Remote-Controlled Mixed Reality Driving Experience

by

Greg V. Klimov
Samuel L. Kwok
Cole R. Manning
Mason V. Powell
Sam P. Rowe
Adam Yang

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
March 2023

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.
1

Acknowledgements

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

We also want to thank Ashkan Ganj for his assistance with the project’s computer vision components and Maya Angeles for her help with the hardware design.

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

Virtual reality (VR) headsets offer a unique means of immersive entertainment. However, traditional VR headsets typically only operate within a completely virtual environment. Our team proposes an approach that combines the VR experience with the classic remote-controlled car to create an augmented reality (AR) experience, where users can leverage the field of view, controls, and software of their device while interacting with the real world. We modified an existing kit to house a new, 360-degree camera that would allow the user an unrestricted field of vision, developed a pipeline that could stream the camera’s output directly to a VR headset, and tested computer vision models for object recognition. Our project offers a feasible way for future designers to make use of these features to create more complex and engaging activities for users to enjoy.
Authorship

Cole Manning, Mason Powell, and Sam Rowe handled all web server-related affairs, including both implementation and write-up. Each member of the web server team also contributed to end-to-end evaluation and figure creation.

Greg Klimov and Samuel Kwok headed the Unity team and were responsible for all development and writing in that area. Both also contributed to the User Stories as well as end-to-end evaluation.

Adam Yang led and managed the computer vision components of the project, as well as all modifications made to the original PiCar-X design. He wrote the vision-related sections of the report, as well as background entries.

All members collaborated to write the problem statement, abstract, and conclusion sections of the report.
Contents

1 Acknowledgements i

2 Introduction 1

3 Background 3
   3.1 PiCar-X and Raspberry Pi ............................................... 3
   3.2 Virtual and Augmented Vision .......................................... 4
      3.2.1 Visual Passthrough .................................................. 4
      3.2.2 Object Detection and Tracking ..................................... 5
   3.3 VR App Development Framework ........................................ 5
   3.4 3D Asset Creation ...................................................... 6
   3.5 Client and Server Communication ...................................... 6
   3.6 Evaluation Tools ....................................................... 7

4 Problem Statement 8

5 Design 9
   5.1 PiCar-X ........................................................................... 11
   5.2 Unity .............................................................................. 12
   5.3 Web Server ..................................................................... 13

6 Implementation 15
   6.1 Hardware Assembly .......................................................... 15
   6.2 PiCar-X .......................................................................... 16
      6.2.1 360 Degree Camera Resolution and FPS .......................... 17
      6.2.2 HDMI Adapter ............................................................ 17
      6.2.3 3D-Printed Camera Mount Platform ............................... 17
      6.2.4 Raspberry Pi Software Configuration ............................. 18
   6.3 Web Server ..................................................................... 18
      6.3.1 Video Stream Codec ..................................................... 19
      6.3.2 Input Handling Via WebSockets ..................................... 19
   6.4 VR Application ................................................................ 20
      6.4.1 RC Car Control Schemes ............................................. 20
      6.4.2 Connecting Unity to WebSocket API ............................... 21
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The original, assembled PiCar-X kit</td>
<td>4</td>
</tr>
<tr>
<td>3.2</td>
<td>Software stack of all parts of the original PiCar-X kit used in this project</td>
<td>4</td>
</tr>
<tr>
<td>5.1</td>
<td>Software Design Diagram</td>
<td>9</td>
</tr>
<tr>
<td>5.2</td>
<td>Vision Pipeline Design Diagram</td>
<td>10</td>
</tr>
<tr>
<td>5.3</td>
<td>Intended end goal of the combined Unity and computer vision pipelines</td>
<td>11</td>
</tr>
<tr>
<td>5.4</td>
<td>Side Profile of Modified PiCar-X</td>
<td>12</td>
</tr>
<tr>
<td>6.1</td>
<td>Layout of the project’s hardware, showing the headset [13], controllers [14], and PiCar-X</td>
<td>16</td>
</tr>
<tr>
<td>6.2</td>
<td>3D design of the trapezoidal platform for supporting the Pixpro 360 camera</td>
<td>18</td>
</tr>
<tr>
<td>6.3</td>
<td>Our go-kart design</td>
<td>22</td>
</tr>
<tr>
<td>6.4</td>
<td>The Unity project’s asset tree and the location of all of its associated C# scripts</td>
<td>26</td>
</tr>
<tr>
<td>6.5</td>
<td>This figure depicts the Unity app’s Assemblies, their organization, and their dependencies</td>
<td>27</td>
</tr>
<tr>
<td>6.6</td>
<td>Four puppies detected as “dog” class [23]</td>
<td>28</td>
</tr>
<tr>
<td>6.7</td>
<td>Object detection of objects throughout a given video file [24]</td>
<td>28</td>
</tr>
<tr>
<td>7.1</td>
<td>Boxplot of response packet round trip time</td>
<td>31</td>
</tr>
<tr>
<td>7.2</td>
<td>Real-time delay between actual image and image displayed to the user</td>
<td>32</td>
</tr>
<tr>
<td>7.3</td>
<td>Comparison between sources of delay in the app and average human reaction times of 250 and 384 milliseconds [27]</td>
<td>33</td>
</tr>
<tr>
<td>7.4</td>
<td>Boxplot of the delay between receiving each frame from the camera stream</td>
<td>34</td>
</tr>
<tr>
<td>7.5</td>
<td>Marginal FPS values of the end-to-end application, outliers (FPS &gt; 187) removed</td>
<td>35</td>
</tr>
<tr>
<td>7.6</td>
<td>Comparison of end-to-end FPS values and the estimated human perception speed of 30-60 FPS [29]</td>
<td>36</td>
</tr>
<tr>
<td>7.7</td>
<td>Delays between sent commands and responses from the PiCar-X</td>
<td>38</td>
</tr>
</tbody>
</table>
7.8  Numerical instability arises from analyzing multi-class results.  . . . . 41

8.1  Misclassification and bounding box offset issue on the car wheel.  . . . 44
Introduction

The field of virtual reality (VR) and augmented reality (AR) has been a popular area of development in the past decade, with large companies such as Meta going through entire re-branding processes to focus on the exploration of extended reality (XR) experiences. However, the development of commercial products in the field of extended reality entertainment has not grown at the same pace, both due to the prohibitively high costs of headsets as well as a lack of compelling use cases. Likewise, the combination of physical devices (e.g., smartphones, wearable tech) alongside XR technologies has received little attention in recent years. The challenge of combining VR and the real-world environment in a way that consumers find appealing and worthwhile remains unsolved today.

Remote-controlled (RC) vehicles have dominated consumer toy markets since their introduction in 1971, with a focus on younger audiences [1]. Dedicated controllers allow users to pilot these vehicles from a distance, mimicking the act of traditional driving or racing. The general market of remote-controlled vehicles has remained relevant due to long-lasting interest in the field, with some users honing their driving abilities to compete against others. In contrast to their original audience, in recent decades, we have seen a market shift towards older audiences, with the introduction of more practical usage (due in part to the falling costs of production) in industrial, scientific, and military fields [2]. This newer generation of remote vehicles gets much of their practicality from the addition of sensors that allow for remote data collection. For example, the addition of a camera, such as in the case of consumer photographic drones, scientific probes for use in hazardous environments, and in military unmanned vehicles (UAVs/UGVs), has proven to be especially powerful[3].

