Designing Data Capturing Rig and Software for AR Experimentation

by

Federico Galbiati
Reese J. Haly

advised by

Tian Guo

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelors of Science

in

Computer Science

December 2023

1

Acknowledgements

We would like to thank Professor Tian Guo for her guidance throughout this project.

We also want to thank Ashkan Ganj and Yiqin Zhao for their guidance with the computer vision and image alignment pipelines.

Lastly, we would like to thank the WPI Computer Science Department for sponsoring the equipment used in this project and National Science Foundation under Grant #2236987.

Abstract

This paper introduces a novel approach to assist augmented reality (AR) research by creating a versatile data-capturing rig and an integrated software pipeline. In this project we focus on monocular depth estimation.

The project's primary objectives encompass designing and implementing a user-friendly software pipeline and a scalable hardware rig. The scalability of the hardware rig is enabled through the support for multiple sensors with varying computational capabilities and position on the rig. The pipeline serves as an interface to enable the simultaneous activation of a suite of sensors, session recording, post-processing of collected data, and efficient storage for future utilization. The physical rig, featuring wireless connectivity and mobility, allows mounting a computational device and an array of diverse sensors on an easily constructed frame. The seamless integration of various sensors enhances the rig's utility, allowing researchers to gather diverse datasets essential for various AR tasks, such as depth estimation, object recognition, and light estimation.

Authorship

Federico Galbiati and Reese Haly handled all hardware and software development. Both members contributed to the entire pipeline development and testing, Python client development, and physical build rig. Each team member also had a specific focus area.

Federico Galbiati handled the backend development, including control server, storage server, and Airflow server.

Reese Haly handled the development of the client capture Android app.

Both members collaborated to write the final Major Qualifying Project Report.

Contents

List of Figures

2

Introduction

Augmented Reality (AR) has entered the general consumer market through standard portable devices such as mobile phones, blending technology into the world to see the world differently or to see a different world. AR has become a way for users to interact with physical worlds through virtual overlay. Analysts estimate AR will become a $\tilde{\$}$1.2 trillion market by 2035 [1]. AR will help us with applications ranging from tourism to advertisement [2].

AR research and experimentation is an expensive, delicate, and time-consuming task. Performing AR experimentation often requires having a multi-sensor setup (e.g., RGB camera and ground truth RGB-D camera for monocular depth estimation). For example, cameras need to be space and time-aligned. Moreover, it is often necessary to set up custom drivers and libraries to collect data [3]. These sensors can also be expensive, even in thousands of US dollars [4]. Additionally, AR research often requires access to a large amount of data as it often relies on deep learning models. Scaling such endeavors to record sessions from multiple aligned sensors is necessary to facilitate the research and development process of AR experimentation.

To assist AR experimentation, we devised a versatile data capture rig to simplify data capture and alignment from two sensors. Moreover, we created a novel software pipeline to collect, store, and process data captured from a generalized list of sensors. When creating this rig and these pipelines, we worked with some specific goals in mind:

- Automatic data capturing
- Easy-to-use software
- Easy to assemble and build the physical rig

These goals aim to minimize the complexity of spinning up the software stack and building a rig. This should allow any researchers, whether technically knowledgeable or not, to set up a capturing rig quickly and easily begin recording sessions from multiple sensors.

In this project, we developed a complete data-capturing rig, including software and hardware components, to capture multi-sensor data to facilitate AR research. The scalable software stack provides a UI to allow users remote control of the session recording. A WebSocket control server coordinates the recording throughout multiple recording devices. A WebSocket storage server facilitates data saving to almost any storage system (e.g., S3, GCP, FTP, etc.) through a generalized filesystem interface. A postprocessing server triggered over HTTP POST allows performing image alignment to align and visually overlap the scenes captured from different cameras. We provided the code repository for this project on GitHub [1].

Using our data capturing rig and pipeline, we collected a dataset to validate our methodology consisting of 50 pairs of RGB and ground truth depth images. This dataset was used to ensure our hardware setup and software pipeline accuracy. It consists of a single scene but can be easily expanded to include a variety of environments.

In summary, we contributed the following to ExpAR through this major qualifying project:

- Built a distributed, generic, multi-sensor, and remote-controlled data collection pipeline. Users can install the software client on any device, such as a phone, robot, remote control car, or drone, to make it into a data-capturing device part of a more extensive coordinated setup.

- Streamlined processes include image alignment, data collection, scene alignment, remote starting and stopping, and storing data on the cloud. This allows easier user interaction with the recording devices and streamlines the steps to obtain a usable post-processed dataset.

- It provided an easily expandable industry-standard Airflow data postprocessing solution. This allows anyone to add new data pipelines to further post-process the data and automate AR research and development tasks.

- Openly available and open source code on GitHub.

- Openly available physical build design instructions, materials, and files.

[1] https://github.com/cake-lab/ExpAR-depth-datacapturing

3

Background and Related Works

3.1 Overview of existing AR data capturing methods

Over the last decade, AR technology has seen tremendous growth in the general consumer market. As a result, high-quality research on AR systems has become increasingly commonplace and valuable in the tech market. AR experimentation now observes a wealth of AR tasks such as depth estimation, object detection, and simultaneous localization and mapping [5, 6]. Each task has an active research community focused on developing systems and algorithms to solve these problems as efficiently as possible.

However, as the quantity and scale of these research experiments grows, it becomes challenging to accurately evaluate the AR systems proposed [4]. To streamline AR experimentation, a scalable and generalizable solution for evaluating the diverse range of AR tasks would be invaluable. Two approaches have emerged to address this challenge: creating large, diverse datasets for real-world environments and developing specialized capturing methods for standardized AR data acquisition.

3.2 Related Works: Diverse Datasets

As AR technology becomes increasingly accessible to a broad consumer audience, AR systems must adapt to diverse environments. Ensuring that an AR system generalizes effectively across various settings is a crucial challenge in the field. However, many studies have only evaluated their systems within a narrow range of scenes [6]. Moreover, existing datasets often focus solely on indoor or outdoor environments [7]. Nevertheless, several datasets aim to lower the entry barrier for comprehensive evaluation.

3.2.1 NYU-Depth V2

NYU-Depth V2 stands as one of the pioneering RGB-D datasets, offering 1449 meticulously labeled pairs of RGB and depth images from 464 indoor scenes captured through an RGB-D Kinect camera [8]. Additionally, it provides 407,024 raw, unlabeled frames. While widely used for computer vision depth estimation, its dependency on Kinect cameras confines the dataset to indoor settings. Further, the methodology's reliance on a single sensor for RGB and depth frames hampers its adaptability and scalability. In contrast, our data capturing setup accommodates a range of depth and RGB sensors, facilitating the acquisition of higher quality data suitable for indoor and outdoor environments.

3.2.2 ARKitScenes

ARKitScenes, a comprehensive RGB-D dataset for indoor scene comprehension, is captured using an iPad Pro and integrates synchronized data from wide and ultra-wide cameras, a LiDAR scanner, and IMU [9]. The dataset includes ARKit camera poses and scene reconstructions, serving diverse tasks like 3D object detection, scene reconstruction, and indoor navigation. However, its reliance on sensors exclusive to Apple devices restricts its generalizability. Conversely, our methodology enables the utilization of various depth data sources, such as time-of-flight or stereo cameras, in combination with any RGB sensor.

3.2.3 DIODE (Dense Indoor and Outdoor DEpth)

The DIODE (Dense Indoor and Outdoor DEpth) dataset marks a significant advancement in monocular depth estimation by offering a wide array of indoor and outdoor scenes captured using a single sensor setup—the FARO Focus S350 scanner [7]. Prior datasets tended to focus solely on indoor or outdoor environments, limiting algorithm generalization across diverse settings. DIODE's inclusivity of indoor and outdoor scenarios empowers researchers to assess algorithm performance across real-world conditions. Its high-resolution images and precise, dense depth measurements are invaluable for benchmarking and advancing depth estimation techniques. Nonetheless, the high cost and limited accessibility of the FARO sensor pose barriers to widespread adoption. In contrast, our data capture rig's versatility with an array of depth sensors empowers the acquisition of indoor and outdoor depth data.

3.3 Related Works: AR Data Capture Methods

In addition to training on pre-extisting, general datasets like those listed above, researchers usually must collect specific data in order to fine tune their systems precisely to their task. Because of this, developing AR systems often requires a specialized setup to enable data collection on specific data. Many works have included their hardware setup to enable data capturing based on specific tasks.

3.3.1 Immersive Light Field Video with a Layered Mesh Representation

Broxton et al. [10] introduces an innovative data capture rig to capture high quality immersive light field video. The rig is a six degree-of-freedom camera array: a hemispherical rig consisting of 46 action sports cameras placed equidistantly across the hemisphere. This setup enables the capture of light field video from a multitude of perspectives, essential for creating realistic, immersive virtual environments with accurate motion parallax and view-dependent reflections.

