame rate (5 fps). Both modes generate depth frames with uncolored point clouds. The near-depth sensing mode is normally used for hand tracking and the far-depth sensing

mode is used for spacial mapping. Since we need to capture the environment with objects in distance, we will enable the second mode. the IMU data comes in the form of 4×4 3D spacial transformation matrices at the rate of 60 Hz.

2.2.2 Meta Quest 2

The teleportation client will need to use a VR device to visualize and become immersed in the reconstructed environment. Therefore, we choose Meta Quest 2 (previously known as Oculus Quest2) as the client's rendering device. It is a popular VR headset on the market equipped with a Qualcomm Snapdragon XR2 SOC, 6 GB memory and a 1.2 TFLOPS Adreno 650 integrated GPU. the display resolution is 1832 x 1920 per eye and it supports up to 90 Hz refresh rate. The device also has a built-in head-mounted display IMU for head position tracking and four cameras for hand tracking and video passthrough. Since we need to render different views of the reconstructed model based on different head poses, the IMU data is essential for our application. See Figure 2.2 for the device and its hardware specs.

	
SoC	Qualcomm Snapdragon XR2
CPU	Qualcomm Kryo 585, 8-Core
GPU	Qualcomm Adreno 650 (~1.2 TFLOPS)
Memory	6 GB LPDDR4X
Display Resolution	1832 x 1920 Per Eye
Refresh Rate	Up to 90 Hz
Sensors	1 HMD IMU, 4 Infrared Cameras

Figure 2.2: Meta Quest 2 hardware specs

Chapter 3

3D Point Cloud Reconstruction

The first method we implement is 3D point cloud reconstruction. This method records and reconstructs the environment in real time using 3D points. It has the advantage of fast reconstruction and rendering speed. But the rendered views are not as good as NeRF. In this chapter, we first explain how to color individual point clouds by combining the RGB and depth frames. Then, we introduce the process of merging colored point clouds to create the final environment model. We highlight two techniques used during the merging process to help improve the reconstruction speed and model quality: downsampling and point cloud alignment. Finally, we demonstrate a client-server architecture that enables simultaneous data collection and reconstruction.

3.1 Combine RGB and Depth Frames

The depth camera on Hololens 2 generates a point cloud in each depth frame and runs at the frame rate of 5 fps. The RGB camera run at 30 fps. Since the depth camera operates at a lower frequency, for each depth frame, we find the corresponding RGB frame with the closest timestamp and discard the rest to align the frame timestamps. The timestamp difference is at most $1/60 \approx 0.017$ seconds, which gives enough alignment accuracy.

To generate a colored point cloud from the RGB and depth frame pair, we project the 3D points onto the image plane and color them by the corresponding pixel values. In each depth frame, the point's coordinate $p = (x, y, z)^T$ is in the depth camera coordinate system. Together with the frame, Hololens provides a 4×4 homogeneous transformation matrix $T_{world\_to\_depth}$ which is the transformation from the world origin to the depth camera origin. For each RGB frame, Hololenss also provides a 4×4

homogeneous transformation matrix $T_{world\_to\_rgb}$ which is the transformation from the world origin to the RGB camera origin. From above, we can get the transformation matrix from the RGB camera origin to the depth camera origin by multiplying the two matrices

$$T_{rgb\_to\_depth} = {T_{world\_to\_rgb}}^{-1} \cdot T_{world\_to\_depth} \tag{3.1}$$

Frist, we transform the point coordinates from the depth camera coordinate system to the RGB camera coordinate system using $T_{rgb\_to\_depth}$. Let point P_i's homogeneous coordinate be

$$\hat{p}_i = \begin{bmatrix} p_i \\ 1 \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ z_i \\ 1 \end{bmatrix} \tag{3.2}$$

By applying $T_{rgb\_to\_depth}$, we get the point coordinate in the RGB camera coordinate system p'_i

$$\hat{p'_i} = T_{rgb\_to\_depth} \cdot \hat{p}_i = \begin{bmatrix} p'_i \\ 1 \end{bmatrix}, \text{ where } p'_i = \begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} \tag{3.3}$$

Next, we compute the pixel coordinate of each point. Since Hololens 2 uses a pinhole camera model where the RGB camera origin is the projection center of the camera. We can project the point onto the image plane by multiplying it with the camera intrinsic matrix K.

$$K \cdot p'_i = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} = \begin{bmatrix} f_x x'_i + c_x z'_i \\ f_y y'_i + c_y z'_i \\ z'_i \end{bmatrix} \tag{3.4}$$

Where f_x, f_y are the focal length and c_x, c_y are offsets of the principal point from the top-left corner of the image frame. The values are all expressed in pixels. Let

$$p''_i = \begin{bmatrix} x''_i \\ y''_i \\ z''_i \end{bmatrix} = \begin{bmatrix} f_x x'_i + c_x z'_i \\ f_y y'_i + c_y z'_i \\ z'_i \end{bmatrix} \tag{3.5}$$

Then we can get the pixel coordinate for point P_i

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} \frac{x''_i}{z''_i} \\ \frac{y''_i}{z''_i} \end{bmatrix} = \begin{bmatrix} \frac{f_x}{z'_i} x'_i + c_x \\ \frac{f_y}{z'_i} y'_i + c_y \end{bmatrix} \tag{3.6}$$

Where u_i, v_i is the number of pixels from the top left corner of the RGB image frame along the x and y axis. In practice, we will perform the above calculations on the points matrix instead of individual points for speed. We can use the following OpenCV [2] function to perform the calculations in the matrix form.

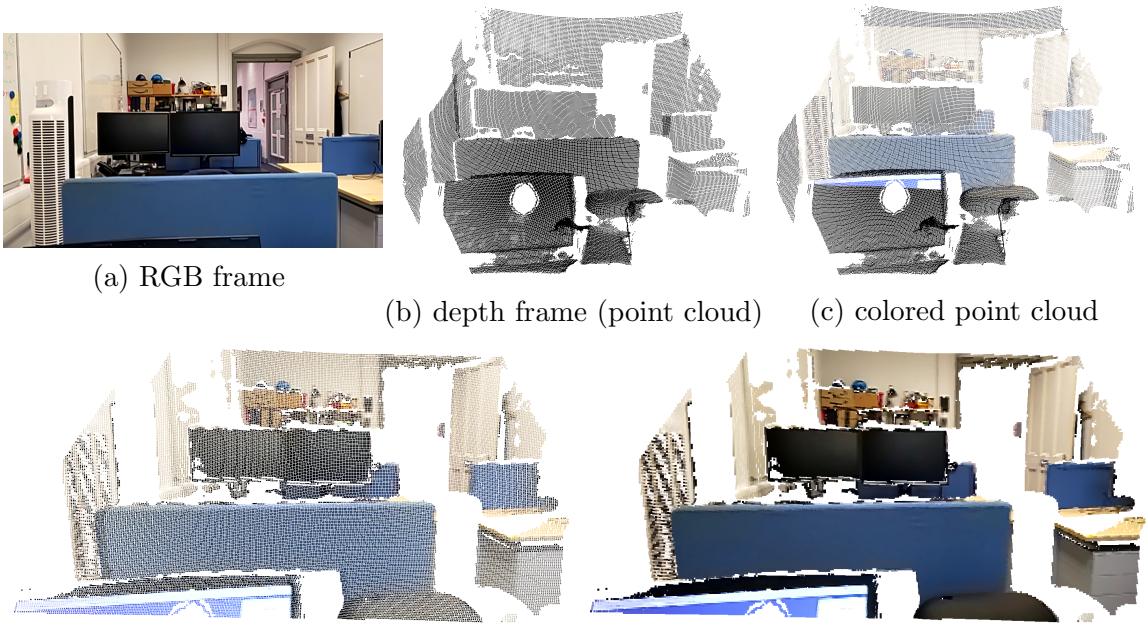
```
cv2.projectPoints(image, points, rvec, tvec, K, None)
```

The input `points` is a matrix of the form

$$M = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix} \tag{3.7}$$

where each row (x_i, y_i, z_i) is the coordinate of point P_i in the depth camera coordinate system, input `rvec` and `tvec` are the rotation and translation vectors of the $T_{rgb\_to\_depth}$ matrix and `K` is the camera intrinsics matrix. the result is a matrix of the form $\begin{bmatrix} u_1 & v_1 \\ \vdots & \vdots \\ u_n & v_n \end{bmatrix}$ where (u_i, v_i) is the pixel coordinate of point P_i.

Finally, we clamp the point cloud by filtering out points that fall outside of the image frame. We select points with $u_i < w$ and $v_i < h$ where w, h are the width and height of the image frame and assign the pixel RGB color to the point. Figure 3.1 demonstrates the process of coloring the point cloud.



(a) RGB frame

(b) depth frame (point cloud)

(c) colored point cloud

(d) clamped colored point cloud

(e) adjusted point size for better visulization

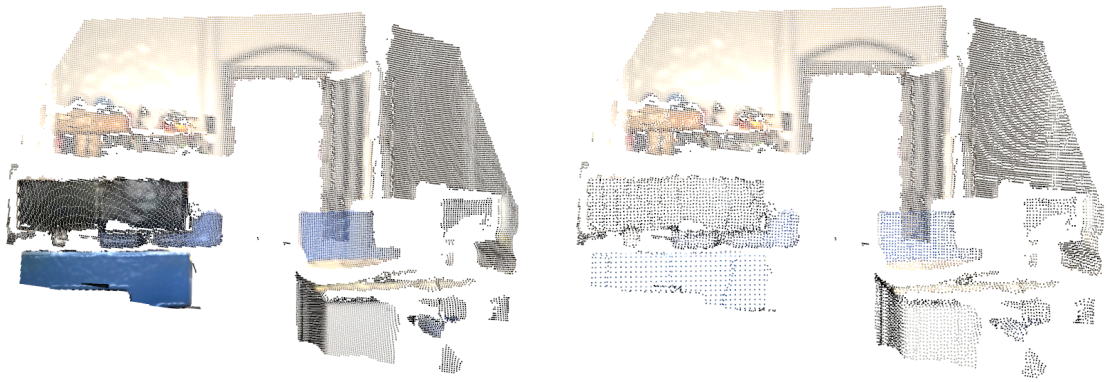
Figure 3.1: The process of coloring a point cloud

3.2 Merge Colored Point Clouds

After coloring the point cloud for each depth frame, we will merge them together to form a larger and denser point cloud. We will introduce the downsampling and point cloud alignment techniques which improve the performance and quality of the final merged point cloud.

3.2.1 Downsampling

Since the depth sensor generates a relatively dense point cloud in each depth frame, and consecutive frames largely overlap with each other, we can downsample the individual point clouds before merging them to improve the merging speed while still keeping a good visualization quality. We take a voxel-based downsampling approach by creating a voxel grid of certain resolution over the 3D space; then, we only keep the point that is closest to the centroid inside each voxel cube. The bigger the voxel size, the lower the resolution and thus the fewer points we keep. This method has the advantage of creating a uniformly distributed point cloud that is good for visualization. Figure 3.2 demonstrates the comparison between a full-resolution and a downsampled point cloud. We can see that near-range and far-away objects have the same point resolutions after downsampling.