Many notable components powering XR are based on computer vision and artificial intelligence advances. The field of computer vision has recently surged in popularity due to advancements in AI, object detection, recognition, and tracking models, along with improved datasets, and more powerful GPUs. In fact, recent research into AR-enhanced medical imaging has helped medical professionals scan and visualize many organs and body parts for diagnostic purposes[4]. Within these
applications, AR can enhance the user’s environment through a wide selection of filters, object recognition, and the addition of virtual assets. Consequently, computer vision and AR can provide a means to add obstacles to an area for entertainment or enhance the user’s awareness of their surroundings when attention to detail is critical. Further research into computer-vision-powered AR may lead to innovative and compelling solutions for many more applications.

This project aims to combine both AR and remote-controlled cars while effectively leveraging computer vision libraries to create a seamless, intuitive, and entertaining experience. Our end goal is to allow the user to remotely pilot a miniature vehicle while wearing a VR headset and seeing the world from the car’s perspective. To that end, this project contributes the following:

1. Modifies an existing RC car kit to be compatible with a 360-degree camera.
2. Offers a means of streaming the camera’s live feed to a client.
3. Offers a means to view the streamed feed within a VR headset and to control the car with its associated controllers.
4. Evaluates a separate computer vision model that can potentially be integrated into the headset in the future.

As a whole, this project is intended to act as a framework for future development in both hardware and software. We have created both a means to view and control the car and tested a vision model that can be integrated with the rest of the pipeline in the future. The current design for the car is a modification of an existing product, which offers extensive opportunities to customize the layout of all integrated parts. Our pipeline is modular, which will enable future teams to swap between different means of sending, receiving, or processing the video stream as desired.
3

Background

3.1 PiCar-X and Raspberry PI

The PiCar-X, as depicted in Figure 3.1, is an add-on kit for the Raspberry Pi comprised of a car body along with a hardware-attached-on-top (HAT) that interfaces with the Pi’s GPIO pins. The kit is manufactured by Sunfounder, who also maintains the SDK, which is hosted on a public Git repository [5]. A Raspberry Pi acts as the control hub for the car. A Raspberry Pi, or RPI, is a single-board computer made by the Raspberry Pi Foundation. An RPI works alongside Sunfounder’s Robot HAT, a multifunctional expansion board that provides a standard to connect the Pi’s 40 general-purpose input and output pins to add-ons that provide additional functionality such as motors and sensors. Sunfounder’s Robot HAT board comes equipped with an MCU to extend the pulse width module output and analog to digital converter input for the Pi and additional modules. The PiCar-X functions are implemented in Python, with the existing stack depicted by Figure 3.2. Although there exists code already within the PiCar-X’s code that focuses on object recognition with OpenCV’s computer vision software and Google’s TensorFlow, it was unused due to the team focusing on processing outside of the Raspberry Pi.
3.2 Virtual and Augmented Vision

Virtual reality (VR) headsets offer an immersive simulated environment for users to experience at the expense of preventing them from observing their physical surroundings. A lack of visual information regarding the wearer’s surroundings can result in damaging objects and the environment around the wearer. To resolve this issue, external/environment-facing cameras can provide the wearer with much-needed real-time information about their surroundings without the need to constantly remove the headset. For example, the Meta Quest 2, the headset used in this project, comes with embedded cameras that allow the user to see their surroundings to a limited degree when another application is not running. Meanwhile, augmented reality (AR), deals with real-time video of the real world, but with some form of digital alteration or enhancement. For this project, algorithms that deal with processing external camera feed for the wearer are considered to be visual passthrough algorithms.

3.2.1 Visual Passthrough

One type of visual passthrough algorithm that was considered is scene reconstruction. Researchers at Meta placed external cameras on the outside of the headset and developed these algorithms to create a digital reconstruction of the view outside
the headset [6]. The algorithm consists of four primary steps: stereoscopic imaging, density mapping, map smoothing, and projection. The most important visual passthrough algorithm that this project uses is streaming a live camera feed from the external cameras on the RC car to the headset, which will be referred to as RC visual passthrough. The wearer of the headset will obtain a first-person perspective of the RC car that they are controlling. Similar to first-person car-racing video games, the wearer would be provided an interior view of the RC car as if they were driving the car themselves. Unlike the scene-reconstruction algorithm discussed above, the cameras on the RC car are not directly connected to the headset. Instead, the RC camera feed will be captured by an external camera, then the feed will be sent to the user so they may view the physical surroundings of the car. The added benefit of this is the ability to potentially bypass the aforementioned obstacles brought on by directly connecting to cameras mounted on a VR headset.

3.2.2 Object Detection and Tracking

The RC car may need to handle object detection and tracking itself to reduce the potential delay between the image being fed to a computer vision model and the result. One framework that the team will utilize is Unity’s built-in API designed for basic object detection and tracking [7]. However, more complex computer vision tasks such as video analysis, image processing, and object detection should be achieved using other open-source resources such as OpenCV. OpenCV is an open-source software library dedicated to real-time computer vision and machine learning tasks [8]. It can be integrated into the web server to process the live-camera feed and perform computer vision algorithms before sending the results to Unity.

3.3 VR App Development Framework

Unity is a free, popular cross-platform game engine and game development suite that supports a huge number of platforms and uses, including XR app development on the Meta Quest 2, the VR headset being used for this project. It is one of the most commonly used and most accessible engines for new users, providing them with both the project templates and the documentation necessary to complete any project.

Our team had initially considered Unity as a viable front-end platform because members of our group already had some Unity and C# experience, but after investigating the use cases, supported platforms, and available online resources, our group quickly realized that Unity should greatly simplify the implementation of the final product. First, Unity is easy to set up and work with as it is free and provides users with ample project templates. Because it is free, Unity is very popular with game developers and hobbyists alike, who have as a result created a plethora of online resources to help others develop their projects. Unity tutorials and guides are very
easy to find and cover nearly every imaginable topic, as do online question forums like StackExchange that feature Unity-related questions. Moreover, Unity’s base capabilities, especially when combined with both Unity and third-party plugins, cover all of the group’s needs: Meta Quest 2 support, 3D rendering, camera inputs (including from an online location), custom scripts, etc. Unity also features an asset store where developers can download or add their own custom assets, plugins, and third-party libraries, known as packages.

However, Unity is not the only framework that offers VR capabilities like those our group needed. WebXR is another free XR Development platform that is compatible with a wide range of devices including the Meta Quest 2, partly because it is designed to run directly in the internet browser. And while its functionality or the functionality of other free XR game engines like the Unreal Engine may also cover the team’s needs, Unity’s popularity and the extent of its community support resources like the Asset Store, online tutorials, and online forums both helped our team to choose Unity and ultimately to complete the project.

For these reasons, our team chose Unity as our main front-end platform, and as such, Unity and the successes and challenges associated with it play a very important role in this report.