The rig's ability to capture immersive light field video greatly enhances the realism and interactivity in AR environments. By providing a multitude of perspectives, the technology developed by Broxton et al. not only achieves a high level of detail in virtual reconstructions but also significantly improves the user's sense of presence and immersion. However, while this rig is capable of capturing high quality light field video, it's versatility towards other AR tasks is limited by it's shape and lack of sensor diversity.

3.3.2 MobiDepth

Zhang et al. [11] proposes a new approach for mobile AR depth estimation by leveraging existing dual-camera systems in smartphones. MobiDepth uses iterative field-of-view cropping and heterogeneous camera synchronization to align dual cameras efficiently for use in stereo depth estimation. To evaluate their AR system, they created a simple hardware setup consisting of an Intel Realsense L435i RGB-D camera and a phone camera horizontal aligned on a small platform. While this capture rig easy to set up and implement, it has made no considerations for different sensors, different amounts of sensors, or different computational devices handling the sensors. In contrast, our capture rig has been built from the ground up to include these attributes. Moreover, our work provides a software stack to automate capturing and processing data from a range of capturing devices.

3.4 Common dataset format for AR research

For a dataset to be useful for research, it should be structured in a way that is easy to access, understand, and implement during training and testing. Because of this, most datasets follow a very similar structure. Data is usually split up by scene, each containing a list of RGB and ground truth frames. Other valuable data can be added to the dataset for greater training accuracy. For instance, ARKitScenes includes intrinsic camera parameters for each frame, while NYU-Depth V2 contains one set of intrinsic parameters for the entire dataset but includes accelerometer data for each frame [8, 9]. While almost all datasets contain a set of raw dumps of frames marked with timestamps, some contain manually labeled frames such as NYU-Depth V2 and KITTI depth [12].

4

Designing the Physical Data Capturing Rig

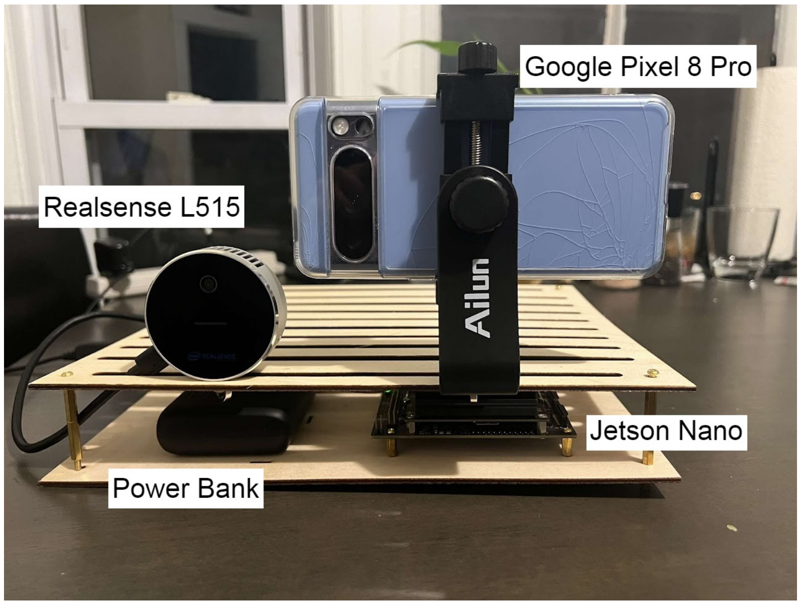


Figure 4.1: Picture of the final rig build.

To address the challenges of evaluating AR systems at scale, we designed a versatile capture rig. Our capture rig was designed to be compatible with a wide range of sensors physically mounted simultaneously while being portable enough to capture diverse scenes. The design is reminiscent of a server rack design, with one or more computational layers and a top layer to hold several sensors. As a proof of concept, we created a rig with two layers; we mounted an Intel Realsense L515 Lidar camera to collect ground truth depth data, an NVIDIA Jetson Nano as the computational host for the L515 sensor, and a Google Pixel 8 Pro to capture high-quality input images for depth estimation.

Figure 4.1 shows the complete assembled rig, including the sensors and the host device.

4.1 Hardware architecture and components required

We had three main goals for our data-capturing rig. First, we wanted our rig to be modular, allowing different sensors and computational devices to be compatible. Second, we wanted our rig to be scalable, allowing many different sensors to collect data simultaneously. Lastly, we wanted our rig to be portable, allowing it to capture many different scenes, environments, and objects. To accomplish this, we decided on a design that resembled a server rack, a rectangular platform with multiple layers for different computational devices.

In the top layer, ten horizontal cuts were made across the layer, with each cut being ¼" wide. These slots allow users to mount any sensor with a standard ¼" tripod thread to the top of the rig. Because of the design of the slots, sensors can be moved horizontally along the slots or vertically by moving between slots. Ideally, this will allow a suite of sensors to be attached to the device in whichever configuration is most optimal.

The rig’s design allocates the lower tiers for housing computational hardware. Along with several pilot holes for our computational device, we have also added a hole in the center of the bottom layer to allow a tripod mount to be added. The rig is 12" in width, 10" in depth, and 3" in height without any sensors on top. This size was chosen to allow the rig to be large enough to accommodate several sensors or computational devices while also being small enough to be portable.

Using these specifications, we produced an SVG file to create the layer platforms. We used a laser cutter to cut the designs into two 12" by 10" by ⅛" pieces of plywood. Then, using a brass standoff kit, we assembled the layer platforms by securely fastening the laser-cut plywood pieces together. The brass standoff kit facilitated a sturdy yet adjustable structure, allowing us to configure the layers according to the specific requirements outlined in the design. The precision of the laser cutter ensured accurate cuts, enabling seamless alignment during assembly. These designs are summarized in appendix A for reference and reproducibility.

Overall, the build was completed with the following bill of materials:

- Stainless Steel D Shaft D-Ring 1/4: Attached to the center of the bottom layer. Allows for mounting the rig on a tripod.

- 1/4 " "-20 Male Thread Screw Mount: Used to screw in sensors to the top layer of the rig
- Structurally strong wood or other material to laser cut: Used as the base layers of the rig.
- Tripod Phone Mount Holder with a Head Standard 1/4" Screw: Used to mount phone to the sensor layer.
- TP-Link USB Wi-Fi Adapter: Used to enable wireless communication to the host device.
- Portable Charger 22.5W 20000mAh: Used to power the host device. Allows for rig mobility.
- UBS-A to Micro USB Cable 1ft for Fast Charging and Data Sync: Used to connect the host device to the batter pack.
- M2 M2.5 M3 Male Female Hex Brass Spacers, Standoffs, Screws, and Nuts Assortment Kit: Used to attach individual rack layers together.
- Jetson Nano with SD Card: Host device used to manage sensors.
- Micro USB Power Supply (>2A): Used to power host device when mobility is unnecessary.
- Ground truth sensor (e.g., Intel RealSense L515): Used to capture ground truth data.

4.1.1 Computational Requirement of the Capture

The data capturing rig requires dedicated client device(s) to manage its suite of sensors. This includes crucial tasks like sensor synchronization and calibration, data collection, and data transfer. Client devices can include microcontrollers, phones, or any other computational device programmable to manage sensors. To fulfill the computational need to manage our sensor suite, we opted for the NVIDIA Jetson Nano.

The Jetson Nano strikes a crucial balance between compactness, performance, and power efficiency. Its capabilities enable it to simultaneously handle data streams from multiple sensors while remaining small enough to fit within the rig. Furthermore, its low power consumption allows for continuous operation powered by a power bank. However, the Jetson Nano has its own set of drawbacks: its limited I/O ports restrict the number of sensors per control board, and while powerful compared to other options, it might face challenges

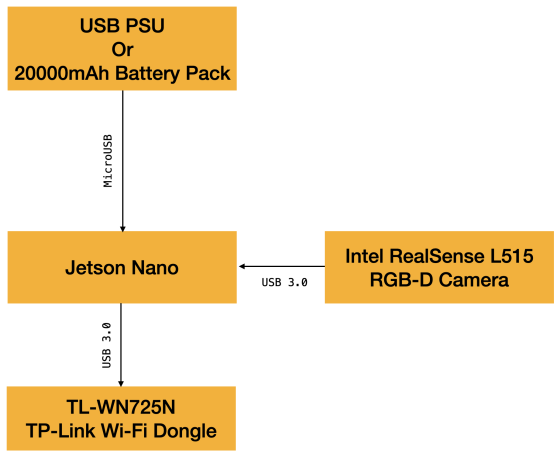


Figure 4.2: Cable connections of the various hardware components.

handling numerous data streams concurrently. Despite these drawbacks, the Jetson Nano met our requirements and was integrated into the final rig design. Figure 4.2 shows the cable connections from the Jetson Nano to its sensor and other necessary peripherals.

4.2 Selection of sensors

To ensure that our dataset was useful, we needed to use high-quality sensors that provided accurate and high-quality RGB and ground truth images. We used an Intel Realsense L515 Lidar camera to collect our ground truth depth images and a Google Pixel 8 Pro to collect our input RGB images.