(a) full resolution point cloud

(b) downsampled point cloud

Figure 3.2: A full resolution point cloud and a downsampled point cloud with 2cm voxel size of the same view

3.2.2 Point Clouds Alignment

Individual point clouds are in the depth camera coordinate system. Since we have the world to depth camera transformation $T_{world\_to\_depth}$ for each depth frame, we can

simply transform the points into the world coordinate system and combine them to generate the merged point cloud. However, in practice, $T_{world\_to\_depth}$ may not be accurate. This transformation is calculated by aggregating IMU's high-frequency acceleration and angular rate data that can be noisy from time to time. On the other hand, the merging process requires very high accuracy because a small error in the transformation may lead to deviation of all points in the point cloud and result in duplicate objects. Therefore, we align the point clouds using the iterative closest point (ICP) algorithm [1] before merging them. The algorithm takes a source point cloud P, a target point cloud Q and an initial transformation guess $T_{initial}$ and manages to find a final transformation T_{final} from P to Q that aligns them. It calculates T_{final} by iteratively minimizing an objective function. In our application, we will use the $T_{world\_to\_depth}$ transformation from Hololens 2 as the initial guess. Since the algorithm works best when the two point clouds largely overlap, we will perform ICP on the consecutive point cloud pairs along the timeline instead of any pair for faster speed.

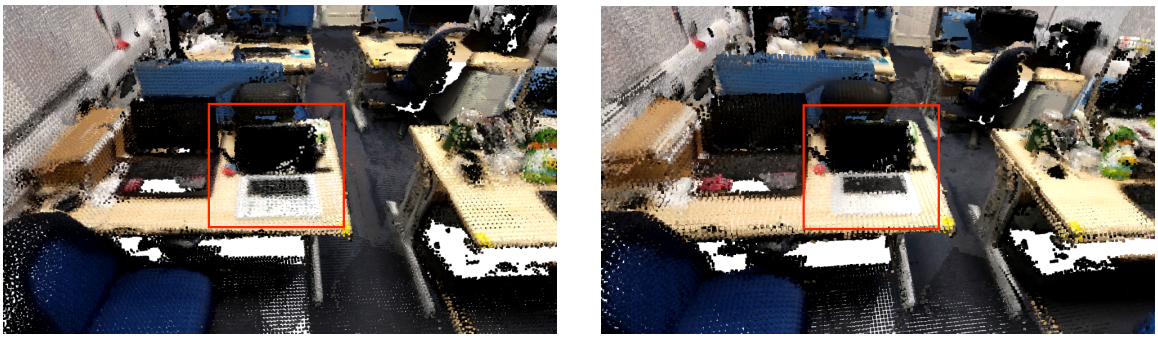
There exist modified ICP algorithms that use the color information to better match the points [21, 28]. However, since our case is fairly simple and the initial alignment is already good enough, we will just perform the simple point-to-point version of the ICP algorithm `p2p_icp(P,Q,T_initial)` that uses only the point XYZ information. In each iteration, the algorithm runs the following steps: given the current transformation matrix T_i, it first constructs the set $S_i = \{(p_i, q_i) | p_i \in P, q_i \in Q\}$ where for each point $p_i \in P$, it transform it to p_i' by applying T_i, and find its closest point $q_i \in Q$. Then it updates T_i by minimizing the sum of the squared differences between p_i' and q_i

$$E(T_i) = \sum_{(p_i, q_i) \in S_i} \|T_i \cdot p_i - q_i\|^2 \tag{3.8}$$

The algorithm repeats step 3.8 until it converges. We run the `p2p_icp` algorithm on each consecutive depth frame pair and use the aligned source point cloud from the previous pair as the target for the next pairs.

3.2.3 Alignment Quality

We use the "fitness" score to measure how good the point cloud alignment is. It is the mean squared distance from each point p_i in the source point cloud to its closest point q_i in the target point cloud. Figure 3.3 shows a comparison between the aligned and unaligned point clouds. The aligned point cloud has a lower fitness score.



(a) unaligned point cloud, fitness: 0.00343 (b) aligned point cloud, fitness: 0.00232

Figure 3.3: **Point cloud before and after the alignment** Note that the white points in front of the laptop screen are aligned into the laptop body by ICP

3.2.4 Final Algorithm

After the alignment, we generate the merged point cloud by combining all the points in the individual point clouds together. We then perform another downsampling process on the merged point cloud with the same voxel size to keep the points uniformly distributed. Algorithm 1 gives the final algorithm for merging the point clouds. Note that we perform downsampling (line 7) before the alignment (line 8) to reduce the number of points and speed up the ICP algorithm.

Algorithm 1 Merge Point Clouds

Input: *pclouds*: list of point clouds in the depth camera coordinate system
tfs: list of world to depth camera transformations for each point cloud
Output: merged point cloud in the world coordinate system

1: **procedure** MergePointClouds(*pclouds*, tfs)
2: $result \leftarrow \emptyset$
3: $target \leftarrow tfs[0] \cdot \texttt{downsample}(pclouds[0])$
4: $result.\texttt{addPoints}(target)$
5: **for** $i \leftarrow 1$ to n **do**
6: $T_{initial} \leftarrow tfs[i]$
7: $source \leftarrow \texttt{downsample}(pclouds[i])$
8: $T_{final} = \texttt{p2p\_icp}(source, target, T_{initial})$
9: $target \leftarrow T_{final} \cdot source$
10: $result.\texttt{addPoints}(target)$
11: **end for**
12: $result \leftarrow \texttt{downsample}(result)$
13: **return** *result*
14: **end procedure**

3.3 Client-Server Architecture

After downsampling, the teleportation client's Meta Quest 2 headset will be capable of rendering the merged point cloud in real time. The data collection, point cloud coloring and merging process can all be done on the teleportation host's Hololens 2. However, since the data processing tasks can be quite computationally heavy, we will employ a client-server architecture and offload the point cloud coloring and merging steps to a more powerful server for faster data processing speed. The other benefit of this client-server architecture is that it lets data collection and processing happen at the same time. While the Hololens 2 collects data, it also streams the recorded RGB, depth and transformation frames to the server. When new frames arrive, the server can color the point clouds, downsample, and aggregate them into the existing merged point cloud on the fly. Once the host finishes recording the environment, it sends a confirmation signal to the server, and the server performs a final downsampling on the merged point cloud. After that, it packs the merged point cloud as the final 3D model of the scene and sends it over to the client's Meta Quest 2 for rendering. Figure 3.4 demonstrates the software architecture and the whole workflow.

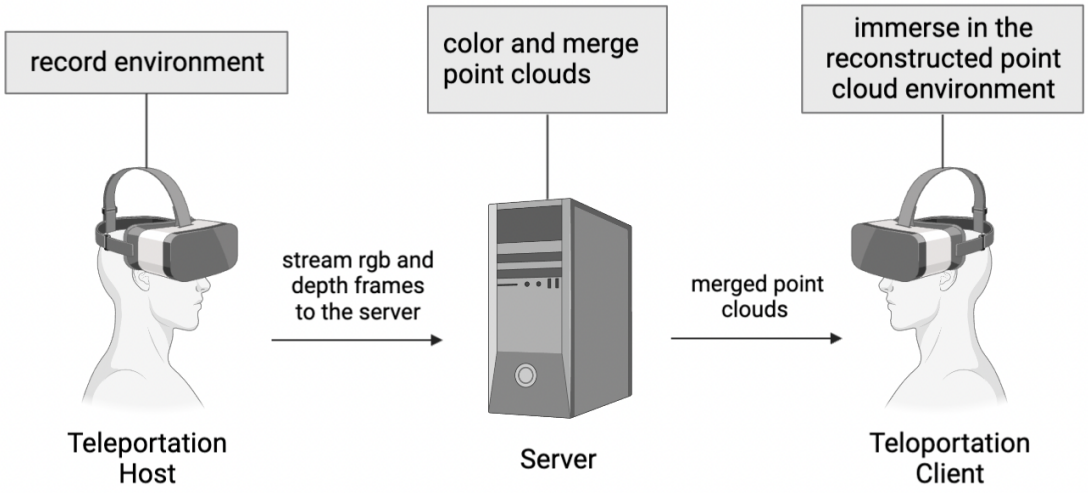


Figure 3.4: Software architecture and workflow of 3D point cloud reconstruction

Chapter 4

Neural Radiance Fields (NeRF)

The second method we explore is NeRF. We will train a NeRF model to implicitly represent a 3D reconstructed scene and use it to render novel views in VR given head poses in space. This method achieves the best rendering quality but has a limitation of slow training and rendering speed as compared to the 3D point cloud reconstruction approach. In this chapter, we first introduce how to pre-process the data by sub-sampling the image frames with the best sharpness. We will also mention the ground truth camera poses and intrinsics that we acquire from the device to train the model. Then, we give a detailed review of the NeRF [23] model and how to speed up with Nvidia's Instant Neural Graphics Primitives (Instant-NGP) [24]. Next, we propose our own improvement on top of Instant-NGP by using depth data from the XR device. Finally, we introduce a client-server architecture that runs training and inference on the GPU server and remotely renders novel views to the client in real time.

4.1 Data Pre-processing

Same as the 3D point cloud reconstruction method, we record RGB and depth frames on the Hololens 2 and stream them to the GPU server along with the corresponding camera poses for processing. NeRF works well when the images are non-blurry, so we perform a data sub-sampling process by picking out the clearest image frames in every recording window and then feed them to the NeRF model.

4.1.1 Sub-sampling with Best Sharpness

Based on the sample size, we divide the whole recording into (*total / sample size*) chunks of frames along the timeline and only keep one frame with the best sharpness in each chunk. This sub-sampling process keeps frames across the recording period so that we can still cover the whole recording area. We use variance of the laplacian [30] to measure the sharpness of an image. It is done by convolving over the image with the laplacian operator, which can be expressed as a mask

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Then, we calculate the variance of the convolved values. For an image of size $m \times n$, the sharpness is

$$sharpness = \sum_{i}^{m-2} \sum_{j}^{n-2} (L(i,j) - \overline{L})^2 \tag{4.1}$$

where $\overline{L}$ is the average of convolved values

$$\overline{L} = \frac{1}{mn} \sum_{i}^{m-2} \sum_{j}^{n-2} L(i,j) \tag{4.2}$$

We can use the following OpenCV [2] code to calculate the value for a given image

```
cv2.Laplacian(image, cv2.CV_64F).var()
```

4.1.2 Camera Poses

NeRF trains with both RGB images and their camera poses. It also requires the camera's intrinsic parameters for image projection. The original NeRF paper does not require ground truth camera poses; instead, it uses the structure-from-motion (SfM) package COLMAP [36] to estimate relative camera poses from the 2D images. In our case, since the Hololens 2 has a built-in IMU, it will provide us with the ground truth camera pose for each RGB frame. The poses come in as a 4×4 homogeneous transformation matrix in the world coordinate system where the origin is the recording start point. Hololens 2 also provides the camera's intrinsic parameters from the factory information. Therefore, we can directly feed the ground truth camera poses and intrinsic parameters into the NeRF model for training. This lets us skip the process of running SfM and also helps improve the accuracy of the NeRF model.

4.2 NeRF Review

In this section, we will give a detailed review of the original NeRF [23] paper on how it calculates the expected color of a pixel and the two optimization techniques it introduces: positional encoding and hierarchical volume sampling. In the next section, we will introduce how Nvidia's Instant Neural Graphics Primitives [24] uses a new input encoding mechanism to improve the performance of the original NeRF model significantly.