### 3.4 3D Asset Creation

Our end goal is to create a framework for an augmented reality experience. To that end, it is necessary to have some digital assets, such as a model of a car, to overlay on top of the real-life view. For creating assets, the team opted to primarily use Blender, a free and open-source graphics software tool for modeling and animation. This will allow us to avoid relying on models on the Unity Asset Store, which are often too expensive to justify. Some members of the team had prior experience working with Blender, but the main reason we decided to use it over an alternative (such as Maya) was the ease of use; despite Maya’s better features on a professional level, our use case won’t be in-depth enough to warrant their use. Blender still offers a wide variety of professional-grade tools and features, considering the fact that it is free, and there is no shortage of easily accessible support. Unity also supports importing native Blender files as assets, allowing for a smooth workflow between the two. This makes our usage of models within Unity and therefore within the image presented to the user within the VR headset, much simpler.

### 3.5 Client and Server Communication

The WebSocket API is a protocol primarily utilized in the client and server communication channels. Websocket is bidirectional which allows communication to happen to-and-from the client and server.[9] This enables the user’s browser and a server to start an interactive two-way communication session. This allows for
low overhead compared to HTTP polling. This is achieved by offering a standard method for the server to transmit data to the client without the client first requesting it. Enabling messages to be transferred back and forth while maintaining the connection. This gives the ability to not poll the server for a response when sending messages to a server and receiving event-driven responses.

An alternative solution to real-time communication between a client and server is Web Real-Time Communication (WebRTC). This protocol is highly compatible with many different devices and includes built-in means of connecting to a user’s media such as a camera. Being designed for audio and video content, WebRTC offers extremely robust peer-to-peer communication for browser applications [10].

3.6 Evaluation Tools

Wireshark is an open-source, live packet capture tool used for in-depth analysis of network activity [11]. This offers a convenient, reliable way to detect if a given device is streaming, how long it took for individual packets to arrive at their destination, and if the contents of those packets were the intended payload. In the scope of this project, Wireshark acts as the primary means of ensuring all streaming and general-purpose networking functionalities were properly executed. Chrome DevTools is a set of web developer tools built directly into the Google Chrome browser that allows users to edit and debug websites within their browser [12]. This includes the analysis and recording of the contents of any HTTP messages received by a client. In this project, the DevTools allowed us to record the timestamps of a series of inputs and responses produced from a computer’s browser that was connected to the car’s Raspberry Pi.
Problem Statement

This MQP project aims to integrate and enhance the vision of a virtual reality user with the real world by combining the field of view of an external 360-degree camera with the immersion of a VR headset. This type of mixed reality (MR) has seen little development in the realm of RC cars. Most commonly, remotely-controlled vehicles use a standard 2D screen to display their feed, which limits a driver’s awareness of their environment. A successful unification of a virtual environment and the physical world would provide users of remote-controlled devices with a more thorough understanding of their surroundings as it would no longer restrict the user’s field of view to that of a standard camera as seen through a monitoring device. Through the emulation of a life-like driving experience, the user will be able to control the remote device in a more natural and intuitive fashion, lessening the need for product or application-specific training.

To accomplish this goal, the camera feed from an RC car must be transmitted to a remote user wearing a VR headset, the feed must be altered to include the mixed reality elements desired, and then it must be displayed to the user. The creation of these digital elements, such as bounding boxes or 3D models inserted into the user’s field of view, should be handled in a way that does not impede the display of the unmodified camera feed. In the case of object recognition, the efficacy of any used models must be confirmed. Furthermore, the video feed must be capable of being successfully transmitted over WiFi, introducing potential concerns of security and latency that need to be properly addressed for an adequate real-world implementation of this technology. There must also be an adequate adaptation of the image from the external camera(s) overlain with computer-generated assets while maintaining a consistent, and comfortable experience for the end user. To that end, the latency of the video feed must be tested with the car at varying distances from the user as well as testing the general user experience.
As shown in Figure 5.1 this project aims to use Raspberry Pi and a 360-degree camera attached to the PiCar-X to run a web server that can stream video to a connected client. Our remote client is the VR headset, which itself hosts a Unity application that receives and displays the video. The headset is connected to its
associated controllers which send input to the Unity app to translate the user’s inputs into a format that can be understood by the car. These inputs are then sent to the web server to be executed by the PiCar-X’s movement API, thus enabling remote control of the car.

![Vision Pipeline Design Diagram](image)

Our team also developed a separate project described in Figure 5.2 that was developed concurrently with the car and Unity stream described above. Due to the difficulty of running an object recognition model within Unity, we decided it was best to work on each separately with the goal of combining the two in the future. The vision model instead takes a pre-recorded video, processes it, the returns any recognized objects as well as bounding boxes and confidence levels to describe them. This can then be drawn onto the feed to highlight which objects were detected.
Figure 5.3 depicts a mock-up of the final product. That is, the view of the 360-degree camera as seen from the connected headset, with objects recognized by the vision model being highlighted in real-time.

5.1 PiCar-X

As mentioned in 3.1, the PiCar-X is an RC car designed by Sunfounder that can be controlled through an open-source web server. Our team replaced the original camera with one that is capable of 360-degree viewing. That setup enables the capture of higher-quality footage as well as provides an uninhibited view of the environment around the car, with the end goal of creating a smoother experience for the headset wearer. The integrated Raspberry Pi manages the code for controlling the motors and functions as a means of communication to the unity app via the hosted web server and Wi-Fi. That connection allows remotely delivered commands to be given to the car in real-time, in addition to providing a live video stream of the camera feed. By integrating the two, we aim to allow a user wearing a VR headset to control the car via their controllers while seeing through the remote camera.
The associated driver code for the PiCar-X had the following commands available:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_motor_speed(self, motor, speed)</td>
<td>Sets a specified motor to the specified speed</td>
</tr>
<tr>
<td>set_dir_servo_angle(self, value)</td>
<td>Sets the direction servo to the given angle</td>
</tr>
<tr>
<td>backward(self, speed)</td>
<td>Makes the car drive backward at the given speed and at the current angle</td>
</tr>
<tr>
<td>forward(self, speed)</td>
<td>Makes the car drive forward at the given speed and at the current angle</td>
</tr>
<tr>
<td>stop(self)</td>
<td>Sets the motor speeds to 0</td>
</tr>
</tbody>
</table>

### 5.2 Unity

Having chosen Unity to create the front-end VR headset app, the team now faced the challenge of planning exactly how that app would interact with the end user and
with all the other project components. The app receives the processed video stream and shows it to the wearer of the VR headset, who then reacts by interacting with the headset controllers, the input from which, the headset sends back to the car. The app must enable the end user to not only seamlessly change the driving angle or accelerate and decelerate with minimal input delay, but it must also immerse the user into the environment of the car.

To do this, our team began by downloading a Unity VR template project and configuring the skybox material to show a new texture that displayed a 360-degree .mp4 video file by updating the texture with each frame of the video. This made it so the user could look around in their virtual space and be completely surrounded by the 360-degree video. The team then enabled the skybox texture to read data from the web server, and with the addition of a car asset from Unity’s asset store, as well as one created in-house, this enabled the app to immerse users in the same space as the car.

5.3 Web Server

Our web server serves as the main hub for our project, being the medium of communication between the user and the car. WebSockets act as the means of establishing that connection due to their ability to maintain a connection and transmit data without being prompted by the client. This allows data to be moved simultaneously between our car and the client. Keeping the web server on the car serves to keep latency between command receipt and execution to a minimum while leveraging the processing ability of the car’s onboard Raspberry Pi.