4.2.1 The Intel Realsense L515 as the ground truth

The L515 sensor operates as an RGB-D camera leveraging a Micro-Electro-Mechanical System (MEMS) mirror for environmental scanning. It generates high-resolution depth maps with exceptional precision, maintaining accuracy within 14mm up to $9m^2$ at a smooth framerate of 30 fps [13]. This capability facilitated the collection of remarkably precise depth data. Integrating the L515 into our setup was facilitated by its ¼" tripod screw interface, allowing for effortless attachment to the top layer of our capture rig, as shown in Figure 4.3.

Several other depth sensors could have been used to capture depth information. The Microsoft Kinect has historically been popular for assembling datasets containing depth information [14]. Its widespread availability and affordability have made it valuable for research. However, its depth mea-

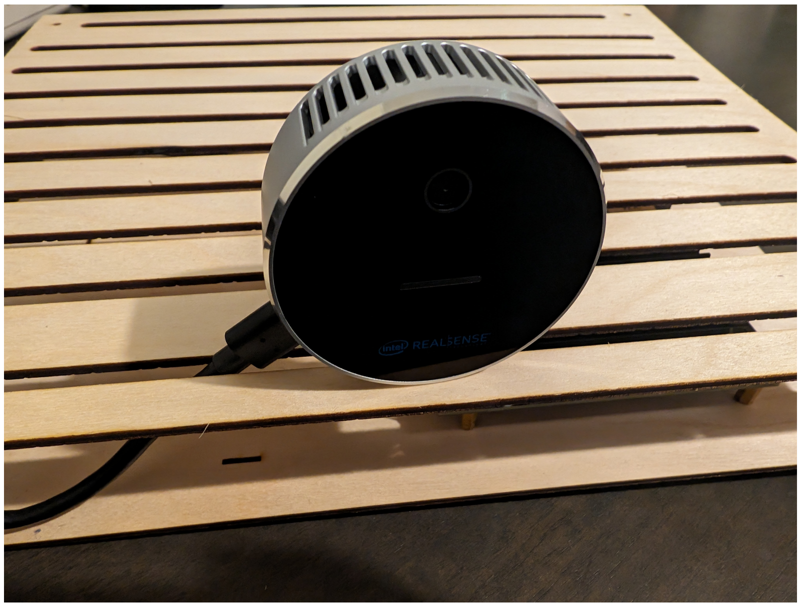


Figure 4.3: Picture of the Intel Realsense L515 Lidar camera attached to the capture rig.

surement technique (structured light) is ineffective in bright sceneries and, therefore, has limited application outdoors [15].

Time-of-flight sensors like the Intel Realsense D435 could also have been used as they are compact enough to fit atop the rig and provide accurate depth information. However, time of flight sensors tend to be less accurate and have shorter ideal ranges compared to Lidar-based depth cameras [16]. For these reasons, we chose the L515 Lidar sensor as it combines accuracy, range, and compactness to be an effective camera for collecting depth information.

4.2.2 Google Pixel 8 Pro

To capture input RGB images, we used the main 50 MP rear camera on a Google Pixel 8 Pro. This was chosen mainly for its availability to us and its incredible image quality. Since most people have access to an Android phone with a relatively high-quality camera, we believed it would be important to create an Android application to allow these devices to interface with our data-capturing architecture, lowering the barrier of entry to data collection. Consequently, we needed to create an Android application that would enable the phone to be integrated into our pipeline, which is detailed in section 5.5. The Pixel 8 Pro was attached to our rig using a phone mount with a ¼" tripod screw interface which allowed the phone to be placed sideways, as shown in Figure 4.4. This allows for close placement to the L515 sensor to ensure

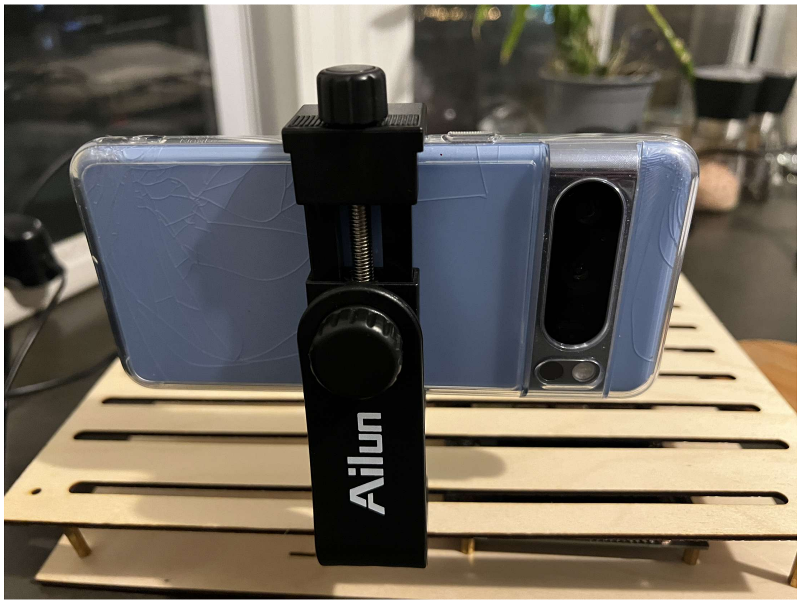


Figure 4.4: Picture of the Google Pixel 8 Pro attached to the capture rig.

approximate manual alignment between the RGB input and the ground truth depth images. It also guarantees that post-processing can correct any disparities in height or orientation.

5

Scalable Software Stack

5.1 Overview of the software stack

The software stack was designed to record, store, and process the data from the rig. The software was split architecturally into microservices to keep the processing power requirement low for the recording device. The recording client, whether in Python or Kotlin for Android, only handles the data capturing and transmission to the server side, making it suitable for most low-power SoCs. The storage server handles receiving and saving data. The session recorder coordinator is a stateful microservice that coordinates recording with all the connected recording devices. Last, the Airflow server, triggered at the end of a session recording, handles data post-processing pipelines for the rig. Figure 5.1 shows the entire software architecture of the data-capturing rig.

There are several difficulties involved with creating a scalable data-capturing architecture. As the amount of sensors grows, the amount of data handled by the pipeline must also increase to accommodate them. To create an architecture that is both generalizable and scalable, we noted several development tools that can be beneficial in creating a versatile pipeline.

When capturing large amounts of sensor data, it is inconvenient to store each image locally as datasets routinely take up hundreds of gigabytes of storage. Therefore, implementing a cloud storage platform is beneficial for dataset creation. Shared cloud storage platforms like AWS, Google Cloud Storage, and Azure can be easily set up to store terabytes of data reliably by paying low prices per gigabyte per month. FTP servers could also be set up to store data in dedicated storage for free if one is available. Each storage platform has well-documented libraries to allow easy data transfer from the capture pipeline into storage buckets. Additionally, Python has a library called `PyFilesystem` that manages data transfer to all common cloud storage

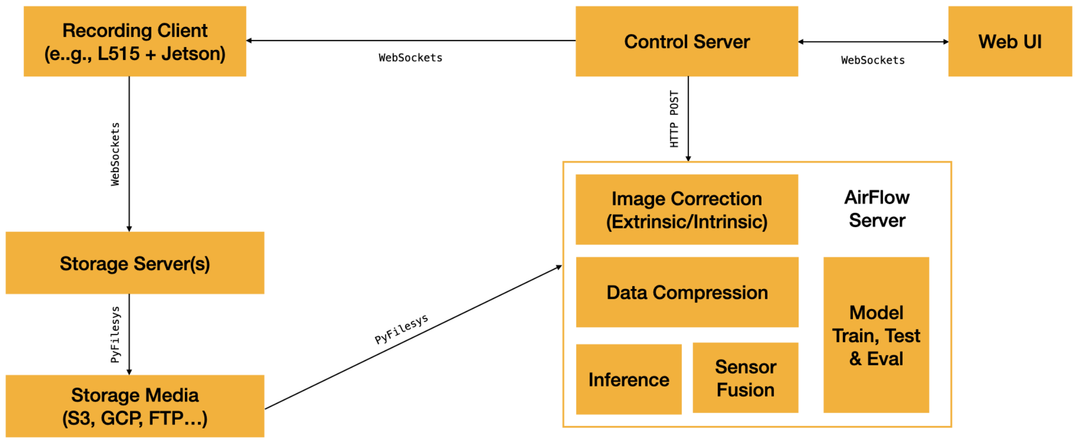


Figure 5.1: Software stack architecture.

platforms in a single library.

Apache Airflow is a platform designed to streamline and simplify data pipelines [17]. It is designed to be scalable and flexible, allowing easy manipulation of data streams before or after processing. Airflow integrates seamlessly into most common cloud storage platforms such as AWS, Google Cloud Storage, and Azure. Because of this, Airflow is a valuable tool for post-processing sensor data. Once sensor data has been stored in local or cloud storage, Airflow can post-process and calibrate the captured images to ensure the input and ground truth are aligned.