4.2.1 Volume Rendering

As opposed to an explicit 3D model represented by meshes and voxels, NeRF uses the radiance field to implicitly model a scene and then employs the classical volume rendering approach to render the color of each ray traced through the pixels. The radiance field is defined by the following function

$$g : (\mathbf{x}, \mathbf{d}) \rightarrow (\sigma, \mathbf{c}) \tag{4.3}$$

which maps a 3D coordinates $\mathbf{x} \in \mathbb{R}^3$ and 2D viewing direction $\mathbf{d} \in \mathbb{R}^2$ to a volume density $\sigma \in \mathbb{R}^+$ and RGB color $c \in \mathbb{R}^3$. It models the function g by training a basic Multi-Layer Perceptron (MLP) network F_Θ that optimizes the weights Θ over the cost function of total squared error between the rendered and true pixel colors.

$$L = \sum_{\mathbf{r} \in R} (C(\mathbf{r}) - \hat{C}(\mathbf{r}))^2 \tag{4.4}$$

Here, R is the set of rays in each optimization batch, $C(\mathbf{r})$ is the ground truth color of ray $\mathbf{r}$ and $\hat{C}(\mathbf{r})$ is the predicted color of ray $\mathbf{r}$. The original paper did not provide details on how it derived the formula to calculate the expected color $\hat{C}(\mathbf{r})$ of a camera ray. We will provide the derivation here since the intermediate formulae are used by our improvement method in section 4.4. The first part of derivation largely follows the formulae given by N. Max in his volume rendering's optical models paper [20].

In the NeRF paper, a particle system is used as the model to calculate the color and density of a specific spacial position. In the particle system, assume each particle is a sphere with radius r, then the particle's cross-sectional area is $A = \pi r^2$. Consider a cylinder with base area E and height Δs, and a ray of light passes through it at distance s. let $\rho(s)$ be the cylinder's particle density (the number of particles per unit volume) at distance s. Then, there are a total of $E\Delta s\rho(s)$ particles in the cylinder. If $\Delta s \rightarrow 0$, then we can assume the particles are arranged side by side there

is no overlapping area between them. Therefore, the total cross-sectional area of the cylinder they occupy is $E\Delta s\rho(s)A$. When the light passes through the cylinder, the fraction of light that gets blocked is

$$\frac{E\Delta s\rho(s)A}{E} = \Delta s\rho(s)A \tag{4.5}$$

Let the light intensity at distance s be $I(s)$, then after the cylinder, it becomes

$$I(s+\Delta s) = (1-\Delta s\rho(s)A)I(s) \tag{4.6}$$

Which implies

$$\frac{dI}{ds} = \frac{I(s+\Delta s)-I(s)}{\Delta s} = -\rho(s)AI(s) \tag{4.7}$$

Define the volume density to be

$$\sigma(s) = \rho(s)A \tag{4.8}$$

This is one of the values that NeRF outputs (4.3). Obviously, the solution to Equation 4.7 is

$$I(s) = I(0)\exp\left(-\int_0^s \sigma(t)dt\right) \tag{4.9}$$

Define

$$T(s) = \exp\left(-\int_0^s \sigma(t)dt\right) \tag{4.10}$$

and

$$S(s) = 1 - T(s) \tag{4.11}$$

Then, the function $T(s)$ denotes the accumulated light transmittance along the ray until distance s, and $S(s)$ is the probability of the light hitting some particle before distance s, which is a cumulative distribution function. Denote the color of particle at distance s to be $c(s)$. Then the expected color of the light ray is

$$E(c) = \int_0^\infty S'(s)c(s)ds = \int_0^\infty T(s)\sigma(s)c(s)ds \tag{4.12}$$

In practice, we only calculate the integral from range s_n to s_f. Let us parameterize s with t and define

$$s = \mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{4.13}$$

where $\mathbf{o}$ is the ray origin and $\mathbf{d}$ is the direction. If we limit the range and substitute 4.13 into 4.12, We get equation 1 in the original NeRF paper [23]

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))c(\mathbf{r}(t),\mathbf{d})dt, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t}\sigma(\mathbf{r}(s))ds\right) \tag{4.14}$$

To estimate the integral, the original paper converts 4.14 into a discrete form using a stratified sampling approach. It divides the integral interval $[t_n, t_f]$ into N sub-intervals and then uniformly draw a random sample t_i from each sub-interval. Let $\delta_i = t_{i+1} - t_i$, the discretized form of $T(t_i)$ is defined as

$$T(t_i) = \exp\left(-\sum_{j=1}^{i-1} \sigma(t_j)\delta_j\right) \tag{4.15}$$

and if we replace $S'(s)c(s)ds$ in Equation 4.12 with the discretized form, we get the following discretized color expectation

$$\begin{aligned}\hat{C}(\mathbf{r}) &= \sum_{i=1}^{N}((1 - T(t_{i+1})) - (1 - T(t_i)))c(t_i) \\ &= \sum_{i=1}^{N}(T(t_i) - T(t_{i+1}))c(t_i) \\ &= \sum_{i=1}^{N} T(t_i)(1 - \exp(-\sigma(t_i)\delta_i))c(t_i)\end{aligned} \tag{4.16}$$

which is the same as formula 3 in the original NeRF paper [23]. When rendering, we sample the set of $(\sigma(t_i), c(t_i))$ from our MLP model and calculate the expected pixel color using Equation 4.16.

4.2.2 Positional Encoding

The paper further introduces a positional encoding technique to increase the feature dimension by borrowing the idea from [41]. For a scalar value $x \in \mathbb{R}$, it encodes it into a feature vector $\in \mathbb{R}^{2L}$

$$encode(x) = [\sin(2^0\pi x), \cdots, \sin(2^{L-1}\pi x), \cos(2^0\pi x), \cdots, \cos(2^{L-1}\pi x)] \tag{4.17}$$

Where L is a tunable parameter. Since a deep neural network tends to learn a lower frequency function [32], it may not model an image well and can lead to underfitting. The reason is that clear images usually have many sharp edges that are variations of color and geometry. Increasing the feature dimension can accommodate for such high-frequency variations. The encoding is applied to the values of the 3D coordinates $\mathbf{x}$ and 2D viewing direction vector $\mathbf{d}$ separately. Although this encoding scheme significantly improves the model expressiveness, the drawback is that it requires a large neural network and thus, slower training and rendering speed. Such limitation makes the basic NeRF model incapable of our application. In section 4.3, we will introduce how to speed up with a new encoding mechanism.

4.2.3 Hierarchical Volume Sampling

The paper introduces another optimization technique, hierarchical volume sampling, to improve rendering performance. It uses a more efficient sampling strategy so that the expected pixel color in Equation 4.16 can be better computed. The basic idea is to sample more on the actual object instead of free and occluded space which does not contribute to the final rendering. This technique requires to train both a coarse and a fine MLP network. When rendering, it first uniformly draws samples along the ray and uses the coarse network to predict their volume densities σ. Then, it estimates a distribution of the relevant part based on the volume density and performs another informed sampling. Finally, it uses the fine network to compute the expected color using points from both the first and second sampling processes.

4.3 Speed Up

Although the original paper introduces two optimization techniques to improve the model quality, they do not contribute to any speed improvement. The training time can take up to two days which obviously does not work with our application. To speed up the training process, We will adopt the method proposed in Nvidia's Instant Neural Graphics Primitives paper (Instant-NGP) [24]. It helps reduce the training time from days to minutes on a single GPU. We now briefly talk about how the speed-up is achieved.

As mentioned in subsection 4.2.2, The reason why the original NeRF implementation is slow is because of the fixed positional encoding (4.17). It requires a big MLP network with many trainable parameters. To increase speed, Nvidia's researchers manage to reduce the MLP size significantly by introducing a multiresolution hash table of learnable feature vectors. Then, the training process can optimize the small MLP plus a fraction of the feature vectors using stochastic gradient descent (SGD).

The multiresolution hash table is structured in the following way: First, it creates L levels, and each level contains a 2D grid of different resolutions over the input image. Let the finest resolution be N_{min} and the coarsest resolution be N_{max}. Then for level l, the resolution can be calculated by

$$N_l = \lfloor N_{min} \cdot b^l \rfloor \tag{4.18}$$

$$\text{where } b = \exp\left(\frac{\ln N_{max} - \ln N_{min}}{L-1}\right) \tag{4.19}$$

and the number of grid vertices is

$$V_l = (N_l + 1)^2 \tag{4.20}$$

Next, it associates T F-dimensional feature vectors with the vertices on each level with a hash table. The hash table maps a vertex of the grid into a feature vector. Since the number of feature vectors is fixed to the number T, for a coarse resolution level with the number of vertices $V_l \leq T$, the mapping is one-to-one and the unmapped feature vectors are discarded. For a fine resolution level where $V_l \geq T$, the mapping is many-to-one, which means there will be hash collisions. The paper simply proposes to do nothing with the collision and let the neural network disambiguate them. Here, $L, T, F, N_{min}, N_{max}$ are all tunable parameters.

To encode a pixel in the image, it first finds the grid containing it and the four feature vectors hashed from the grid vertices on each level. Then it performs a linear interpolation according to the relative position of the pixel in the grid to get a final feature vector. Next, the L interpolated feature vectors from each level are concatenated as the final encoding of the pixel. Finally, the encoding plus any auxiliary inputs (such as the view direction) are fed into a small MLP for training.

Although the multiresolution hash table plus the MLP network has a larger memory footprint than the original NeRF model, the training speed is significantly improved because each backward propagation step only needs to update the parameters in the small MLP plus the corresponding feature vectors. According to the Instant-NGP paper, the number of trainable parameters of the MLP is reduced from 438k to 10k for equivalent results, and training time is reduced from around 13 minutes to 2 minutes. Note that although there are 494k encoding parameters, only a tiny portion is updated in each backpropagation step.

The speed improvement by Instant-NGP over the original NeRF model makes it possible to train NeRF within a reasonable amount of time and thus fit into our application. Although Instant-NGP requires significant memory usage, it does not become an issue since the training process happens on a GPU server instead of the XR device.

4.4 Depth Supervision

The original NeRF model does not require any geometry knowledge of the scene it learns. Instead, it implicitly infers 3D structures by calculating the expected termi-

nation position of a ray. In our case, since Hololens 2 has a depth sensor, we can utilize the depth data it provides to further supervise the training. Related work has been done in DS-NeRF [7]. The original NeRF paper uses the SfM packages COLMAP [36] to estimate relative camera poses of the views. During the estimation process, COLMAP also projects certain pixels in the 2D images to 3D points in space. The position of the 3D points can give explicit information about where the ray terminates. However, the mapped point position may be noisy and subject to a probability distribution. To supervise training with depth information, DS-NeRF adds a loss of KL divergence between the rendered ray termination distribution and the point position distribution to the cost function defined in Equation 4.4.