In order to provide as much flexibility as possible, the web server operates through two distinct WebSockets that run simultaneously. The first is dedicated to handling commands that are meant to control the PiCar-X, taking in any outside input while replying with information about what was received. The second provides any connected clients with the camera stream and requires no inputs otherwise. The server is compatible with any device capable of connecting to a WebSocket and can handle several clients at once. This is most convenient for the video stream, as that enables any number of users to view the car’s perspective through a web browser.

The video stream from the 360-degree camera attached to the car will be constantly sent to the web server. At that point, the server will process the video stream such that it can be easily transmitted to the VR headset. There are two primary problems we believe the server’s processing must overcome for a comfortable viewing experience, as outlined in [6]. The first, as outlined in the paper, is plausible parallax, or the accurate displacement of objects when viewed at an angle. This is necessary for reducing motion sickness in the user. To that end, we must also keep latency as low as possible by minimizing the processing time of the stream. Excess lag and stuttering, especially when the camera is moving, will further induce discomfort and disorientation for the user. Another aspect brought up by the paper
is the importance of maintaining “Haptic Trust”, or the perception that the user could reach out and touch the environment around them at any point. Accurate mapping of the camera feed to the headset, as well as the quality of the camera itself, is required to maintain this perception. While the user will not be able to physically interact with the area around the car itself, maintaining haptic trust is still important for general comfort and navigation.

Beyond the two peripherals, the web-based video stream allows multiple clients to see the video feed by connecting to the appropriate port via their web browser, assuming they are connected to the same network or that the network is configured for port forwarding. This proved to be valuable both for testing purposes as well as general usage.
6

Implementation

6.1 Hardware Assembly

Our project requires multiple components to meet our goal of a remote-controlled, augmented reality experience. The VR headset is a necessity because it is the interface the user will be using to connect to the rest of our pipeline. We chose the Meta Quest 2 because it offers solid computing power and compatibility with Unity, which is a popular means of using digital assets. The controllers bundled with the headset are also commonly used for video games with wildly different control schemes, meaning they would be highly adaptable to our needs. The last piece of hardware, the PiCar-X, functions as a remote platform that is already designed to be remotely controlled. It is meant to be managed by a Raspberry Pi that can wirelessly receive and execute any commands. The original design of the car was modified to better suit our needs, as will be described in Section 6.2.
Figure 6.1: Layout of the project’s hardware, showing the headset [13], controllers [14], and PiCar-X.

The layout shown in Figure 6.1 is the arrangement of this project’s hardware. The PiCar-X/camera streams to the headset, which in turn relays commands sent from the controllers back to the car.

6.2 PiCar-X

In order to implement VR image capabilities, we had to modify the original design of the PiCar-X. The original design had a singular camera mounted at the car’s head that was capable of rotating for different views, but this was not suitable for our intended goal of giving the driver an uninhibited field of view. Our team chose to use the Kodak Pixpro Orbit 360-degree VR Camera, as it was capable of producing a live stream, panoramic view of the entire area around the car. Given the view range of the new camera, we decided to remove the redundant original camera at the head of the car, instead opting to mount the replacement at an elevated position closer to the back using a 3D-printed platform. This was done to reduce any obstacles that may obstruct the new camera’s vision, as well as to better balance its heavier weight on the car. Furthermore, due to the comparative bulkiness and weight of the
new camera, it was necessary to design a platform it could safely be mounted upon.

6.2.1 360 Degree Camera Resolution and FPS

The Kodak Pixpro Orbit 360-degree camera offers a variety of different resolutions and frame rates, all of which have a major bearing on the potential stream quality. For videos, we had the option of recording a 3840x2160 at 24 FPS, 3840x1920 at 15 FPS, and 1920x960 at 30 FPS. Ultimately, our team decided to prioritize maximizing the number of frames per second in order to make the stream as smooth as possible, so we chose the last setting for our project. Furthermore, we believe the smaller resolution would reduce the size of our streamed data, thus lowering the stream’s bandwidth requirements.

6.2.2 HDMI Adapter

Unlike the original attached camera, the Kodak Pixpro was not an official Pi Camera module, so our team also had to integrate an adapter to connect it to the Raspberry Pi. We chose the Geekworm HDMI to CSI-2 module because its board was compatible with the Raspberry Pi and able to convert HDMI video signal without any major changes. Notable, the board did induce its own limit on the video quality, as it is only able to transmit up to 1080p at 25 FPS. As mentioned in 6.2.1, we had already chosen to set the camera to stream around this resolution, so we did not expect this to introduce any major problems.

6.2.3 3D-Printed Camera Mount Platform

The platform had a simple design that aimed to secure the Pixpro 360 camera to the RC car. The team brainstormed ideas for what the platform would look like and how the platform would be secured to the car and the camera. After identifying four screw holes on the RC car to secure the platform, we created a preliminary design and sketch of the platform. The design consisted of a trapezoidal shape with four screw holes near the four vertices. The team chose the trapezoidal design due to it being the simplest shape that overlaid the four screw-hole positions. Furthermore, the thickness of the platform was robust enough to handle the mostly vertical forces exerted by the 360 camera. Thus, the team concluded that the effects of horizontal forces on the platform were sufficiently negligible.

Additionally, in order to be able to secure the camera to the platform, a fifth hole was designed to be at the center of the trapezoidal platform. Next, our team measured the distances and sizes of the platform screws and camera screws to incorporate them into the platform sketch. The team collaborated with a Robotics Engineering student in order to create a proper 3D design file to be sent to the printing lab. The design of the platform is shown in Figure 6.2 below,
After the platform was printed, we discovered that the four screw holes were slightly too small, meaning the platform-securing screws were unable to fit through. Therefore, the team used equipment from the MakerSpace lab to make the screw holes slightly bigger. Lastly, we ordered standoffs and screws to not only secure the platform to the car above the Raspberry Pi but also to ensure the platform will be level.

### 6.2.4 Raspberry Pi Software Configuration

The Raspberry Pi integrated into the PiCar-X runs using the Raspbian ”Buster” operating system (OS), which was released in 2019 and is slightly outdated, but due is necessary due to current versions being incompatible with the car’s driver code. Consequently, the Raspberry Pi is also running Python 3.7 for the same reason [16].

### 6.3 Web Server

As mentioned in Section 5.3, we chose to implement communications between Unity and the car via a WebSocket-based web server hosted on the PiCar-X’s Pi. In the interest of keeping latency to a minimum, we used FastAPI [17] as our framework. Furthermore, we limited communications with the server to be very simple strings of the format \([\text{speed}] \sim [\text{direction } 1 \text{ or } 0] \sim [\text{wheel_angle}]\) which would be parsed into understandable parameters for the Raspberry Pi’s functions. For example, the
web server would receive the command "5∼1∼0" and set the car to move forwards at a speed of 5 with a wheel angle of 0 degrees. The server would then send an acknowledgment message to the client connected to the open socket, or an error message if the received input was incorrectly formatted. At present, we are simply streaming our video feed directly from the 360-degree camera via the HDMI cable connected to the Raspberry Pi installed on the PiCar-X to the VR headset. We believe any modifications to the feed would be better executed on different, more powerful hardware such as the Meta Quest 2 headset. Both connections are handled using HTTP, a standard protocol for sending information between web servers. The video stream in particular relies on the Picamera library (version 1.13), which is a Python package that enables interfacing and streaming with the Raspberry Pi camera module [18].