To facilitate simple deployment of the architecture, Docker can be used for containerization and standardization. Docker allows each section of the pipeline to be quickly deployed and executed while ensuring accurate installation to minimize compatibility issues. This enhances portability and facilitates scalability, as the encapsulation of each pipeline segment within Docker containers enables seamless scaling without intricate reconfiguration. Docker simplifies the orchestration and reproducibility of the pipeline architecture, enabling efficient and reliable execution across a diverse range of sensors and computational devices.

Figure 5.2 visually shows the workflow from the UI to the output. The software components communicate with each other through WebSockets or HTTP POST requests. The web User Interface (UI) (1) provides session recording information, such as the session name or destination storage, to the control server, which will relay it to the recording devices (2, 3). The devices will start recording when the "Start Recording button is pressed on the UI. The frames are saved in storage (4, 5) and then aligned through the image correction (6).

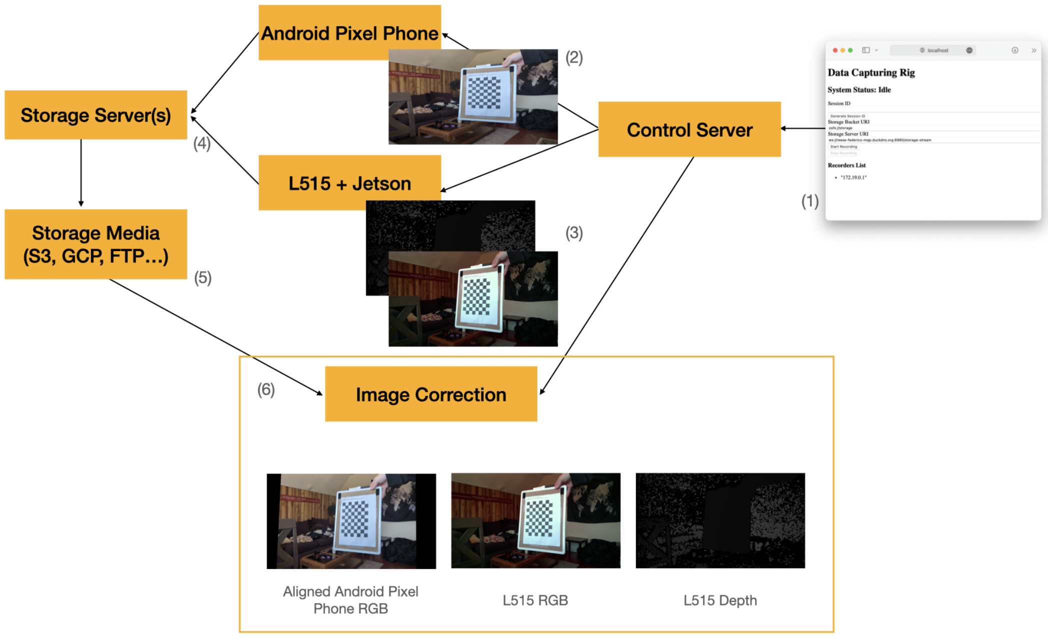


Figure 5.2: Visual example software workflow using the capturing rig.

By offloading the recording coordination, file system saving, and data processing to an external server, it is possible to minimize the computing power required for the recording client. This has additional benefits, such as lower power usage, which is necessary to allow long session recordings when using a battery pack to power the right in a mobile setting.

Two types of recording clients are supported: (1) those running the Python data-capturing script (e.g., Jetson Nano) and (2) those running the Android app. Both types get data from a sensor (e.g., a phone camera on the Android app) and transfer the data to storage while coordinating the session recording with the control server.

5.2 Control Server

The control server (GitHub) was developed using FastAPI with the standard uvicorn ASGI server. The server holds three endpoints: (1) GET request handler on "/" to provide the recording controller web user interface, (2) WebSocket handler on "/ws-recorder" to handle recorder connections (e.g., Jetson Nano), and (3) Websocket handler on "/ws-webui" to handle connections with the recording web user interface. Figure 5.3 shows the flow of events between the web UI, control server, and recording devices.

The networking architecture of the system allows client devices to connect to the server, requiring only the server to have a public IP address accessible by the clients. This removes the need to setup port forwarding for each recording device, which could lead to security vulnerabilities as well as a less streamlined user experience.

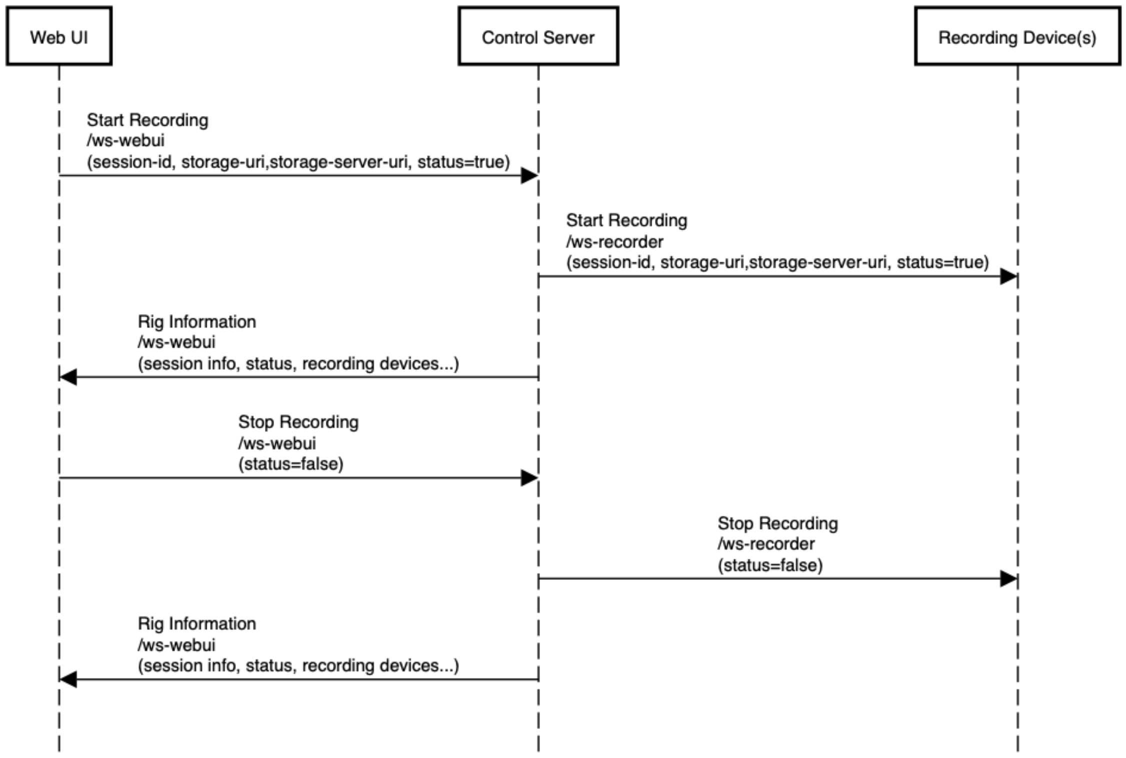
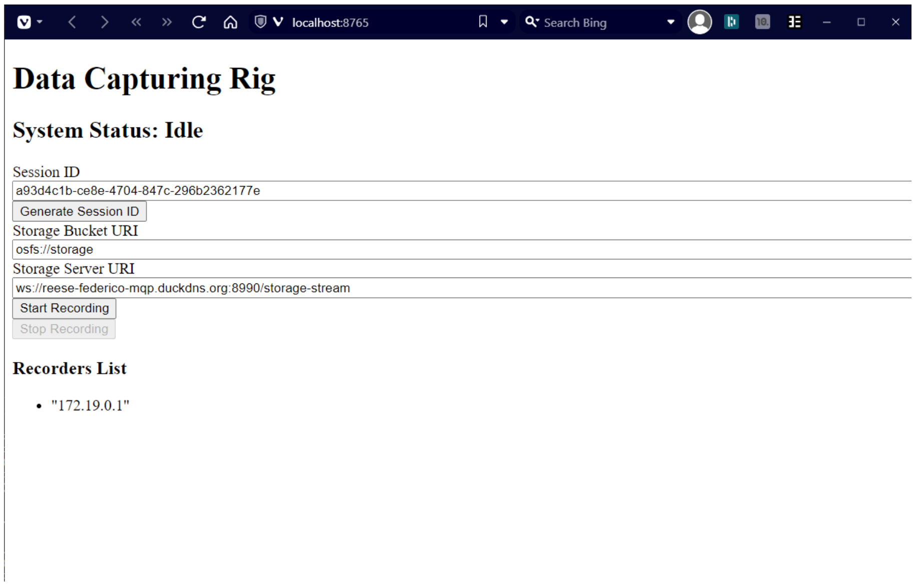


Figure 5.3: Sequence diagram of the event messages between the user interface, control server, and recording devices.