Since we have the ground truth depth information from the depth sensor, we do not need to compute the difference between two probability distributions. Instead, we can just compute the difference between the rendered expected ray depth and the true depth of the pixel and add the squared error to the cost function as the depth loss term. Recall Equation 4.16, during the stratified sampling process, if we let

$$p(t_i) = T(t_i)(1 - \exp(-\sigma(t_i)\delta_i)) \tag{4.21}$$

We can rewrite the equation as

$$\hat{E}(c) = \sum_{i=1}^{N} p(t_i)c(t_i) \tag{4.22}$$

Therefore, we can view $p(t_i)$ as the likelihood of the ray terminating at position t_i. By multiplying it with $s_i = \mathbf{r}(t) = \mathbf{o} + t_i\mathbf{d}$ (4.13) and summing over t_i, we get the expected depth of the ray

$$\hat{D}(\mathbf{r}) = \sum_{i=1}^{N} p(t_i)(\mathbf{o} + t_i\mathbf{d}) \tag{4.23}$$

We can then add the loss term

$$L_{Depth} = (D(\mathbf{r}) - \hat{D}(\mathbf{r}))^2 \tag{4.24}$$

, where $D(\mathbf{r})$ is the true depth of the pixel, to the cost function 4.4

$$L_{Color} = (C(\mathbf{r}) - \hat{C}(\mathbf{r}))^2 \tag{4.25}$$

and optimize for the total loss

$$L_{Total} = L_{Color} + \lambda L_{Depth} \tag{4.26}$$

where λ is the hyperparameter controlling the weight of depth supervision. The final cost function we optimize is

$$L = \sum_{\mathbf{r} \in R} \Big((C(\mathbf{r}) - \hat{C}(\mathbf{r}))^2 + \lambda (D(\mathbf{r}) - \hat{D}(\mathbf{r}))^2 \Big) \tag{4.27}$$

4.4.1 Project RGB frames Onto Depth Frames

To find the depth information of the pixels, we need to project RGB frames onto the depth frames. In section 3.1, we introduced how to project depth frames onto the RGB frames to color the points. By running the same steps, we can map a pixel to a 3D point if it ever gets projected. For each RGB frame, we find the closest depth frame by timestamp and only projects the pixels if the timestamp difference is acceptable. Since the RGB frames come in a higher frequency (30 fps) than the depth frames (5 fps), only a portion of the RGB frames will be projected. Also, the RGB camera has a bigger field of view, so only a portion of the pixels in each frame can be mapped to 3D points. Nevertheless, the depth data we acquire from this process is enough for depth supervision.

One point worth mentioning is that since there exists a time difference between the RGB frame and its projected depth frame, we are essentially mapping an RGB frame to a depth frame taken at a slightly different time. During the period, the host may have moved or turned his head around, which led to slightly different views. Therefore, the depth values we get for the pixels can be biased. However, since we only use depth loss as a weighted term in the cost function, it still provides good supervision.

4.5 Rendering

For NeRF, rendering a novel view is to run the inference process for all pixels in the view image. Recall Equation 4.3, after training the MLP, we need to call g to sample points along the rays traced from every pixel and then perform volumetric integration to compute the final color. This process can be computationally heavy. For example, to render an image of 1080 x 1080 pixels and 100 points sampled along each ray, we need to query the MLP 116.64 million times and this would take several seconds on a high-end GPU. Clearly, it is too slow for a real-time VR application that needs to render at 60 fps.

As mentioned in subsection 2.1.3, How to speed up the rendering process is currently an active research area. The most recent works that enable the inference process on a mobile GPU is MobileNeRF [5]. It represents a NeRF model with a set of textured polygons and takes advantage of the z-buffers and fragment shaders technology provided by modern GPU hardware to achieve massive pixel-level parallelism. Currently, only SNeRG [14] and MobileNeRF have been shown to work on mobile GPUs.

Since our application requires real-time rendering, if we were to perform local inference on the VR device, only SNeRG and MobileNeRF satisfy the requirement. However, both methods require hours of training time on a high-end GPU which clearly does not satisfy our requirements. Therefore, we stick to Nvidia's Instant-NGP framework [24] plus our depth-supervision improvement for training and rendering. We propose a client-server architecture that performs inference on the GPU server and streams the rendered image to the VR device in an interactive way. This method takes advantage of the powerful server GPU for inference and fast training time provided by Instant-NGP. Also, the high-speed Wifi and 5G technology provide low network latency which makes remote rendering possible. In fact, there are already XR streaming services in the industry such as NVIDIA CloudXR [9].

4.5.1 Implementation Details

Rendering is an interactive process between the GPU server and the client's Meta Quest 2 device. For each frame in the device rendering loop, we acquire the current head pose provided by Quest 2 and then send it over to the GPU server to run inference process. Once the GPU generates a rendered image, it copies to CPU for serialization and returns it back to Quest 2 for display. We will repeat this step for each frame update in the VR device's rendering loop.

For efficient client-server communication and best performance, we implement the communication process with remote procedure calls (RPC) in C++ using Google's Remote Procedure Call library (gRPC). We use Google's Protocol Buffers (Protobuf) for data serialization and deserialization. The .proto file is defined with the following Protobuf messages and RPC service:

```
// The request message containing a 4x4 transformation matrix
// that represents the head pose, the matrix is represented
// by a 16-elements float array
message Transformation {
    repeated float data = 1;
}

// The response message containing the rendered image in the
// form of a byte sequence
message Image {
    bytes data = 1;
}

// The RPC service that takes a head pose and returns a
// rendered image from the trained NeRF model
service Render {
    rpc Render (Transformation) returns (Image) {}
}
```

We use the protocol buffer compiler to compile and generate source code for the message structures and RPC calls. Then, we include the generated code in our server and client application. The server code is written in Python using the Pybind interface to the C++ code of Instant-NGP. It runs an infinite loop until interrupted and performs the following operations in each iteration:

1. Wakes up for an RPC call
2. Receives the Protobuf message from the client and deserializes it into a head pose in space
3. Decompose the head pose into a 3D position and viewing direction as the input to the Instant-NGP NeRF model
4. Run Instant-NGP's inference process on GPU for every pixel in parallel
5. Combines the color result of each pixel to generate a full image and copy the data out from GPU to CPU
6. Serialize the image into a Protobuf message on CPU and sends it back to the client over the network
7. Go to sleep and wait for the next RPC call

The Python programming language brings us simplicity and quick development speed to implement the server application. On the other side, since Python is just a wrap-

per and both gRPC and Instant-NGP use C++ to implement the computationally heavy part, the performance degradation introduced by Python interpreter is minimum.

For the client, since it is a mobile VR device with limited computing power, we need to optimize the application performance as much as we can to make room for the server-side rendering delay. Since we choose Meta Quest 2 as our client VR device and it runs an Android operating system, the standard way to implement the client application is to use Java with Android Software Development Kit (SDK). However, running the Java Virtual Machine can be slow and the uncertainty of garbage collection time does not meet our real-time rendering requirement. Fortunately, Android and Quest 2 provide another Native Development Kit (NDK) that enables us to implement the app in C++ and directly compile into machine code for maximum performance. For each iteration in the device rendering loop, the client APP performs the following operations:

1. Wake up for the current rendering frame and acquire the head pose from the API of Quest 2's NDK
2. Serialize the head pose into a Protobuf message, transmit it to server over the network using RPC, and wait for a reply
3. Receive the Protobuf message reply from the server and deserialize it into a rendered flat image
4. Apply the proper transformation to convert the flat image into a stereoscopic image suitable for VR viewing
5. Render the stereoscopic image for the current frame
6. Go to sleep until the next rendering frame

In general, the server performs the computation and streams VR content to the client for display. Figure 4.1 shows a diagram of the whole application architecture.

4.5.2 Performance

The Quest 2 headset has a display resolution of 1832 x 1920 per eye and supports up to 90 Hz refresh rate. While the recommended frame rate is 90 fps, it is more than enough for a regular utility APP. The 90 frame rate requirement is actually for intense games so that it won't introduce nausea and disorientation to the player. For

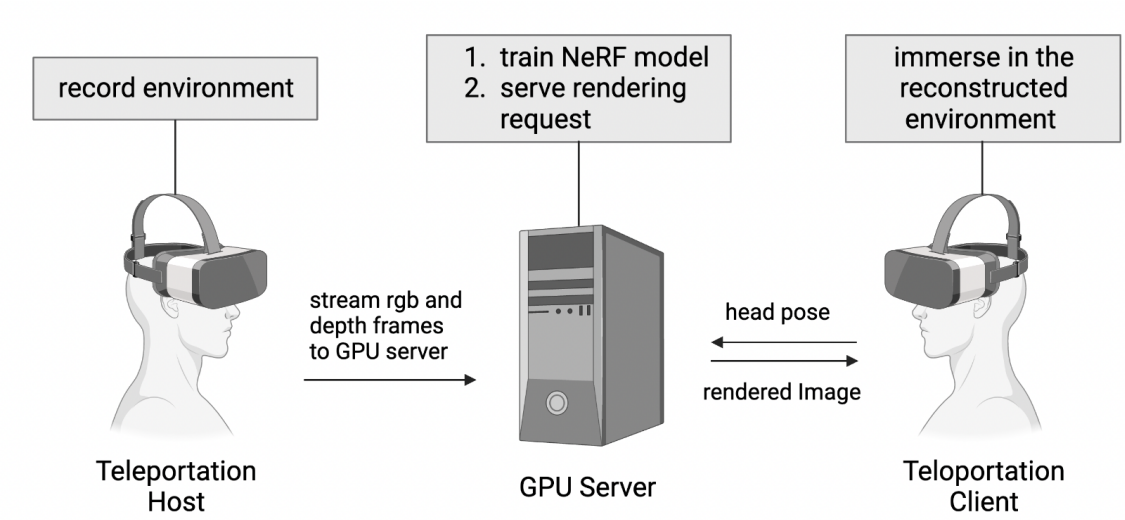


Figure 4.1: Software architecture of the NeRF approach

our application, we will only target 15 fps (66.6 milliseconds per frame) as it is enough for a smooth viewing experience. In our lab GPU server, we managed to render a 720 x 720 image for each frame within 52 milliseconds. Figure 4.2 shows a latency breakdown of the time spent at different stages.

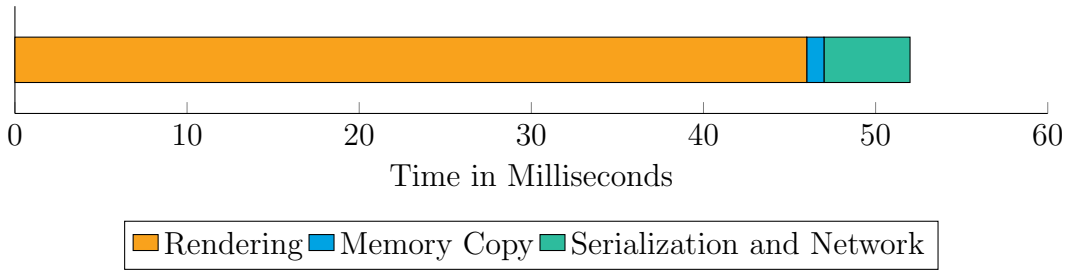


Figure 4.2: Latency Breakdown

Rendering takes 46 milliseconds on the GPU, then it takes 1 millisecond to copy the data out to CPU. Finally, it takes 5 milliseconds to perform data serialization and network transfer. We will experiment with more resolutions in the next chapter.

Chapter 5

Results

In this chapter, we explain the experiments we run with the 3D point cloud reconstruction and NeRF approach and show their results. We first introduce the experiment setup by explaining the data collection process and elaborating on the computational hardware. Then we show the 3D point cloud reconstruction result and model performances over different downsampling voxel sizes. Next, we introduce the training parameters used in the Instant-NGP model and the NeRF reconstruction result. We also demonstrate the accuracy and performance improvement of the model with our depth-supervision optimization. Finally, we show the rendering speed difference with different resolution and samples per pixel (SPP) values in Instant-NGP.