6.3.1 Video Stream Codec

The video codec of the video stream from the car has been a major point of consideration for our team due to the inherent effect on performance, and therefore, user experience. We decided to capture and stream the camera feed directly to a connected end user via Motion JPEG (MJPEG) over HTTP at a frame rate of 30 fps. This involves taking every frame captured by the 360-degree camera, then compressing each as a singular JPEG image before sending it to the client device. Generally, this format offers a simple, computationally lightweight method of streaming that is widely supported on a variety of platforms. However, our platform of choice, Unity on the Meta Quest 2, does not natively support this format and thus requires an additional script in order to properly interpret the stream data that will be described in Section 6.4.8.

During our initial research, we explored other streaming libraries such as FFmpeg and GStreamer, but are currently using HTTP because it offers convenient support for the MJPEG format. Another major benefit inherent to HTTP is the ease of testing, as native web browser support for displaying a video stream is widespread. This allowed us to make use of a simple HTML web page to test both receiving commands and transmission of our recording data.

6.3.2 Input Handling Via WebSockets

Sending commands to move the car is handled through WebSocket communication from the VR headset to an ASGI web server on the Raspberry Pi. ASGI (Asynchronous Server Gateway Interface) is a standard for asynchronous communication that defines the specifications for forwarding web requests (viz. WebSockets) to a Python application. This asynchronous nature allows our application to run all of its operations in the background, without being blocked by potentially poor network conditions. The ASGI web server on the car was built with Uvicorn (version 0.21.0) and FastAPI, as the former already meets all ASGI specifications by design.
This ASGI web server facilitated the transport of WebSocket events originating from the controllers (Meta Quest 2 joysticks) to the Python API used to control the car servos.

The WebSocket setup itself is very simple, only sending and receiving basic movement commands in the form of strings with three-number strings. Each number is a value for speed (0 to 10), direction (binary), and angle (x to y), respectively, with the tilde (‘∼’) character as a delimiter.

6.4 VR Application

We implemented the front-end, virtual reality application using version 2021.3.9f1 of the Unity game engine [7]. Apart from displaying the camera stream to the user, the app enables users to configure the connections to the WebSocket and video stream, drive the car using the Meta Quest 2’s controllers, change which control scheme to use when driving the car, and change which of the 3D models to overlay on top of the physical PiCar-X. Though our team encountered a number of challenges, including those ranging from organizing scripts to utilizing version control to finding compatible packages, many other aspects of our project design and organization, as well as some of the project features were greatly aided by our choice of the Unity framework.

6.4.1 RC Car Control Schemes

In today’s world, controllers, remote control cars, and their control systems have become nearly universal; however, after our team created an initial control scheme to test the WebSocket connection, it became clear to us that the standard scheme of left thumb stick for turning and right thumb stick for accelerating may not be the best solution to everyone. Many consumers learn to control cars in video games with controls made for a traditional controller, so the best approach would be to offer a multitude of options that could accommodate a user’s needs the best. Moreover, the Meta Quest 2 controllers have two analog triggers, two buttons, and a joystick per controller, giving our team the opportunity to find an optimal control scheme, best tailored to precision and intuition. We considered and briefly began to implement a motion-tracking-based approach but decided that it would be clumsy and unnecessary due to the lack of a physical wheel.

We implemented a couple of other control schemes that we determined to likely be more intuitive. The A/B button system and the Left/Right Trigger system for forwards and backward acceleration respectively are two of those control schemes. Turning relies on the controller’s left joystick. We also implemented an optional setting for artificially restricting acceleration to be slower for a more natural feeling.
6.4.2 Connecting Unity to WebSocket API

The web server manages all of its connections through the WebSocket protocol, however, Unity does not natively support these connections. To create the final WebSocket connection code, we researched a number of methods for connecting to WebSockets in C# and ultimately used the built-in .NET WebSocket API to create our own implementation. We also created the WebWork namespace to house the WebSocket functionality, and we added a custom C# event class that updated subscribers to that event with the latest log information from the WebSocket, including error information and connection debug messages. This enables the Unity app to conveniently connect to the web server in a way that is functional for normal usage as well as debugging.

6.4.3 360-Degree Video Parsing

Unity does not natively support MJPEG streams, meaning our team needed to adapt to this streaming format. Fortunately, we quickly found a custom third-party MonoBehavior script that does this for Unity [20]. This script, found in the MainCamera object, creates a new thread that connects to a given URL, reads each frame, and copies the received image to the texture of the skybox material, which then shows up as the 360-degree video shown around the user in VR. The team also added a default video that would play when the video stream is not connected instead of stranding users in a black void.

6.4.4 UI

A key feature that augmented reality offers is access to digital assets superimposed upon the real world. In order to best manage the various settings of the app, our team implemented a main menu using Unity’s built-in UI system. The organization of this menu is meant to intuitively guide the user to whichever function or setting they wish to change. We split the menu into several tab pages with corresponding icons on the left-hand side of the menu: General Settings, Connections, Controls, and a hidden Evaluation page.

The General Settings page allows users to change the sound level, choose which car model they would like to use (car or go-kart), and show and hide the debug panel connected to the right-hand controller. The Connections page allows users to specify, connect, and reconnect to the car’s input WebSocket and the video stream URLs. The controls page allows users to specify which of the provided control schemes they would like to use and, should the given control scheme support it, whether the forward control sets the speed directly or causes the car to accelerate. The final page, Evaluation, enabled our team to more easily record performance measurements and is not accessible to end users of the app. Additionally, settings in the menu are saved automatically to the given device between sessions, and the
code controlling the menu can be found in the `MenuInteractionScript.cs` within the main menu `canvas` object.

The menu is controlled via digital pointers from the controllers, which allow the user to point and click menu items. However, our creation of the menu object alone did not immediately result in the UI’s appearance on the VR headset. To do this, our team had to convert the `XRRig` object provided to us by Unity’s default XR template into a Quest-specific `OVRCameraRig` object. Moreover, to allow the menu to be interactive, members of our team had to create a new `UIHelpers` prefab to house the laser pointer and the laser pointer selection sphere which together enable the Oculus controllers to interact with the menu, as well as the new OVR equipped `Event System` object which controls the functionality of the laser pointer. For text input in VR, the menu uses the Meta Quest’s system keyboard functionality.

### 6.4.5 3D Virtual Car Design

The assets for the virtual car were created by the team using Blender, a popular 3D modeling software [21]. The team ultimately ended up deciding to use a more basic, roofless car design as shown in Figure 6.3 in order to maximize visibility for the sake of actual functionality. Initially, the team found a free, realistic sports car asset, but because the physical PiCar-X is so small, it is likely that many of the things the user may want to look at while driving the virtual car would be somewhere up above them, blocked by the roof of the sports car. Our final design is a less-realistic go-kart-like design with a handful of interactive elements For example, when the
user turns the car, the steering wheel and front wheels on the car turn accordingly, maximizing user immersion.

6.4.6 Computer Vision/Visual Models

Computer vision models are crucial for this project in order to process the 360-degree camera feed. Vision models are a type of machine learning model that is capable of analyzing and extracting useful information in order to gain an understanding of the content within images and videos. For instance, one type of neural network known as Convolutional Neural Networks (CNN) utilizes convolutional layers to identify and classify objects within images. CNNs accomplish this by identifying unique patterns and features within an image and associating them with labeled objects. Furthermore, computer vision models are necessary to perform useful vision tasks such as object detection and object tracking within a video. In particular, the team was interested in vision models that were not only capable of classifying objects within an image, but also identifying their coordinates. The combination of classification and coordinate information can be used to create visual rectangles known as bounding boxes around each object.