5.2.1 GET WebUI /

The GET request handler provides the recording controller web user interface. The page displays buttons to start and stop the recording and generate a v4 UUID as an identifying name for the session. Additionally, the webpage includes text fields to manually enter a recording session's unique identifier, storage server URI, and storage bucket URI. Once a connection is established with the control server, the web UI displays the currently connected recording devices. Figure 5.4 shows the design of the web UI, including all necessary fields and the list of recording devices by IP address.


localhost:8765

Search Bing

Data Capturing Rig

System Status: Idle

Session ID

a93d4c1b-ce8e-4704-847c-296b2362177e

Generate Session ID

Storage Bucket URI

osfs://storage

Storage Server URI

ws://reese-federico-mqp.duckdns.org:8990/storage-stream

Start Recording

Stop Recording

Recorders List

- "172.19.0.1"

Figure 5.4: Screenshot of the web UI used to control the suite of sensors or host devices.

5.2.2 WS /ws-webui

The "/ws-webui" WebSocket endpoint handles messages from the interface to the control server and vice versa. The interface sends messages with the following JSON payload to the control server when a recording session is started:

```
{
  "session-id": "75442486-0878-440c-9db1-a7006c25a39f",
  "storage-uri": "osfs://storage",
  "storage-server-uri": "ws://reese-federico-mqp.duckdns.org:8990
     /storage-stream",
  "status": true
}
```

When a recording session ends, the web UI sends a message with the following payload:

```
{
  "status": false
}
```

The control server provides updates about the rig status to the web UI through the "Rig Information" event. This event is triggered as an update to the web UI every time a start or stop recording message is sent. The "Rig Information" event contains the following payload:

```
{
  "recorders": [ "ip-address-of-recorder1" ],
  "session-recording": {
    "status": false,
    "storage-uri": "osfs://storage",
    "storage-server-uri": "ws://reese-federico-mqp.duckdns.org:89
       90/storage-stream",
    "session-id": ""
  }
}
```

5.2.3 WS /ws-recorder

The control server provides a "/ws-recorder" endpoint, which allows opening connections with client devices to connect to it and share system events. System events are only sent from the control server to the client and never vice versa.

An array of connected devices is updated every time a recording device connects. A device is removed from the array when it disconnects.

Every time a message is sent to the "/ws-webui" endpoint, it is directly relayed to all the registered recording devices through the "/ws-recorder" endpoint.

5.3 Storage Server

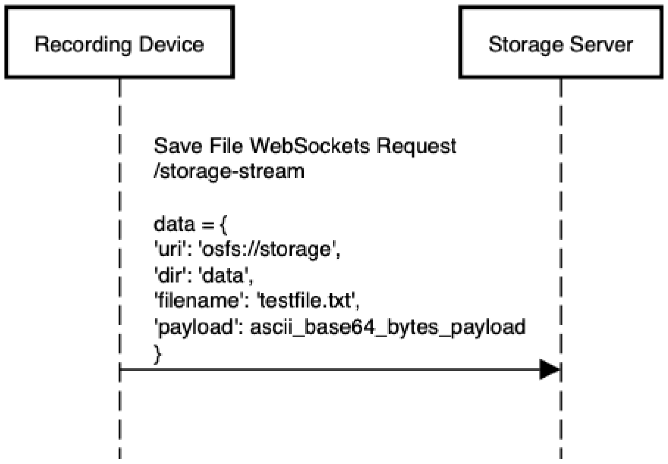
The storage server receives a base 64 encoded ASCII string of the file's bytes over a WebSocket channel and saves it to a specified storage target using the `PyFilesystem` interface. The storage server supports any file type, as the file format is directly specified in the message. Extrinsic and intrinsic data on the L515, for example, are dumped as .json files. Images could be saved as png/jpeg depending on the sensor. The storage server (GitHub) was developed using FastAPI with the standard uvicorn ASGI server. This architecture has several benefits.

First, the `PyFilesystem` library allows saving files to various storage systems, including the local filesystem, Amazon S3, Google Cloud Storage, FTP, and several more. The library allows easy integration with all of them simply by specifying the target URL of the storage media through a protocol specifier. For example, for a local filesystem, "osfs://," for FTP, "ftp://." Usernames and passwords can also be encoded within the URL to specify the authentication parameters — more about the `PyFilesystem` interface is available at PyFilesystem.

Another benefit of this architecture is the ability to offload the job of saving files, a time, network, and computationally expensive operation prone to errors (e.g., network stability) to a separate microservice. If the server hosting this microservice were overloaded, a second server could be spun up to balance the load.

Last, using WebSockets instead of traditional HTTP GET/POST requests reduces the overhead of handshakes and other standard TCP operations that would increase network latency and bandwidth usage.

The sequence diagram in Figure 5.5 shows the client-server message communication and payload required to save a file.



```
data = {
'uri': 'osfs://storage',
'dir': 'data',
'filename': 'testfile.txt',
'payload': ascii_base64_bytes_payload
}
```

Figure 5.5: Sequence diagram of the event messages between recording devices and storage server.

5.4 Python Recording Client

The Python script was designed universally to allow for any number of sensors to be recorded. It was also designed parametrically to allow specifying configuration settings by the user through a standard JSON file (Section 5.4.1). The script was separated architecturally into different layers responsible for (1) capturing, (2) storage communication, (3) sensor communication, and (4) control server communication.

The Python script was tested on a Jetson Nano running Python 3.8 with the latest release of the Intel librealsense library built from scratch from their repository (Section 5.4.2).

5.4.1 JSON Configuration

The Python script uses a JSON configuration file provided by the data capturing rig user to interpret which sensors to read data streams from. This configuration file includes an array of sensors to be read and, for each sensor, a list of streams to be read throughout a session recording. These streams can be, for example, RGB, Depth, Intrinsic, and Extrinsic data.

Below is an example of a configuration file used for the Intel RealSense L515 RGB-D camera, which is read over USB 3.0 by the Jetson Nano.

```
{
    "sensors": [
        {
          "name": "L515", "type": "IntelRealSenseL515",
          "identifier": 0,
            "streams": [
            {
            "name": "depth", "format": "png", "identifier": 0,
            "shape": {"width": 640, "height": 480, "channels": 1}
            },
            {
            "name": "rgb", "format": "png", "identifier": 1,
            "shape": {"width": 960, "height": 540, "channels": 3}
            },
            {"name": "intrinsic data", "identifier": 2, "data": "
               data"},
            {"name": "extrinsic data", "identifier": 3, "data": "
               data"}
            ]
        }
    ]
}
```

Each sensor is identified by a *name*, which will be used for storage to identify the saved data. The script uses the sensor type to know which library and handler to get data from that specific sensor. The *identifier* is a unique numerical identifier that can be used to identify the particular sensor, similarly to the *name*.

Each stream is identified by a *name*, used fo separate the data into specific folders for storage purposes.

5.4.2 Controlling the Capturing Sensors

To capture data from the Intel RealSense L515 camera, it is necessary to install the RealSense Library (librealsense). Due to using an arm64 SoC on the Jetson Nano, no pre-builts were available for librealsense. Therefore, the library was built from scratch using CMake for our platform. Python bindings were then built for librealsense to communicate with the Python 3.8 interpreter. Guides for the build steps are available in the librealsense repository documentation under the NVidia Jetson Devices section of the docs. Steps to build the Python bindings are provided under the Python Wrapper documentation.

The librealsense API allows getting frames from the L515 camera through the *self.pipeline.wait_for_frames()* function call. Frames are converted from bytes into PNG, which is the default format from the librealsense API, and sent to the storage server. The format is left unchanged to avoid introducing processing or compression latency in the pipeline.

5.5 Android Recording Client

The Android application offers a streamlined user experience, developed in Android Studio using Kotlin. At launch, it presents the user with a textbox for the URL to the control WebSocket server and a viewfinder to help with camera alignment. Once a URL is entered and submitted, the app automatically connects to the WebSocket controller and waits for commands. The app will then manage 1) connecting to the storage server, 2) ensuring synchronization with other sensors, 3) continuously capturing images, and 4) sending the data to the storage server. There is no equivalent to `PyFilesystem` for android development, therefore the app must send its data to the python storage server running `PyFilesystem` for cloud storage to be handled. Each image is saved with the recording session UUID and ISO 8601 plus millisecond date and timestamp, consistent with the Python script's file format. While recording, the app displays information as toast notifications to the user to provide the recording status or helpful debugging information if a link in the pipeline fails.

5.6 Image Correction

In multi-sensor systems, aligning images captured by different sensors is crucial for various applications such as augmented reality, stereo vision, and 3D reconstruction. This section uses homography estimation to align images from an Intel RealSense L515 depth camera and a Google Pixel Phone. The objective is to achieve pixel-perfect alignment between images from different sensors.