5.1 Experiment Setup

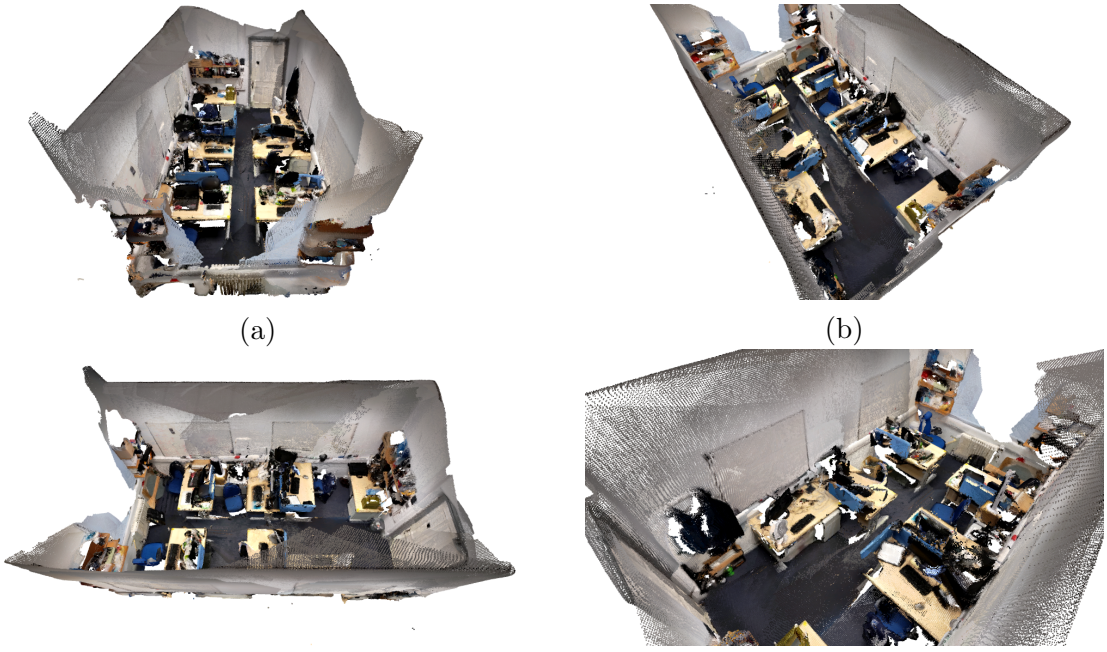
We try to reconstruct a $15m^2$ office of the Systems Security Lab in the Robert Hooke Building, University of Oxford. To record the environment, We use the StreamRecorder application provided by the Microsoft Hololens 2 research mode [39] Github repository to collect data for about one minute. We make the teleportation host wear the Hololens 2 and walk around the office with his head constantly moving to cover as much area as possible. Since NeRF works well when the images are well captured and non-blurry, we ask the host to stay stationary from time to time so that we can capture some clear images. We stop data collection when the host has walked around every area in the office to maximize the scanning region. Since the application records frames at the same rate as the sensors (30 fps for RGB camera and 5 fps for depth camera), it leads to 1920 RGB and 320 depth frames in a 64 seconds recording session. We discard the first and last seconds of data which contains the hand manipulating

the application menu in the view. We then send the recorded data to a GPU server to color and merge the point clouds and train the NeRF model. The server has an Nvidia RTX 3080 GPU and a 10-core Intel i9-10900K CPU @ 3.7 GHz. For the 3D point cloud reconstruction approach, once we have the merged point cloud, we pack it in a ply format file and send it to the client's Quest 2 for rendering. For the NeRF approach, as mentioned in section 4.5, we offload the rendering task to the GPU server and stream the rendered images to Quest 2 through RPC calls.

5.2 3D Reconstruction Result

We initially have around 320 depth frames. After filtering out the first and last second, we have 310 frames left, and each frame has an average of 72 thousand points. After the point cloud coloring process, we crop the frames to around 43 thousand points each. This still leads to 13 million points in the merged point cloud which is incapable of rendering on a mobile device. Therefore, we perform the voxel grid downsampling process with 2cm voxel size. This reduces each depth frame to around 8000 points. Then, we run the point clouds alignment algorithm and another downsampling after the merging process. The ICP algorithm helps reduce the average fitness score from 0.0063 to 0.0052 between the point clouds. The final merged point cloud has around 2.5 million points. For rendering, we make each point an 8 cm^3 cube so that it fills the whole voxel.

Figure 5.1 shows the reconstructed model of our office from different angles. We purposely did not capture the ceiling so that we can better view the model from the top. In the pictures, we can see that the cuboid office space is well reconstructed even though we capture it from the inside. We can clearly tell objects by their shape and color such as desks, chairs and computers. You may see white areas which do not contain any points. They are blind spots that the teleportation host did not reach during the data recording process. One solution to reduce such blind spots is through an interactive recording session. During recording, the application can memorize visited area and guides the host to the unexplored space so that it can cover the whole environment. We will discuss it further in the final chapter.

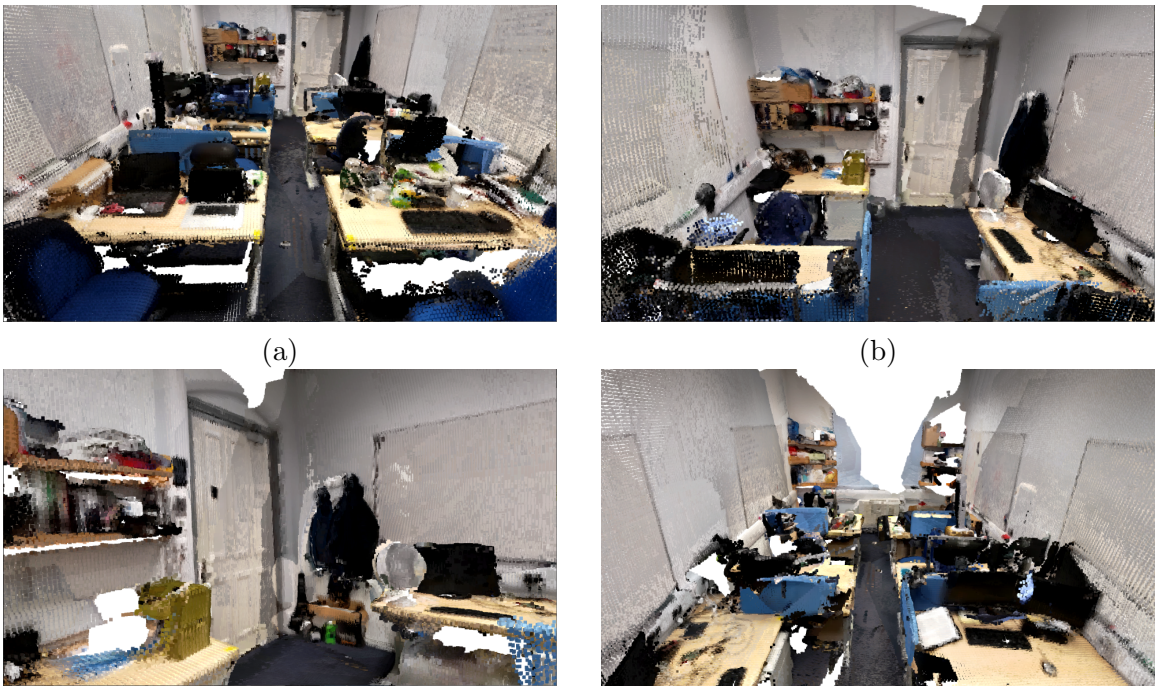


(a) (b) (c) (d)

Figure 5.1: **3D Reconstructed Office Model.** It shows the reconstructed office model from four different angles. The first three views are generated from the same zoom level by just rotating the office model. The last view is zoomed in for better illustration

Figure 5.2 shows example views from inside the reconstructed office. Again, the white area under the desks and next to the yellow box are blind spots. In areas with dense points, the surfaces are well reconstructed and we can even see object details such as logos on the box and the structure of the electric fan.

You may notice that some areas are discretized where the points do not contact each other to form a solid surface. The reason is that although we set the point size to fill the whole voxel, there may still be spots in space that do not get scanned in all of the depth frames. Those spots lead to blank voxels in the final rendered view. However, since the depth sensor scans the environment in a fixed step pattern, the voxels will be uniformly filled and blank voxels are discrete. Therefore, it will not significantly impair the visualization quality. We will discuss a potential solution that uses adaptive-sized points to fill the discretized blank voxels in the final chapter.



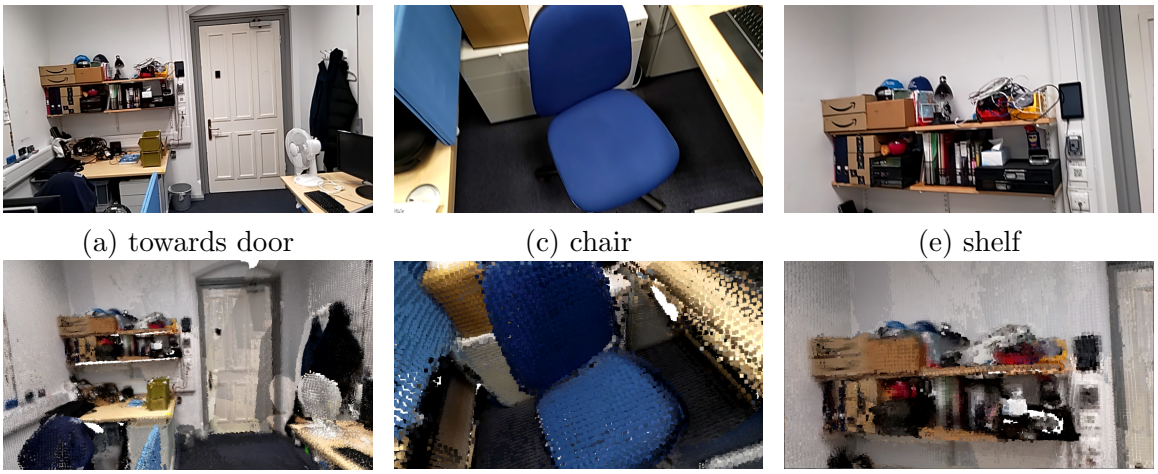
(a) (b)

(c) (d)

Figure 5.2: Views inside the 3D Reconstructed office

5.2.1 Quality of the Reconstructed Mode

Since we do not have the ground truth of our office model. Quantitatively measuring the reconstructed model quality is hard. Therefore, We will just show real camera images and virtual views from the reconstructed model of the same scene side by side in Figure 5.3 for the readers to compare by eye.



(a) towards door (c) chair (e) shelf

(b) towards door (d) chair (f) shelf

Figure 5.3: Comparison of the reconstructed and real camera views

As we can see from the image, the shapes and colors of the objects are well captured and rendered. However, some of the close views may look like mosaics because of the point-based rendering scheme and fixed grid resolution. We will not discuss a solution here and leave it for future work.

5.2.2 Performance and Parameter Tuning

To improve the rendering speed, one parameter we can tune is the voxel size. The smaller the voxel is, the more points we sample, and thus the better resolution we achieve. However, it may slow down the alignment algorithm and increase rendering costs. We experimented with different voxel sizes and generated a table in Figure 5.4 on the number of points, point cloud alignment time and rendering capability for different voxel sizes. The line graphs below are visualizations of the table. Figure 5.5 compares the same view rendered with different voxel sizes. From the table and the view comparison, we can see that setting the voxel size to be 2 cm archives both performance and rendering quality.