While the team planned for a model with bounding box capabilities, the implementation was an open-ended problem. Different applications and machines required significantly different methods of implementation. The primary method of vision model implementation was through the Unity application. Unlike traditional Python implementations, utilizing Unity was significantly more difficult due to less documentation and package support for computer vision applications. Thus, the team had to manually seek out and test vision-related Unity packages to determine their viability.

The team investigated a couple of vision-related Unity packages. The first Unity package that was experimented with was the Unity Perception package [22]. This package boasted support for a Bounding Box 2D Labeler algorithm, which was advertised to yield bounding box coordinates and class predictions. However, there were a few limitations and constraints that ultimately resulted in the team utilizing a different package. First, the 2D labeler algorithm was only capable of identifying and tracking objects that were predefined with “prefab” and “texture” files. While this constraint was excusable for simple-shaped objects such as cereal boxes or canned goods, more complex-shaped objects such as computers, chairs, or people would be significantly unreliable to detect. Additionally, the Perception package required the High Definition Render Pipeline (HDRP) framework as a dependency, which was incompatible with the current project architecture. The team discovered that the HDRP framework conflicted with the Android build needed for the Oculus Quest 2. Therefore, utilizing this package would require the project architecture to be reworked, which would be costly in time and energy. As a result, the team shifted their attention to another vision package that better suited our project.

The second Unity package that the team investigated was TensorFlowSharp.
This package contained a ready-to-use YOLOv3 object detection model which the team could use via a simple function call. Thus, the team discovered that the TensorFlowSharp package provided a more straightforward implementation compared to the Perception package. Additionally, the TensorFlowSharp package provided more thorough documentation and offered other vision pipelining functions that made the model implementation simpler. Consequently, the team leveraged this more user-friendly package and created a C# script to manage the models inputs and outputs. The input of this script was essentially the view of an internal Unity camera, particularly the camera corresponding to the view of the headset wearer. For each frame detected, the script supplies the input camera view to the vision model and gathers all output information once the model processing is completed. The script then generates bounding boxes using classification, confidence scores, and coordinate data and overlays the bounding boxes on a virtual, rectangular canvas that is fixed in front of the VR car. This process aims to create an illusion of the objects being detected in front of the VE headset wearer. The team discovered that this Unity package was capable of identifying certain objects such as clocks and umbrellas within a single camera frame.

6.4.7 Version Control

Version control with Unity was a challenge for the group for a few reasons, but necessary in order to maintain the code base properly. Having chosen GitHub as the main version control software, the team looked into ways of integrating GitHub and Unity. Although the team found a Unity package that accomplished this, the package had not been updated in several years and no longer worked, prompting us to simply use the standard GitHub terminal. After finding a suitable .gitignore file online, the team quickly ran into the issue of file size. Not only was the entire project over 2 GB in size when compressed (initial GitHub upload time of about 30 minutes), GitHub limits all files to be at most 100 MB and prevents files bigger than this from being pushed at all, but after members of the team removed a few offending library files corresponding to IOS builds, the entire project could fit on GitHub.

Our team’s use of GitHub also somewhat impacted the overall organization of the Unity project. Unity saves everything associated with the asset tree in one very long file that is encoded, auto-generated, and very difficult for GitHub to merge. As a result, our team isolated every major area of the Unity project and wrapped all of the assets in said areas in Prefab objects, which are saved in separate Prefab files. This eliminated merge conflicts as teammates working in different areas no longer had to worry about editing the same asset tree file with every change. These Prefab areas include the Video Player, Cars, OVRCameraRig, MainMenu, and UIHelpers.
6.4.8 Unity Scripts Organization

Because of the way Unity itself and the C# scripts within it are organized, our team had to carefully manage the organization of the scripts within the project. These scripts fall into two categories: MonoBehavior scripts and standalone scripts.

The first are the most common scripts found in Unity because they allow for objects in the scene to be added as serialized fields and contain Start and Update methods that run when the program starts and on each frame respectively. These must be included as a component inside an object in the scene. For example, all of the functionality of the main menu can be found in the MenuInteraction.cs script which is attached to the main menu Canvas object. Standalone scripts, on the other hand, do not need to be attached to Unity objects. Instead, they can be referenced by MonoBehavior type scripts and are better suited to API calls, static classes like the WebSocket connection class, and the like.

The team primarily organized the overall project to use MonoBehavior scripts for object-specific scripts and static ones for functionality that did not need to rely on a given game object to work, but this was not always possible. Unlike the WebSocket script and the corresponding web event script (found in Section 6.4.2) which do not rely on game objects for anything and are therefore static, the third-party MJPG stream parser (Section 6.4.3) could not be made static because of its specific implementation [20].
Figure 6.4 is a diagram representing our project’s asset tree, complete with where each script is and where each functionality originates.

Another organizational concern that the team encountered related to the implementation of unit tests was script assemblies. As it turns out, Unity’s Test Framework cannot automatically call methods from within scripts used by the project without having a reference to their assembly, which warranted the creation of assemblies for the team’s scripts. Our team accomplished this by isolating the script files needed by the test framework in the dedicated Assets/Scripts/ScriptsAssemble folder within the project files and creating a Unity Assembly Definition Asset titled STVRScripts. Unfortunately, these manually created assembly definition assets do not automatically incorporate dependencies otherwise included in the project, which instead must be added manually. The STVRScripts assembly references the built-in Unity.TestMeshPro, Unity.InputSystem, as well as the STVRControls assembly which contains Unity’s pre-generated code for taking controller inputs, and the STVREvaluation assembly which exposes methods used for evaluating the deliver-
Figure 6.5: This figure depicts the Unity app’s Assemblies, their organization, and their dependencies.

6.5 Python Vision Models

Although the team primarily focused on a Unity implementation for the object detection model, the team also researched possible Python implementations. This implementation utilizes the Python programming language and open-source machine learning and computer vision libraries such as NumPy, PyTorch, and TensorFlow. Python-based models have many advantages over Unity-based models such as offering significantly more thorough documentation and code support. On the other hand, Unity-based vision packages often have limited and vague documentation. Additionally, machine learning python libraries offer a large variety of different computer vision models to select from, which makes the Python implementation approach much more flexible with computer vision tasks and applications. In contrast, Unity-based vision packages are limited to select vision models chosen by their respective software developers, thus limiting the number of applicable problem applications. Section 9.6 further discusses more intricate details regarding the vision pipeline and model implementations that still need to be addressed.

One of the notable Python models that the team considered for the final design was an object detection script for images. This script utilizes the YOLOv3 (You Only Look Once version 3) object detection algorithm; a sample result is shown in Figure 6.6. While this script had successful functionality, the limitation that the team highlighted was that the script could only process images, instead of live
camera feed streams. On the other hand, one of the strengths was the relatively fast inference time.