5.6.1 Calibration with Checkerboard

A calibration process was employed using a 7x9 checkerboard to establish correspondences between the images. We used a 7x9 checkerboard (Figure 5.6) for camera calibration with 20x20 mm squares from the Mobile Robot Programming Toolkit. OpenCV's `cv2.findChessboardCorners` function automatically detects the corners of the checkerboard in the images. The identified corners serve as the reference points for homography calculation.

```
ret, corners = cv2.findChessboardCorners(image, (9, 7))
```

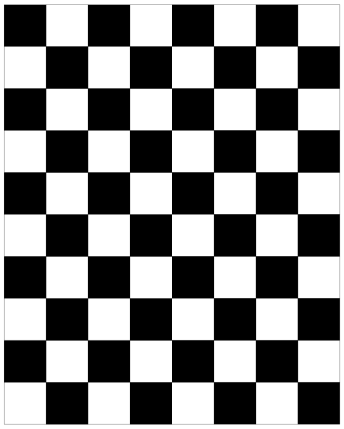


Figure 5.6: A 7x9 checkerboard is used to estimate the homography matrix to perform the alignment between different sensors.

5.6.2 Homography Calculation

Homography matrices play a fundamental role in computer vision and image processing, providing a mathematical representation of the projective transformation between two images of the same scene taken from different viewpoints. A homography matrix, often denoted as H, is a 3x3 matrix that relates the coordinates of points in one image to their corresponding points in another. Mathematically, if $\mathbf{p}$ represents a point in homogeneous coordinates in one image, and $\mathbf{p}'$ represents its corresponding point in the other image, the relationship is expressed as $\mathbf{p}' = H \cdot \mathbf{p}$. The homography matrix is determined by solving a system of linear equations using a set of corresponding points between the images. Typically, a minimum of four non-coplanar points is required to define the homography matrix uniquely. The matrix captures essential geometric transformations such as rotation, translation, scaling, and skewing, making it a powerful tool in various computer vision applications. Mathematically, the planar homography relates the transformation between two planes (up to a scale factor):

$$\begin{bmatrix} x^{'} \\ y^{'} \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The detected checkerboard corners from both devices were used to compute the homography matrix using OpenCV's `cv2.findHomography` function. The homography matrix encapsulates the geometric transformation between the two sets of points.

```
    H, status = cv2.findHomography(image_pts, template_pts)
```

5.6.3 Image Alignment and Downsampling

To ensure efficient processing and prevent the generation of artificial depth data, images were downscaled to the size of the smaller image. Upscaling could lead to inaccuracies and increased computational overhead.

5.6.4 Homography Application and Storage

The calculated homography matrix was applied to all frames captured during a session recording. The aligned frames were saved as a post-processing step using Airflow pipelines. The homography matrix was serialized and held into a Python pickle for easy retrieval and application in subsequent sessions.

```
aligned_image = cv2.warpPerspective
    (image, homography_matrix, (output_width, output_height))
```

5.6.5 Implementation Script

The homography alignment process can be found in *alignment_script.py*. This script performed the calibration and homography calculation and applied the transformation to align images captured by the L515 and Pixel Phone.

5.7 Addressing Scalability and Generalizability challenges

Scaling and universality were considered at every chain level when architecting the rig's software stack.

The storage server was introduced to offload the writing process to the file system or a cloud storage point such as an Amazon S3 or Google Cloud Storage bucket to a server, avoiding slowdowns on the recording client side.

The API with the control and storage servers was generalized to allow any network-connected device, regardless of platform or type, to connect as a recording device.

The Python recording client was also generalized to allow adding custom interfaces to stream data from sensors, allowing virtually any sensor to provide data during a session recording.

5.8 Dataset Data Format and Structure

The clients save data to the storage server by playing it in a nested folder structure identifying the recording session (e.g., v4 UUID from the web UI), the sensor name (e.g., L515, Pixel), and the stream name (e.g., depth, RGB, intrinsic, extrinsic). Each file is saved with an ISO 8601 compliant date and time timestamp with seconds expressed as floating point with four decimals.

```
storage-uri/
    session-id/
        sensor-name/
            stream-name/
                2023-11-30T13:58:02.226Z.png
```

An example using the L515 Intel RealSense and an Android Phone on OSFS storage would be:

```
osfs://storage-uri/
    eda0f965-5f41-4537-a7a0-3524ba2ded93/
        Google_Pixel_8_Pro/
            rgb_stream/
                2023-11-30T13:58:02.226Z.png
        L515/
            color_frame/
                2023-11-30T13:58:02.226Z.png
            depth_frame/
                2023-11-30T13:58:02.226Z.png
            color_intrin/
                2023-11-30T13:58:02.226Z.json
            depth_intrin/
                2023-11-30T13:58:02.226Z.json
            depth_to_color_extrin/
                2023-11-30T13:58:02.226Z.json
```

6

Testing and Experimentation

To validate the efficacy of our methodology, a series of experiments were executed to gauge the accuracy and efficiency of our data capture process. This section delves into assessing latency on our primary host devices, namely the Jetson Nano and the Pixel 8 Pro. Subsequently, we delve into an in-depth analysis of the alignment accuracy of our data, along with an evaluation of the precision of the depth-estimated images compared to our ground truth images. Furthermore, a comprehensive examination of the dataset workflow and the usability of the web-based User Interface (UI) is presented.

6.1 Latency Evaluation

In ensuring the applicability of our capture rig across diverse AR systems, achieving low latency, ideally targeting 30 frames per second (fps) video, becomes imperative. To assess our rig's proximity to this metric, we analyzed the average latency exhibited by our host devices, the Jetson Nano and the Pixel 8 Pro, each analyzed across 100 frames. This was measured by pressing "start recording" and a timer was started. Once 100 frames had come in from a sensor, it was sent the "end recording" command and time was stopped. Once our two sensors were finished sending frames, the average latency was calculated. The computed average latency per frame was 0.76 seconds for the Jetson Nano and 0.43 seconds for the Pixel 8 Pro. Although these figures fall short of the 30 fps benchmark, they represent a notable speed sufficient for data capturing. While each sensor is able to capture data at 30 fps, the data transfer pipeline acted as a bottleneck for our end-to-end latency. Further insights on latency are discussed in Section 7.1.

6.2 Evaluation of Data Processing and Inference Accuracy

To ensure precise alignment between the Pixel 8 Pro and L515 sensors within our processing pipeline, we measured the average Structural Similarity Index Measure (SSIM) score between their respective color images, which were converted to grayscale to eliminate individual color processing discrepancies. We utilized a least squares scaling factor to ensure that the scale between each image was consistent. Our dataset achieved an average SSIM of .37.

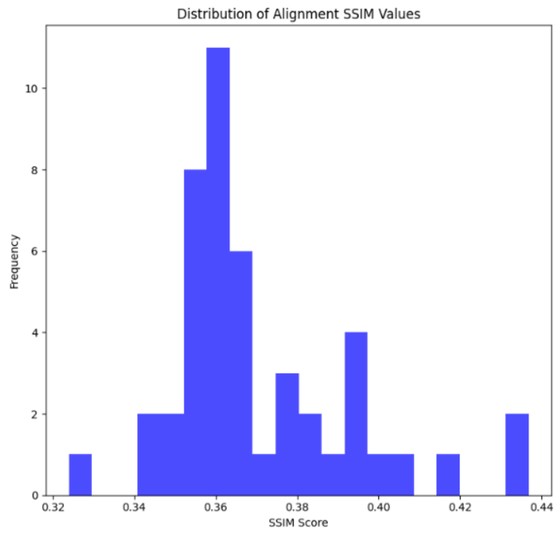


Figure 6.1: Histogram of theSSIM score between aligned Pixel 8 and L515 color images.

Figure 6.1 shows the distribution of AbsRel values across the dataset. The graph appears slightly skewed to the left with a strong peak around .37 and a trail of slightly higher scores. The average SSIM score suggests that our alignment function is under-performing. If our data alignment was perfect, the average SSIM score would be close to 1, whereas a poor score is generally considered anything between -1 and 0. Therefore, while our alignment pipeline is partially aligning the frames, it is a perfect alignment. This exposes a fundamental issue with our pipeline: using a single checkerboard image for sensor alignment resulted in insufficient spatial data to compute an accurate homography matrix.

Furthermore, our evaluation encompassed testing the accuracy of monocular depth estimation derived from our input images against ground truth data. Leveraging DPT MiDaS Large[1] for monocular depth estimation, we compared

[1]https://github.com/isl-org/MiDaS

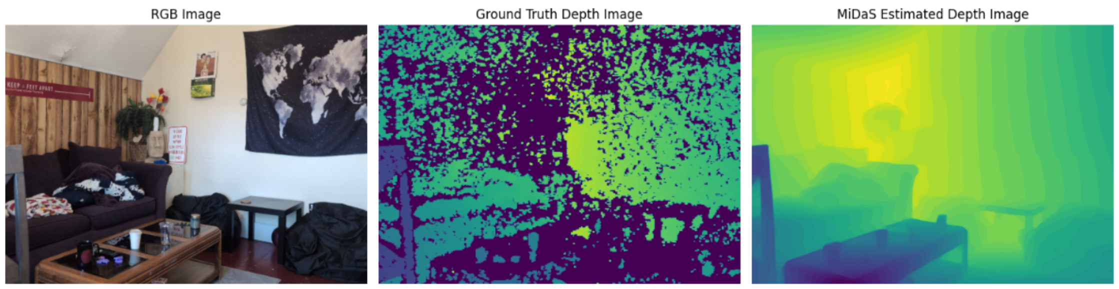


Figure 6.2: Comparison of Pixel 8 Pro color image, L515 ground truth depth image, and MiDaS estimated depth iamge.

the estimated depths with ground truth images. Figure 6.2 shows the color image, ground truth and MiDaS estimated depth images from a single image in our dataset.