Voxel Size	0cm	1cm	2cm	3cm	4cm	5cm
Num Points	13464587	7472467	2512322	1221491	719893	479118
Alignment time (Sec)	42.21	16.63	3.39	1.91	0.85	0.58
Rendering Capability	No	15 fps	60 fps	60 fps	90 fps	90 fps

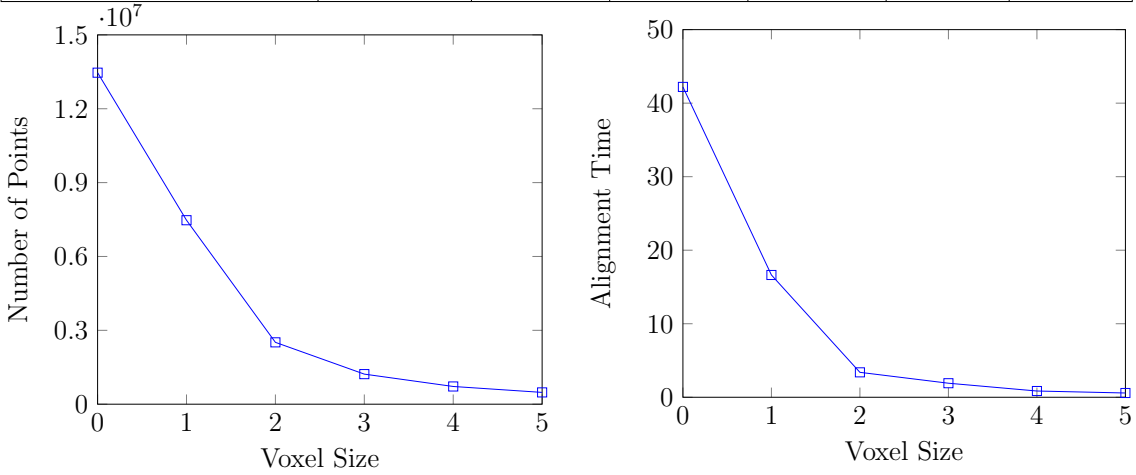
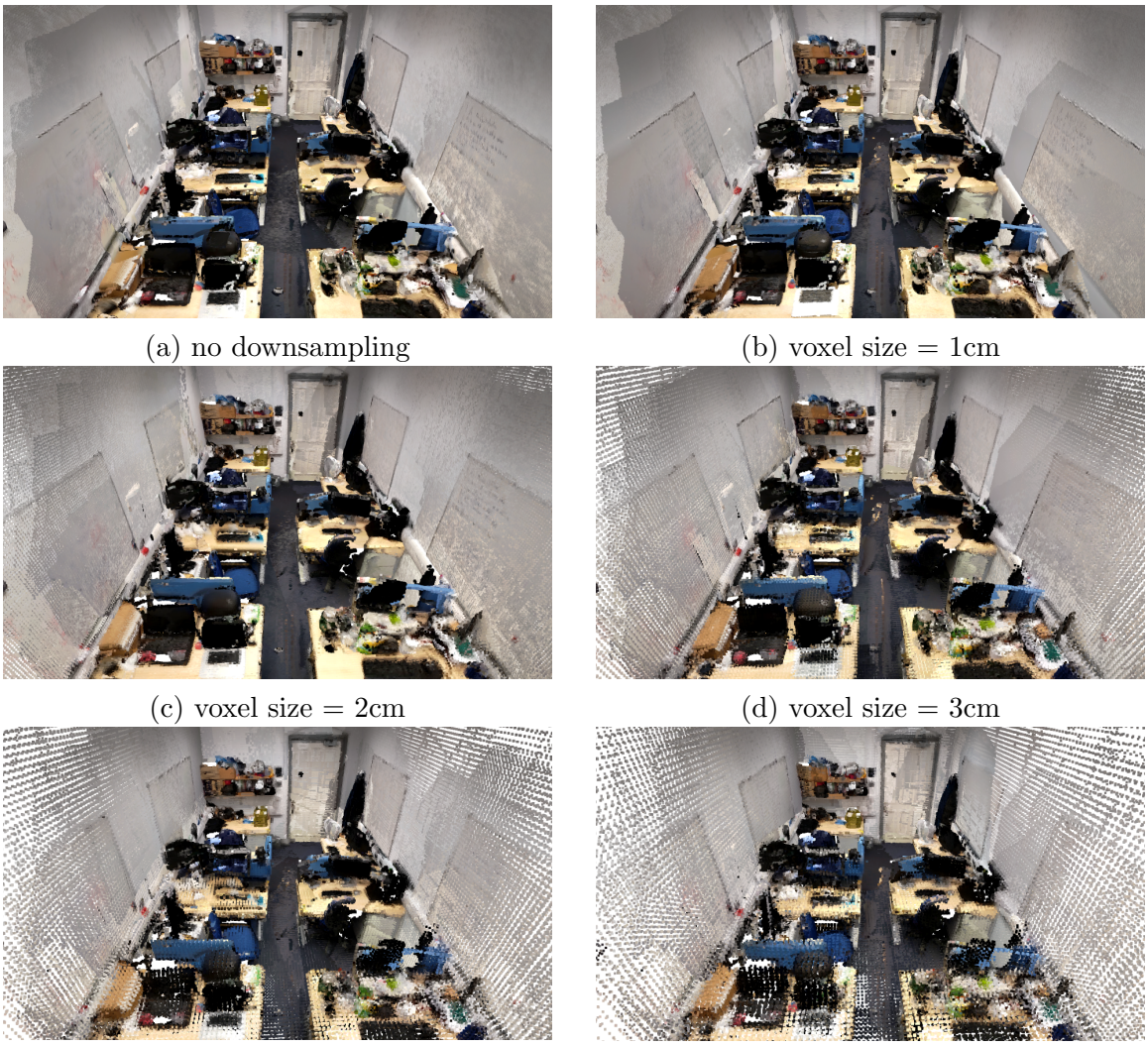


Figure 5.4: **3D Reconstruction Performance Over Voxel Size** 0 cm means no downsampling. The rendering capability row shows the maximum frame rate Meta Quest2 can achieve to render the final merged point cloud. The line graphs show a visualization of the Number of Points and Alignment time over the voxel size



(a) no downsampling (b) voxel size = 1cm

(c) voxel size = 2cm (d) voxel size = 3cm

(e) voxel size = 4cm (f) voxel size = 5cm

Figure 5.5: Comparison between the same view with different voxel sizes

5.3 NeRF Result

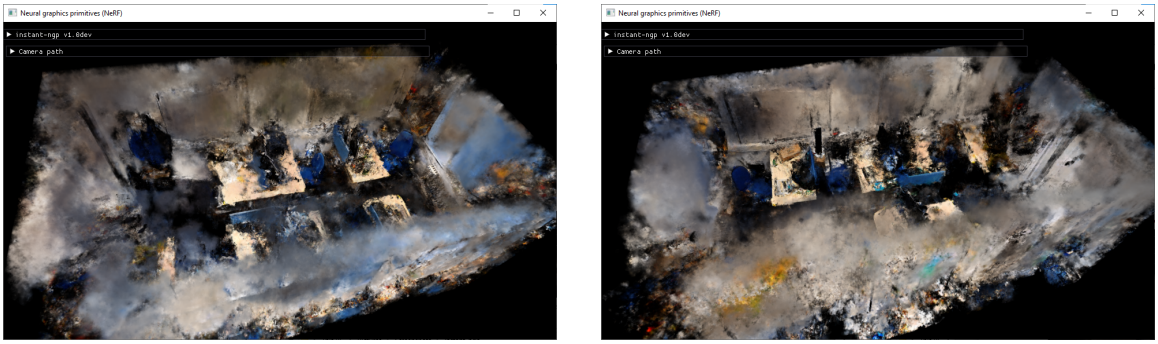
5.3.1 Training Parameters

We use Nvidia's official implementation of Instant-NGP to train the NeRF model. We use 150 images of 760x428 pixels as the training set and 10 images as the test set. The images are subsampled with the best sharpness using the technique introduced in subsection 4.1.1. We use a batch size of 262144 rays and stop training until the loss function delta becomes less than 0.0001. We set the depth supervision lambda to be 0.3. For the multiresolution hash table, we use $L = 16$ levels, $T = 2^{19}$ feature vectors on each level, and set the dimensionality of each feature vector to be $F = 2$. We set base resolution $N_{min} = 16$. For the complimentary small neural network, We

use a fully connected MLP with only one hidden layer and 64 neurons. We use an exponential decay Adam optimizer with learning rate = 0.01, $\beta_1 = 0.9$, $\beta_2 = 0.99$, $\epsilon = 10^{-15}$ and L2 regularization = 10^{-6}.

5.3.2 Implicit Model Reconstruction

Although NeRF does not explicitly create a 3d model, the radiance field can represent a model implicitly through volume rendering. Figure 5.6 shows an implicit office model by rendering a view of a top-down camera pose. We crop out the irrelevant areas in the rendered view for better visualization.



(a) (b)

Figure 5.6: Implicit model representation by NeRF

You may notice that the model has many foggy areas, this is because we are using a particle-based system to perform volume rendering. Since the colors along a light ray are independently computed and only aggregated to render the final color of a pixel, they may have high variations in certain areas. Also, the inside structure does not look as clear as the 3D point cloud reconstruction approach. This is because we are viewing the scene from an outside position that does not have any training images. However, it does not imply a bad reconstruction result because NeRF does not learn to represent the 3D model. Instead, it only generates novel views based on the implicit model it learns. In the next section, we will show some photorealistic inside views that are really close to the reality.

5.3.3 Reconstruction Quality

To test the scene reconstruction quality of our model, we run the inference process to generate novel views using the camera poses of the test image set. Then we compute the average Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index Measure (SSIM) [42] and Learned Perceptual Image Patch Similarity (LPIPS) [45] between

the rendered and ground truth test images. The higher the PSNR/SSIM and the lower the LIPIS, the better. Figure 5.7 shows a table of the reconstruction quality metrics and loss function change over different numbers of iterations. Corresponding line graphs are attached below to show visualizations of the table.