In addition to the object detection script for images, the team made notable progress with an object detection script for videos. This script utilizes the CenterNet object detection model that was pre-trained on the COCO (Common Objects in Context) dataset. Unlike the previous script for images, this model was able to identify and track objects throughout a given video; a sample result is shown in Figure 6.7. Although this script also had successful functionality, the big limitation was that the script required the video file to be downloaded directly to memory prior to processing. Thus, there would be significant latency if this approach was utilized. However, one of the strengths of this script was that it was much more flexible by being able to process both images and video files.
7

Evaluation

7.1 Setup

All evaluation tests were performed on a Raspberry Pi 2 Model B and a Meta Quest 2 headset. The former handled sending the video stream and receiving/executing movement commands while the latter sent commands and received, processed, and displayed the video stream. The Raspberry Pi operated using the Debian-based Raspian “Buster” OS and Python 3.7 while the headset runs version 2021.3.9f1 of Unity. The Raspberry Pi utilized a wired connection while the headset and laptop running the Unity app operated over Wi-Fi. The Pi uses the Fast Ethernet standard, which has a maximum speed of 100 Mbps. In a communication test with similar equipment, we found that there was an average round-trip time of about 2.5 milliseconds between a laptop and the Raspberry Pi.

7.2 Metrics

7.2.1 Response Time

Response time/Input latency, or input lag, is the amount of time between a user inputting a command and the actual execution of the said command by a program or game. Generally speaking, minimizing input latency as much as possible is desirable, and increasingly so for applications where fast reaction times are critical. Some amount of lag is unavoidable due to the time it takes for a system to receive, parse, and act upon any given command, but any amount of delay above 70 milliseconds is problematic [25].

7.2.2 Frame Rate

Frame rate, measured in frames per second (FPS), is the frequency at which a new image is displayed for the user. In the context of this project, this refers to how often
the camera feed is updated. There are a variety of factors in both hardware and software that can limit an application’s frame rate, and variance is common. The Kodak Orbit 360, the camera mounted to the PiCar-X, has a maximum frame rate of 30 FPS at our chosen resolution of 1920x960 (2:1) for live streaming. Therefore, our ideal frame rate should be 30 FPS. Refresh rate, measured in hertz (Hz), is a related metric that describes how frequently new images can be drawn on a screen. The Meta Quest 2 has higher possible refresh rates (beginning at 60 Hz) than our camera, meaning we are not concerned with the display itself being a limiting factor. In a variety of studies measuring the human performance of remote-controlled vehicles with varying frame and refresh rates, it was found that drivers could consistently stay on a predefined track between 7.5-30 Hz, with their performance declining significantly below 5 Hz [26].

7.3 End-to-End Evaluation

The end-to-end tests evaluate the performance of the project components working together and are the closest measure of how well our pipeline would work in practice. We gathered the data by connecting the headset to the PiCar-X’s WebSocket servers over Wi-Fi and measuring the receipt of packets within the Unity app.

7.3.1 Response Time

In order to evaluate the connection between the PiCar-X’s WebSocket server and the Meta Quest 2 VR headset, the team enabled the Unity app’s WebSocket class to record both the exact time a message is sent to the WebSocket and the exact time the corresponding answer is received. Before the team implemented this, the WebSocket response included only the current command being executed by the car in the same format as described in Section 6.3.2. By sending different inputs each time, the team was able to differentiate between server responses and determine a round trip time for each command sent to the server. When both the PiCar-X and headset were connected to the same home Wi-Fi network, the response time averaged out to 8.795 milliseconds over thirty seconds of testing, as shown in Figure 7.1. As detailed in Section 7.2.1, this is well below a problematic value and should not impose any major limitations on the user’s ability to properly control the car.
The team also recorded the time it took Unity to process controller inputs: the time passed when the control values are read from the joystick to before the message is sent. This value was very low, averaging out to 1 millisecond over thirty seconds of testing, and as such, should not pose any threat to app usability, especially since it holds such a small share in total input latency.

7.3.2 Stream Delay

To better understand how quickly the user would be able to respond to a change in the PiCar-X’s surroundings, the team measured several aspects of the MJPEG stream. First, the team manually timed how long it took for the video to reach the headset by filming a timer with the PiCar-X’s camera, streaming the resulting video to the headset, casting the Oculus’ view to a computer, and taking a number of side by side pictures of both the ground truth timer and the delayed video stream. While this test does include the added delay from casting the video to a computer screen, that error should not be more significant than that which would be generated by manual human timing, and it is certain that the actual average stream delay will be
The team performed this test multiple times at equal intervals after the start of the stream and yielded consistent measurements, with Figure 7.2 showing a ground truth time of 15.42 seconds and displayed time of 14.75 seconds to get a difference of 670ms or 0.67s. Figure 7.3 compares this result with the average human simple reaction time (time to respond to a single stimulus) and recognition reaction time (time to select the optimal response to multiple stimuli), both of which are relevant to a remote driving experience [27]. While the visual stream does have a substantially higher delay than the minimum that would be noticeable, past research indicates that a delay of 700 milliseconds or greater is when remote driving ability significantly degrades [28]. A delay of fewer than 0.67 seconds would negatively impact users’ ability to drive the PiCar-X, but is still operable.
7.3.3 Stream Frame Rate

Our team measured the number of frames received by the headset over the span of 30 seconds to get an average FPS. The frame rate of the video stream received by the VR headset was found to be about 59.97 FPS, with 1799 frames being received during the test period. Our team also recorded the time between the receipt of each frame to calculate the marginal FPS, which describes the theoretical FPS an application could output if it were rendering at the rate frames are received.
We demonstrate the delay between each frame being received to be about 16 milliseconds in Figure [7.4]. Figure [7.5] depicts the calculated marginal FPS, which appears to correlate with our measured average of about 60 frames per second. We processed the data by removing any values above or below three standard deviations from the mean, which was either above 187 FPS or 0 FPS. This trimmed about 5 values above the upper threshold. The varying delays between the receipt of each frame caused some calculated values to be far above the actual possible values and consequently trends higher than what other evaluation methods suggest.
Notably, the maximum frame rate of the 360-degree camera is 30 FPS at our chosen resolution, as stated previously. Although this data seems to contradict that, closer inspection of the actual packet payloads within WireShark indicates that any connected source often receives any given frame twice. That is, the camera is recording at 30 FPS while the video stream is attempting to deliver 60 FPS. As a result, the stream attempts to fetch a new frame before it is captured by the camera, so it simply sends the old frame a second time. After examining the Picamera documentation, our team believes the issue is due to an incompatibility between that Python library and the 360-degree camera, as it is unable to properly query the hardware’s FPS limit upon being called [18]. While the headset may receive additional data, the video stream itself still maintains its true speed of about 30 FPS due to there being roughly 30 unique frames being delivered every second even while being live-streamed, meaning transmission from the car to the headset does not significantly reduce video quality.
7.3.4 Overall App Performance

Figure 7.6: Comparison of end-to-end FPS values and the estimated human perception speed of 30-60 FPS [29]

When it comes to the Unity application overall, most of its functionality occurs on a once-per-frame basis; all of the rendering, controller input polling, vision algorithms, etc., are all called once per frame. As a result, if the framerate is too low, users may be unable to respond continuously to the PiCar-X’s environment, leading to frustration and a breakdown of the app’s usability. Because of this, our team implemented measures to monitor how fast the Unity app can perform its update loop every frame by recording the times before and after each new update. We then recorded the average frame rate of the app, which measured 16.57 FPS in our end-to-end test. As shown in Figure 7.6, the frame rate within the application displayed to the user is lower than that of the received camera stream and the estimated human perception speed [29]. Although this frame rate is not ideal for human perception, it is still well within the usable range as described in Section 7.2.2.