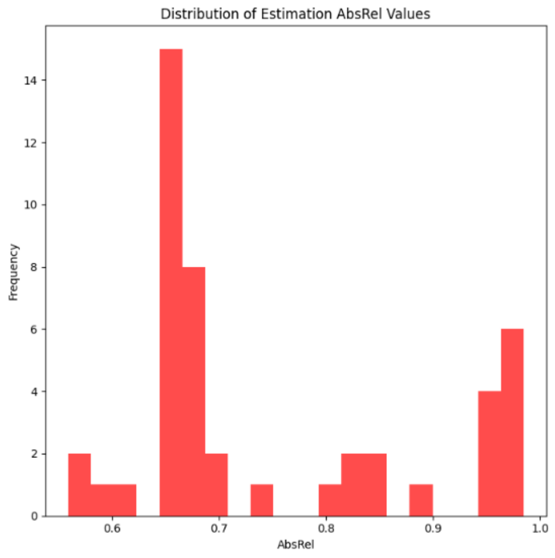
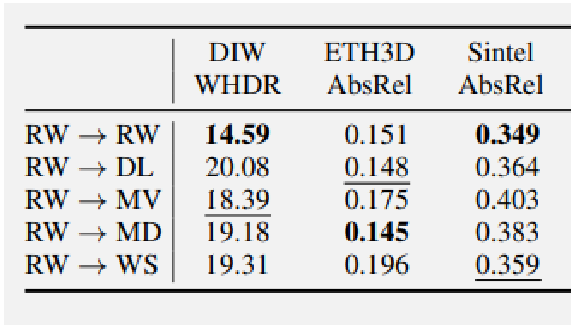


Figure 6.3: Histogram of theAbsRel between estimated and ground truth depth images.

Before calculating accuracy, we used a mask to remove any pixels from the calculation marked as 0 by the L515 sensor due to insufficient depth confidence. Moreover, we once again used least squares scaling to normalized the pairs of images. This was necessary as the MiDaS model outputs images with relative depth information without any concrete units, so we needed to scale our ground truth images accordingly.



	DIW WHDR	ETH3D AbsRel	Sintel AbsRel
RW → RW	**14.59**	0.151	**0.349**
RW → DL	20.08	<u>0.148</u>	0.364
RW → MV	<u>18.39</u>	0.175	0.403
RW → MD	19.18	**0.145**	0.383
RW → WS	19.31	0.196	<u>0.359</u>

Figure 6.4: AbsRel achieved by MiDaS on several common datasets included in [5]

The subsequent average AbsRel is .75. Figure 6.3 shows the distribution of errors across the dataset. Interestingly, both the peak accuracy and spread of outliers are much closer to 0 when compared to the alignment AbsRel. This correlation suggests that an improved dataset alignment could significantly enhance depth estimation accuracy using the Pixel's images with the L515 sensor. Columns 2 and 3 on Figure 6.4 show the AbsRel of a trained MiDaS model against 5 datasets, where the column is the dataset used for training and the row is the dataset used for testing. When compared to the accuracies achieved in [5], MiDaS performs poorly on our dataset. We suspect that the drop in performance is more related to our dataset alignment. We believe that better dataset alignment might result in an increase in accuracy from the MiDaS model.

7 Discussion

7.1 Development Challenges and Constraints

During the creation of our architecture, several challenges emerged which threatened the scalability of our pipeline and thus needed careful consideration. One prominent issue that arose was end-to-end latency concerns. Ideally, the capture rig could capture and transfer frames at a steady 30 fps. However, due to computational limitations with the Nano, we could not process frames quickly enough for this to be the case due to bottlenecks in the RGB and depth alignment and NumPy computations. These bottlenecks could be mitigated with a stronger computational device or through clever multi-threading.

Furthermore, bandwidth limitations from the client to the storage server created a bottleneck throughout the pipeline. While WebSockets enabled the rig to be mobile to capture various scenes, it came at the cost of latency. Replacing the WebSocket connection between client devices and the storage server with a more efficient and compressed data transfer protocol could alleviate this bottleneck.

Additional problems could arise in the scalability of our pipeline. While the architecture was designed from the ground up with scalability in mind, it is challenging to accommodate every possible combination of sensors and computational topologies. Future issues may arise due to compatibility issues with specific sensors or libraries. These issues must be addressed on a case-by-case basis when they arise.

While we were able to implement PyFileSystem for the Python client, there is no alternative to replace it in Kotlin. Because of this, the storage server was created to transfer data from the Android device to varying cloud storage platforms. While this maintained a generalizable structure for our architecture, it increased the complexity of the overall architecture and might cause additional bandwidth or latency concerns.

7.2 Future improvements and scalability considerations

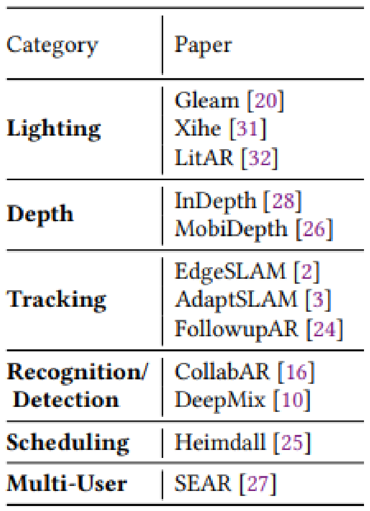
There are many possible future improvements to this project. These are some of the primary ones we have identified with our project advisor and WPI Cake Lab researchers.

First, synchronizing frame capturing is a complex task. Currently, frames are captured as quickly as possible or with a delay of some specified time (e.g., one frame per second). However, it is convenient for the rig to have a synchronization system to capture co-temporal frames on all devices. This trigger could be a hardware pulse device physically connected to the recording clients with a cable. This would reduce timing and synchronization issues and provide better time-aligned quality data.

Some cameras need to be calibrated to obtain the best quality data. Introducing a pre or postprocessing pipeline to perform camera calibration would be helpful. This task, supported by a simple user interface, would help researchers streamline the camera calibration process to a simple web user interface.

Other future improvements revolve around improving the rig's performance to enable data capturing at faster framerates. Framerates can be bottlenecked in many places, such as the capturing camera, real-time data postprocessing, bandwidth, network latency, etc. More sensors should be tested to identify and address rig performance weaknesses.

7.3 Integration with other AR tasks



Category	Paper
Lighting	Gleam [20]
	Xihe [31]
	LitAR [32]
Depth	InDepth [28]
	MobiDepth [26]
Tracking	EdgeSLAM [2]
	AdaptSLAM [3]
	FollowupAR [24]
Recognition/ Detection	CollabAR [16]
	DeepMix [10]
Scheduling	Heimdall [25]
Multi-User	SEAR [27]

Figure 7.1: A survey of recent AR systems work and their evaluation methodology as included in [4]

.

There are several different tasks of relevance in the field of AR research. Table 7.1 provides some examples of recently published papers and their respective categories of AR tasks being researched. Although our focus in this project was depth estimation, there are others such as lighting, tracking, recognition and detection. These are only some examples, but with the quickly evolving field of AR research, more AR research tasks appear every year. Future improvements to this project involve testing the support of our rig for other types of devices and sensors, such as LiDAR and radar, involved in these AR tasks. Depending on the types of sensors, some changes might be necessary, such as implementing a new transport layer in case the bandwidth of a WebSocket connection over LAN is insufficient to support the research tasks.

8

Conclusion

As the field of AR grows ever larger into a general consumer market, it is necessary for the evaluation of AR systems to be scalable and generalizable to ensure excellent results. We created a versatile data capture rig and a scalable software pipeline to contribute to the tools available for AR evaluation. We captured a test dataset to verify that our data-capturing setup was efficient and accurate. While discrepancies in our accuracy were discovered, this project lays a good foundation and methodology for designing and implementing future data capturing rig and software stack. Overall, our robust hardware and software infrastructure is poised to significantly contribute to the advancement and reliability of AR evaluation in an expanding consumer market.