Num Iterations	Training Time	Loss	PSNR ↑	SSIM ↑	LPIPS ↓
2000	23 s	0.0051	20.37	0.675	0.345
5000	46 s	0.0045	21.05	0.695	0.285
8000	74 s	0.0040	21.44	0.708	0.261
10000	88 s	0.0038	21.64	0.712	0.248
12000	115 s	0.0038	21.67	0.713	0.240

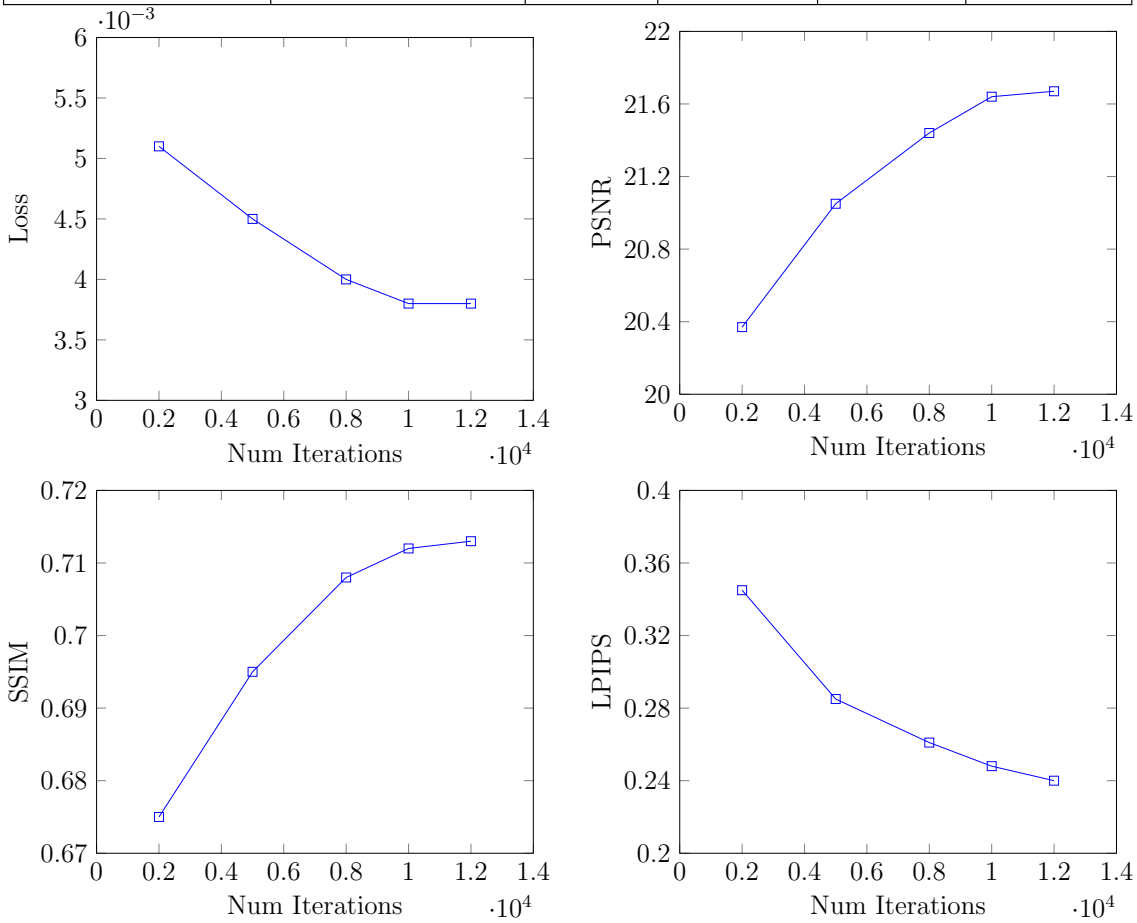
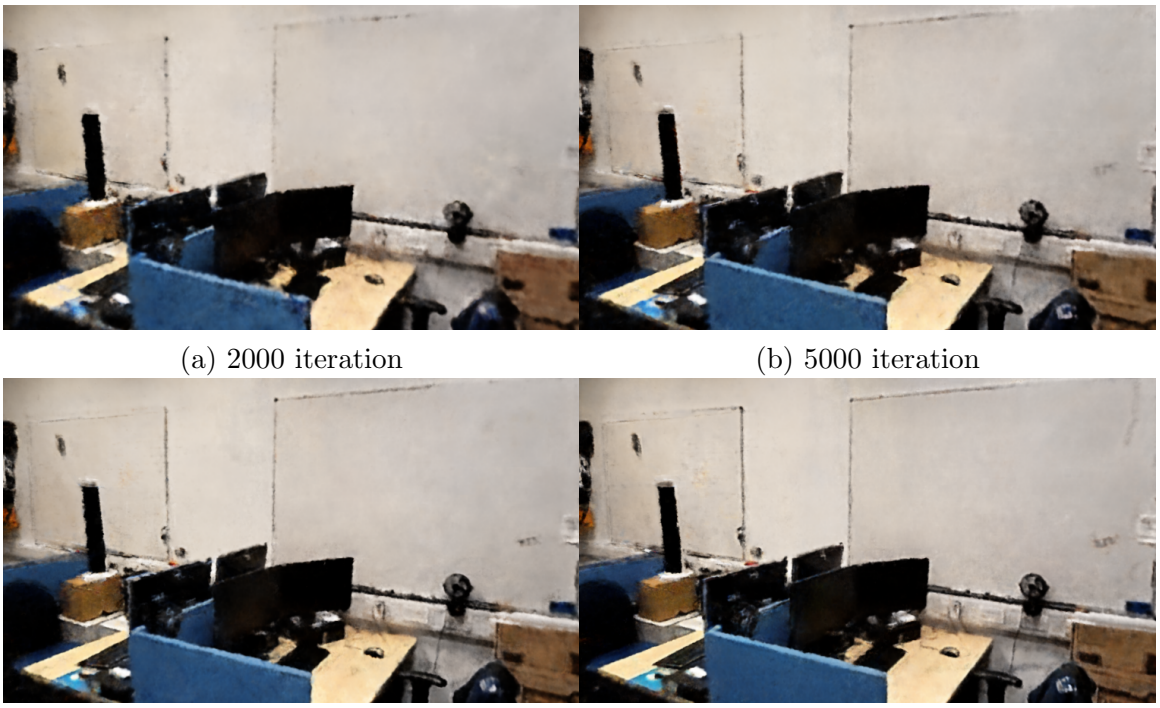


Figure 5.7: NeRF reconstruction metrics over different number of iterations

From the table and the graphs, we can clearly see that the metrics are becoming better as the training goes on. After around 10000 iterations, the model starts to converge and we stopped training short afterwards. Figure 5.8 shows a rendered view from different number of iterations. We can see that the view is becoming visually better along the training.



(a) 2000 iteration (b) 5000 iteration

(c) 10000 iteration (d) 12000 iteration

Figure 5.8: Image rendered at different iterations during NeRF training

Figure 5.9 shows a comparison between the ground truth and rendered images. We also compute a difference image and put it on the side to help the comparison. The difference image is constructed by calculating the absolute value of color difference for each pixel. Therefore, the darker the area, the closer the two images are on the pixel level.

From the figure, we can see that the ground truth and rendered images are very visually close. We can clearly see geometry and object details in the rendered images, such as thin cables and the structure of the electric fan. Hinted by the difference image, the most discrepant part between the ground truth and rendered images is around the edges. The rendered image tends to have rough edges, while in reality, the objects' edges are usually smooth. Again, this is because NeRF s unaware of the scene geometry and it is based on a particle system where pixels are independently computed and rendered by aggregating the particles. This problem has been addressed by RegNeRF [27] where the author improves it by adding a geometry and color regularizing term during the training process. Nevertheless, the novel views generated by NeRF are photorealistic and really close to reality.

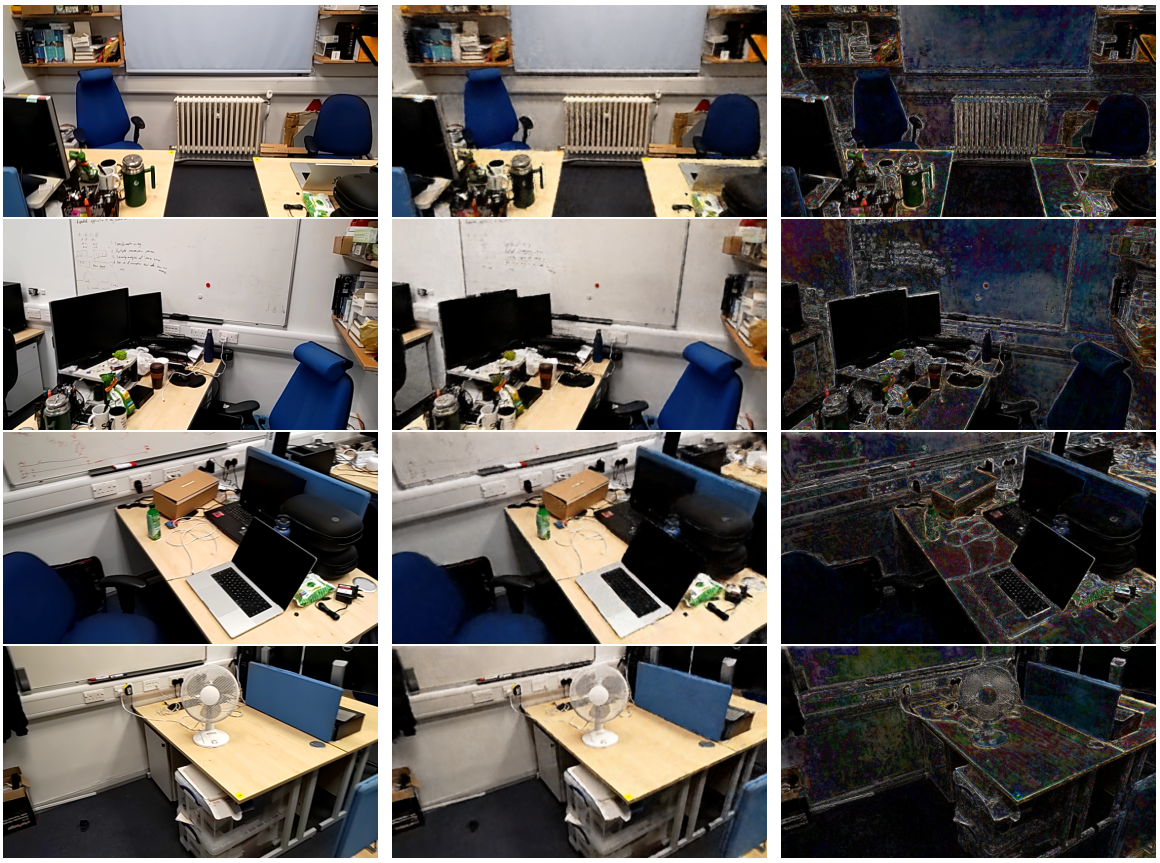


Figure 5.9: **Comparison between the ground truth and rendered images** The first column are the ground truth images from the test set, the second column are the rendered images by NeRF and the third column are the difference images

5.3.4 Depth Supervision Improvement

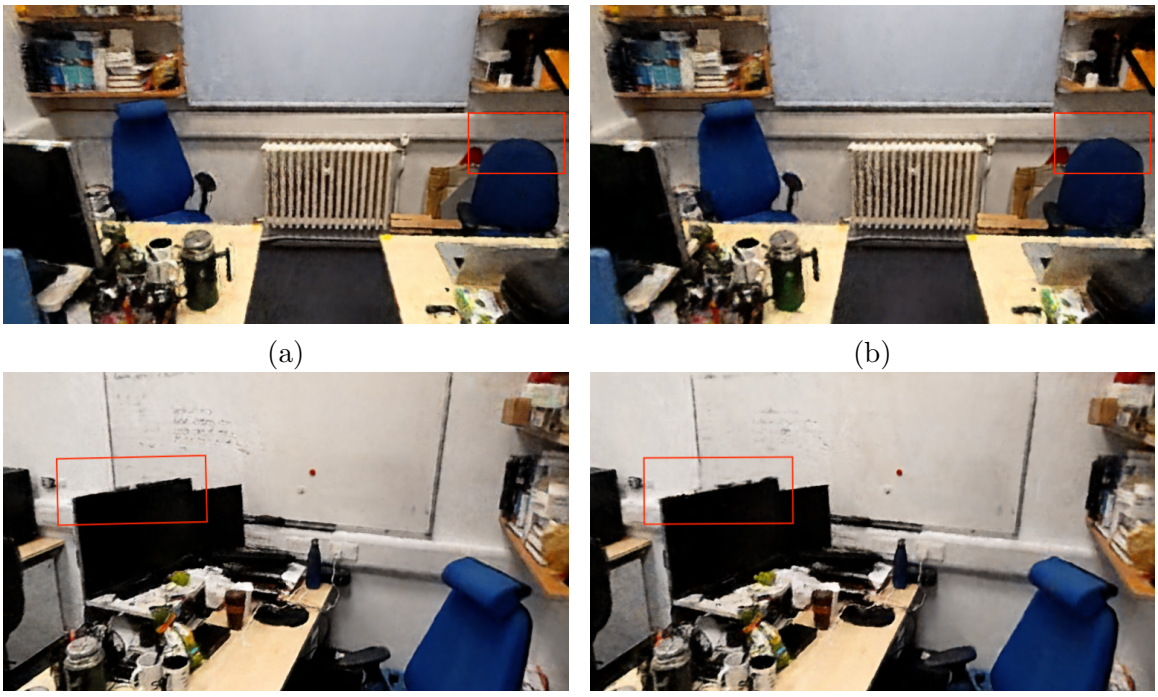
By adding depth supervision to Instant-NGP, we are able to achieve better results and faster training speed for convergence. To compare, we train the model with and without depth supervision until convergence and we record the reconstruction quality metrics, training time and number of iterations before converging. The result is shown in Table 5.1.