Glossary

AbsRel Absolute relative difference. vi, 26–28

AR Augmented Reality. 1–3, 5, 7

ASGI Asynchronous Server Gateway Interface. 15, 18

FTP File Transfer Protocol. 13

HTTP HyperText Transfer Protocol. 19

IMU Inertial Measurement Unit. 4

LAN Local Area Network. 31

OSFS Operating System File System. 24

RGB Red Green Blue. 1, 11, 12, 29

RGB-D Red Green Blue and Depth. 1

SoC System on Chip. 13, 20

SSIM Structural Similarity Index Measure. vi, 26

UI User Interface. vi, 14–17

9

References

[1] [Online]. Available: `https://www.mckinsey.com/spContent/bespoke/tech-trends/pdfs/mckinsey-tech-trends-outlook-2022-immersive-reality.pdf?trk=public_post_comment-text`.

[2] E. E. Cranmer, M. C. tom Dieck, and P. Fountoulaki, “Exploring the value of augmented reality for tourism,” *Tourism Management Perspectives*, vol. 35, p. 100 672, 2020.

[3] Y. Guan, X. Hou, N. Wu, B. Han, and T. Han, *Deepmix: Mobility-aware, lightweight, and hybrid 3d object detection for headsets*, 2022. arXiv: `2201.08812 [cs.CV]`.

[4] A. Ganj, Y. Zhao, F. Galbiati, and T. Guo, “Toward scalable and controllable ar experimentation,” in *Proceedings of the 1st ACM Workshop on Mobile Immersive Computing, Networking, and Systems*, ser. ImmerCom ’23, Madrid, Spain: Association for Computing Machinery, 2023, pp. 237–246, ISBN: 9798400703393. DOI: `10.1145/3615452.3617941`. [Online]. Available: `https://doi.org/10.1145/3615452.3617941`.

[5] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, *Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer*, 2020. arXiv: `1907.01341 [cs.CV]`.

[6] Y. Chen, H. Inaltekin, and M. Gorlatova, *Adaptslam: Edge-assisted adaptive slam with resource constraints via uncertainty minimization*, 2023. arXiv: `2301.04620 [eess.SY]`.

[7] I. Vasiljevic *et al.*, “DIODE: A Dense Indoor and Outdoor DEpth Dataset,” *CoRR*, vol. abs/1908.00463, 2019. [Online]. Available: `http://arxiv.org/abs/1908.00463`.

[8] P. K. Nathan Silberman Derek Hoiem and R. Fergus, “Indoor segmentation and support inference from rgbd images,” in *ECCV*, 2012.

[9] G. Baruch *et al.*, *Arkitscenes: A diverse real-world dataset for 3d indoor scene understanding using mobile rgb-d data*, 2022. arXiv: `2111.08897 [cs.CV]`.

[10] M. Broxton *et al.*, "Immersive light field video with a layered mesh representation," *ACM Trans. Graph.*, vol. 39, no. 4, Aug. 2020, ISSN: 0730-0301. DOI: `10.1145/3386569.3392485`. [Online]. Available: `https://doi.org/10.1145/3386569.3392485`.

[11] J. Zhang *et al.*, "Mobidepth: Real-time depth estimation using on-device dual cameras," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022, pp. 528–541.

[12] J. Uhrig, N. Schneider, L. Schneider, U. Franke, T. Brox, and A. Geiger, "Sparsity invariant cnns," in *International Conference on 3D Vision (3DV)*, 2017.

[13] *Intel realsense lidar camera l515*, Rev. 003, Intel, Jan. 2021.

[14] A. Janoch *et al.*, "A category-level 3-d object dataset: Putting the kinect to work," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2011, pp. 1168–1174. DOI: `10.1109/ICCVW.2011.6130382`.

[15] H. Sarbolandi, D. Lefloch, and A. Kolb, "Kinect range sensing: Structured-light versus time-of-flight kinect," *Computer vision and image understanding*, vol. 139, pp. 1–20, 2015.

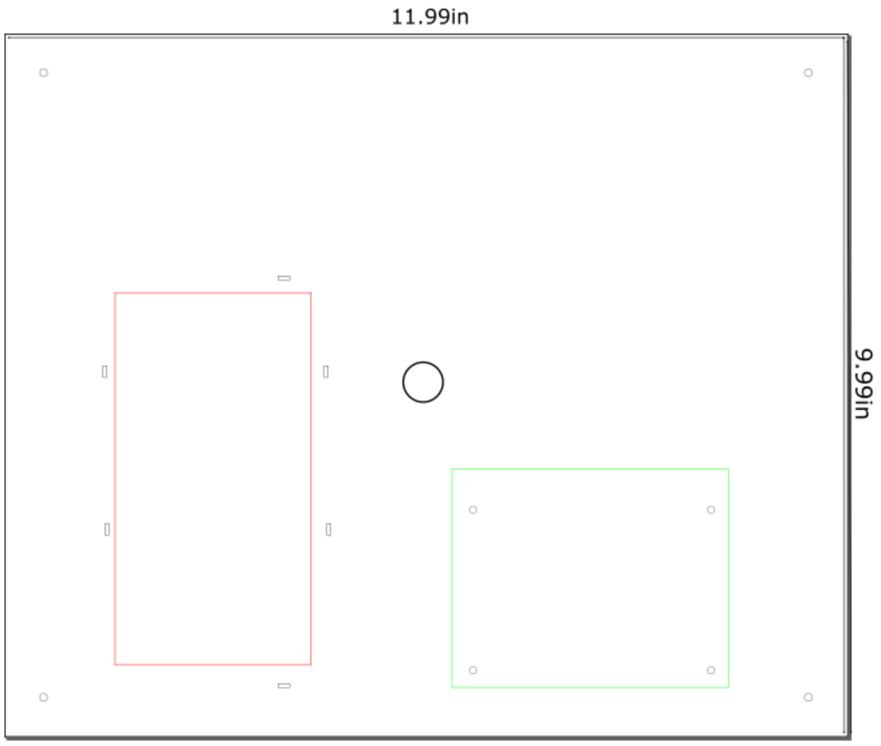
[16] *Intel realsense product family d400 series*, Rev. 017, Intel, Sep. 2023.

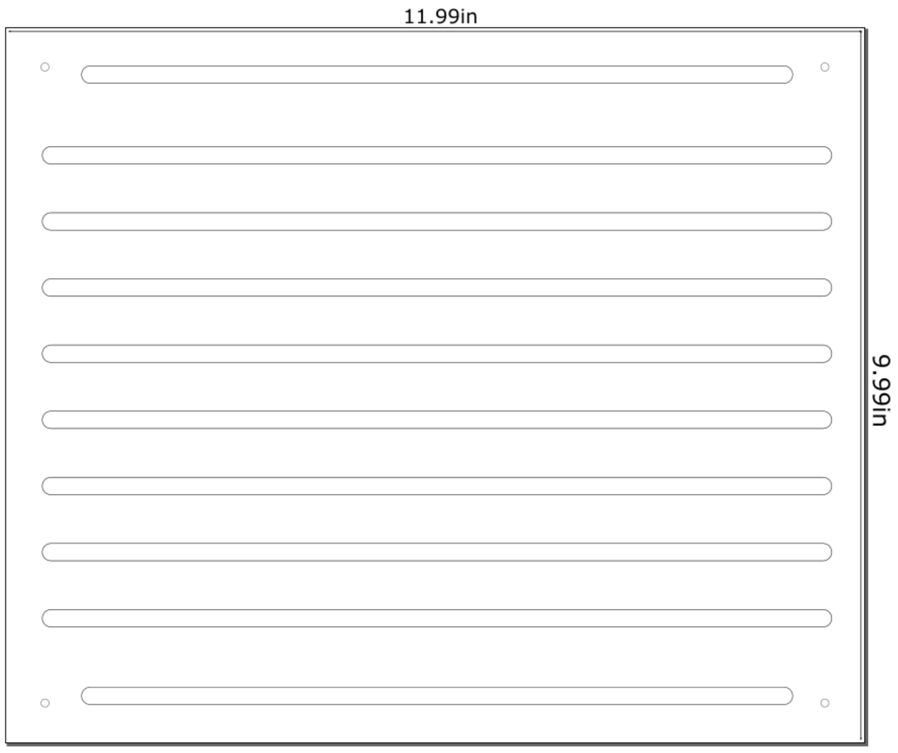
[17] S. Shukla, "Developing pragmatic data pipelines using apache airflow on google cloud platform," *International Journal of Computer Sciences and Engineering*, vol. 10, no. 8, pp. 1–8, 2022.

Appendix A

Laser Cut Design Considerations of the Rig

Below are the designs used to laser-cut the rig. The overall dimensions are 12" by 10". On the control layer(top), we added pilot holes used to mount the NVIDIA Jetson Nano (green bounding box) and slits to wrap zip ties around the power bank used to power the Jetson Nano(red bounding box). In the center, we cut a hole to mount a ¼" tripod adapter to allow the whole rig to be mounted to a tripod. On the sensor layer (bottom), we made ten horizontal cuts across the long side to allow for mounting a suite of sensors in various orientations. On the four corners of both layers, we cut 3mm holes to screw the stand-offs through, allowing the whole rig to be assembled.


11.99in
9.99in


11.99in
9.99in