	PSNR ↑	SSIM ↑	LPIPS ↓	Num Iterations	Training Time ↓
W/O Depth	21.84	0.716	0.268	13559	124 s
With Depth	22.95	0.747	0.239	11025	95 s

Table 5.1: Model Performance With and Without Depth Supervision

As we can see from the table, the model converges with fewer iterations while achieving better reconstruction quality with depth supervision. Figure 5.10 shows a com-

parison between rendered views with and without depth supervision. You can see that views with depth supervision are better in detail.



(a) (b)

(c) (d)

Figure 5.10: **Depth Supervision View Comparison** The left column are views with depth supervision and the right column are views without depth supervision. The red box highlights some of the visually noticeable differences

5.3.5 Rendering Speed

As mentioned in subsection 4.5.2, we experimented on the rendering speed of Instant-NGP by tunning two parameters: the rendered image resolution in pixels and samples per pixel (SPP). We achieved 15 fps with 720p * 720p and 1 SPP. with 720p * 720p and 2 SPP, we could still achieve 10 fps. We tested with higher resolutions and SPPs just for reference.

	Rendering	Memory Copy	Serialization+Network	total
720 * 720, 1 SPP	46 ms	1 ms	5 ms	52 ms
720 * 720, 2 SPP	91 ms	1 ms	5 ms	97 ms
1080 * 1080, 1 SPP	178 ms	3 ms	6 ms	187 ms
1920 * 1920, 1 SPP	869 ms	6 ms	6 ms	881 ms

Table 5.2: Instant-NGP rendering speed with different parameters

Chapter 6

Conclusions

In this thesis, we have successfully achieved VR "teleportation" by reconstructing and rendering the environment in real time through the 3d point cloud and NeRF approaches. For the 3D point cloud reconstruction approach, we introduced methods to create a point-based model of the environment. We showed through the experiments that the model reconstruction could finish within a few seconds, and the rendering speed could reach 60 fps on the Meta Quest 2. For the NeRF approach, we successfully trained an Instant-NGP NeRF model to implicitly model the environment and presented the reconstruction results by showing close-to-reality, photorealistic rendered views. We demonstrated our depth-supervision technique's effectiveness by comparing the reconstruction metrics and rendered views with and without the technique. We showcased the client-server architecture that enables training to finish in two minutes and renders up to 15 fps.

6.1 Model Comparison

By comparing rendered views with real camera images, we can see that the photorealistic views generated by NeRF are superior to the point-based views of the 3D point cloud reconstruction approach. When in the distance, the point-based views have good quality. However, the object surface may appear like mosaics for closer views because of the point-based rendering system and the fixed resolution 3D grid. NeRF does not have this issue since it always renders a photorealistic view. On the performance side, the 3D point cloud approach has faster model reconstruction and rendering speed. It can render the merged point cloud on device with 60 fps.

NeRF, on the other hand, can only perform remote rendering on the server due to the GPU-heavy inference process. The maximum frame rate it can achieve is 15 fps.

6.2 Improvements and Future Work

We mentioned in section 5.2 that our current data collection approach leaves blind spots in the reconstructed point cloud and an interactive data recording process can resolve them. During the data recording, we make the application memorize explored area and localize itself through simultaneous localization and mapping (SLAM) [10]. Then we can have the application guide the host to unexplored areas using a virtual map. The data recording process can automatically stop when the map is fully constructed.

Another improvement for the rendering quality of the 3D point cloud reconstruction approach is to fill the empty voxels in the final merged point cloud. We can use a dynamic point size system where we increase the size of the adjacent points in areas with empty voxels. The final goal is to fill every empty voxel so that we can render concrete surfaces.

To further improve the rendering speed of NeRF. We can employ the gaze-contingent rendering technique. For VR devices equipped with eye-tracking sensors, we will know where the user is gazing in real time. Since NeRF renders with the same speed when the total number of pixels is the same. We can render a dynamic resolution image where only the gazed part has high resolution and the rest are left with lower resolution. This idea is implemented in FoV-NeRF [8] where the reader can further read.

Bibliography

[1] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. Spie, 1992.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] Anpei Chen, Zexiang Xu, Fuqiang Zhao, Xiaoshuai Zhang, Fanbo Xiang, Jingyi Yu, and Hao Su. Mvsnerf: Fast generalizable radiance field reconstruction from multi-view stereo. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14124–14133, 2021.

[4] Shenchang Eric Chen. Quicktime vr: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 29–38, 1995.

[5] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. *arXiv preprint arXiv:2208.00277*, 2022.

[6] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996.

[7] Kangle Deng, Andrew Liu, Jun-Yan Zhu, and Deva Ramanan. Depth-supervised nerf: Fewer views and faster training for free. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12882–12891, 2022.

[8] Nianchen Deng, Zhenyi He, Jiannan Ye, Praneeth Chakravarthula, Xubo Yang, and Qi Sun. Foveated neural radiance fields for real-time and egocentric virtual reality. *arXiv preprint arXiv:2103.16365*, 2021.

[9] Nvidia Developers. Nvidia cloudxr sdk. `https://developer.nvidia.com/nvidia-cloudxr-sdk`, May 2022.

[10] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[11] Ashley Eden, Matthew Uyttendaele, and Richard Szeliski. Seamless image stitching of scenes with large motions and exposure differences. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2498–2505. IEEE, 2006.

[12] John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. Deepview: View synthesis with learned gradient descent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2367–2376, 2019.

[13] Stephan J Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. Fastnerf: High-fidelity neural rendering at 200fps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14346–14355, 2021.

[14] Peter Hedman, Pratul P Srinivasan, Ben Mildenhall, Jonathan T Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5875–5884, 2021.

[15] Abhishek Kar, Christian Häne, and Jitendra Malik. Learning a multi-view stereo machine. *Advances in neural information processing systems*, 30, 2017.

[16] Maik Keller, Damien Lefloch, Martin Lambers, Shahram Izadi, Tim Weyrich, and Andreas Kolb. Real-time 3d reconstruction in dynamic scenes using point-based fusion. In *2013 International Conference on 3D Vision-3DV 2013*, pages 1–8. IEEE, 2013.

[17] Urho Konttori. Teleport to any reality – introducing varjo's vision for future immersive collaboration. `https://varjo.com/blog/varjo-teleport/`, May 2022.

[18] Jason Lawrence, Dan B Goldman, Supreeth Achar, Gregory Major Blascovich, Joseph G. Desloge, Tommy Fortes, Eric M. Gomez, Sascha Häberling, Hugues Hoppe, Andy Huibers, Claude Knaus, Brian Kuschak, Ricardo Martin-Brualla, Harris Nover, Andrew Ian Russell, Steven M. Seitz, and Kevin Tong. Project

starline: A high-fidelity telepresence system. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 40(6), 2021.

[19] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. *Advances in Neural Information Processing Systems*, 33:15651–15663, 2020.

[20] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[21] Hao Men, Biruk Gebre, and Kishore Pochiraju. Color point cloud registration with 4d icp algorithm. In *2011 IEEE International Conference on Robotics and Automation*, pages 1511–1516. IEEE, 2011.

[22] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4460–4470, 2019.

[23] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*, pages 405–421. Springer, 2020.

[24] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *arXiv preprint arXiv:2201.05989*, 2022.

[25] Zak Murez, Tarrence van As, James Bartolozzi, Ayan Sinha, Vijay Badrinarayanan, and Andrew Rabinovich. Atlas: End-to-end 3d scene reconstruction from posed images. In *European conference on computer vision*, pages 414–431. Springer, 2020.

[26] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*, pages 127–136. Ieee, 2011.

[27] Michael Niemeyer, Jonathan T Barron, Ben Mildenhall, Mehdi SM Sajjadi, Andreas Geiger, and Noha Radwan. Regnerf: Regularizing neural radiance fields for

view synthesis from sparse inputs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5480–5490, 2022.

[28] Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Colored point cloud registration revisited. In *Proceedings of the IEEE international conference on computer vision*, pages 143–152, 2017.

[29] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

[30] José Luis Pech-Pacheco, Gabriel Cristóbal, Jesús Chamorro-Martinez, and Joaquín Fernández-Valdivia. Diatom autofocusing in brightfield microscopy: a comparative study. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*, volume 3, pages 314–317. IEEE, 2000.

[31] Eric Penner and Li Zhang. Soft 3d reconstruction for view synthesis. *ACM Transactions on Graphics (TOG)*, 36(6):1–11, 2017.

[32] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310. PMLR, 2019.

[33] Daniel Rebain, Wei Jiang, Soroosh Yazdani, Ke Li, Kwang Moo Yi, and Andrea Tagliasacchi. Derf: Decomposed radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14153–14161, 2021.

[34] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14335–14345, 2021.

[35] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition. *ACM Transactions on Graphics (TOG)*, 21(3):438–446, 2002.

[36] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[37] Steven M Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, volume 1, pages 519–528. IEEE, 2006.

[38] Richard Szeliski et al. Image alignment and stitching: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2(1):1–104, 2007.

[39] Dorin Ungureanu, Federica Bogo, Silvano Galliani, Pooja Sama, Xin Duan, Casey Meekhof, Jan Stuhmer, Thomas J. Cashman, Bugra Tekin, Johannes L. Schonberger, Bugra Tekin, Pawel Olszta, and Marc Pollefeys. Hololens 2 research mode as a tool for computer vision research. *arXiv:2008.11239*, 2020.

[40] Matthew Uyttendaele, Ashley Eden, and Richard Skeliski. Eliminating ghosting and exposure artifacts in image mosaics. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 2, pages II–II. IEEE, 2001.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[42] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[43] Jackie Yang, Christian Holz, Eyal Ofek, and Andrew D Wilson. Dreamwalker: Substituting real-world walking experiences with a virtual reality. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 1093–1107, 2019.

[44] Alex Yu, Vickie Ye, Matthew Tancik, and Angjoo Kanazawa. pixelnerf: Neural radiance fields from one or few images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4578–4587, 2021.

[45] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[46] Yizhong Zhang, Jiaolong Yang, Zhen Liu, Ruicheng Wang, Guojun Chen, Xin Tong, and Baining Guo. Virtualcube: An immersive 3d video communication system. *IEEE Transactions on Visualization and Computer Graphics*, 28(5):2146–2156, 2022.

[47] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *arXiv preprint arXiv:1805.09817*, 2018.

[48] Michael Zollhöfer, Patrick Stotko, Andreas Görlitz, Christian Theobalt, Matthias Nießner, Reinhard Klein, and Andreas Kolb. State of the art on 3d reconstruction with rgb-d cameras. In *Computer graphics forum*, volume 37, pages 625–652. Wiley Online Library, 2018.