7.3.5 User Stories and Manual Testing

Because our evaluation metrics could not cover the entirety of the user-related Unity functionality, the team created a number of manual tests and corresponding user stories (as in Section 7.5.2) to cover the rest of our implemented functionality. Of these tests, those relating specifically to end users include evaluating if the Unity app is able to connect to the video stream and WebSocket, evaluating whether or not the controls can be used to maneuver the PiCar-X, and evaluating how easy or
difficult it is for users to do so. The corresponding user stories and full manual test plans can be found in the appendix A.

7.4 Web Server Component-Based Evaluation

It was important for our team to measure the performance of individual components so that we could accurately describe the sectors of the project most severely impacting or limiting the user experience. The web server portion pertains to the initial transmission of the PiCar-X’s camera feed to the Unity pipeline, as well as the receipt and execution of movement commands. As the web server is hosted by the car’s onboard Raspberry Pi, our testing setup had the server and our receiving systems connected to the same network over Ethernet. This was to ensure that we could ascertain the speed of each server component with minimal variance brought on by Wi-Fi latency. We chose two variables to measure, input latency and frame rate. These were picked because we believed them to be the most likely causes of user discomfort should they fall below unacceptable levels.

7.4.1 Input Latency

To test the delay between sending and receiving inputs to and from our web server, we used a separate computer that used a test HTML page with pre-programmed debug functions to send commands to the Raspberry Pi in the same format as those expected from Unity. That is, text in the format of “[speed]∼[direction]∼[angle]” where all three variables are integers. We then used a script to send 1000 randomized commands over to the car, with a 100-millisecond delay between each to ensure the system has time to properly address each. Each message, including the car’s responses sent right after executing any given command, has an associated timestamp. We used the developer tools built into the Chrome browser to save the resulting timestamps of all the WebSocket transmissions, then calculated the difference between each to get the round-trip time.
Over the course of about 100 seconds, we recorded an average delay of about 11 milliseconds between a client sending a message and the PiCar-X running the corresponding command. We make the simplifying assumption that the physical execution of a command is instantaneous after the function is called. Although there are numerous outliers above the average, none exceeded 35 milliseconds of delay, which demonstrates that the web server is responsive enough for human users to operate the car without any notable interference.

### 7.4.2 Frame rate

To measure the FPS of the camera solely on the web server end, we opened a live video stream, recorded all packets on the network over a 30-second period using WireShark, then parsed the output to get an average frame rate over time. This was possible due to the MJPEG format of the stream, as every frame sent to a client is preceded by a header containing “FRAME” as well as other metadata, meaning the task of finding FPS is as simple as counting the number of occurrences. Our results showed we received about 1740 packets in our 30-second testing window for an average of 58 FPS throughout our testing, which indicates that the web server
itself does not induce a noticeable drop in quality. Furthermore, we had a fairly steady 17.267-millisecond delay between each new frame received.

As mentioned in Section 7.3.3, our average frame rate is being artificially boosted above its expected limits. Although this obfuscates the actual frame rate of our pipeline, we can infer that the lack of dropped frames indicates the web server itself does not impose any major limitations on the FPS of the final product.

7.5 Unity VR App Component-Based Evaluation

7.5.1 Unit Tests

In order to better evaluate the functionality and correctness of the Unity app’s code base, the team created a number of unit tests using the Unity Testing Framework. Apart from a few sample tests provided by Unity, our team created tests for testing if the WebEventArgs class correctly updated its messages when the WebEvent was invoked, for testing whether or not the menu correctly opens, switches pages, loads, and saves user settings, for testing if the debug console correctly reads and displays log messages, and for testing whether the class responsible for handling controller inputs has functioning helper methods. All tests were shown to run successfully by the end of the project.

7.5.2 Manual Testing

While the unit tests and manual end-to-end tests did cover most of the functionality of our app, not all of our app’s functionality directly relied on the connection to the web server, video stream, or even to the Meta Quest itself. The tests that did require the headset but not the PiCar-X include: evaluating if the menu is visible, accessible, navigable, and easy to interact with, evaluating whether the virtual car model is well-lit and visible, and evaluating whether or not the car’s cosmetic functions (such as its steering wheel and front wheels turning when the user turns the PiCar-X) work appropriately. Our team also designed several manual tests geared toward future developers and debug functionality. These tests include evaluating whether or not the debug panel is readable when enabled and whether or not the app works in Unity’s desktop player with a mouse and keyboard (when the appropriate settings are turned on). The corresponding user stories and full manual tests can be found in the appendix A.

7.6 Unity Vision Model Evaluation

Like any machine learning model, an effective and reliable evaluation metric is needed in order to test the performance of the Unity vision model. The Unity vision model uses both classifications and bounding boxes as part of its prediction
output. Thus, different individual metrics will be used to evaluate each component of the model results.

It should be noted that the video data utilized within the model evaluation tests were the raw 360 camera footage, thus content near the edge of the video frames were significantly warped. This warping can be seen in Figure 5.3. Despite the video footage being warped near the edges, both the Unity and Python vision models were able to detect and classify objects near the center of the video footage where objects were significantly less warped. Future work that is needed to address the warping issue is discussed in Section 9.6.

7.6.1 Intersection Over Union

The team utilized the Intersection-over-Union metric to evaluate the bounding boxes. This metric is used to address the quality and comparison of bounding boxes within an image or frame of video. This is accomplished through the Intersection-over-Union (IoU) metric in which the equation is shown below,

$$\text{IoU}(A, B) = \frac{\text{area of overlap}}{\text{area of union}} = \frac{|A \cap B|}{|A \cup B|}$$

(7.1)

This is also known as the Jaccard Index formula. Given two distinct bounding boxes A and B, the IoU metric determines the ratio of the overlap of A and B to the union of A and B. Intuitively, this can be thought of as the ratio between the similarity of two bounding boxes to their dissimilarity. The metric spans from 0 to 1, with 0 corresponding to zero overlap between two boxes and 1 corresponding to the boxes being the same.

7.6.2 Precision and Recall

Another part of the model output that needs to be addressed is the correctness of the predicted classifications. This is accomplished by the precision and recall metrics. Precision is defined to be the fraction of correct classifications amongst positively-predicted classifications. Recall is defined to be the fraction of correct classifications amongst the positively-labeled classifications. Since both precision and recall span from 0 to 1, a perfect classification model would have precision and recall values that are both 1. In essence, a good classification model is both accurate and thorough in the data distribution. Thus, both precision and recall are necessary for the Unity model evaluation in order to obtain a robust measure of overall performance.

7.6.3 FPS and Inference Time

In addition to model evaluation metrics, there are a couple of other statistics that can help obtain a heuristic view of vision models. The FPS metric is useful for determining the speed and efficiency of a vision model; larger FPS values signify
more efficient model performance. Similarly, another metric is the inference time, which is the amount of time required by the model to completely process a single frame. Smaller inference times signify lower latency and faster performance. Thus, these two statistics provide insight into how effective and efficient the model is able to detect objects and draw bounding boxes.

### 7.6.4 Evaluation

There was no existing annotated dataset for the 360-degree camera, so the team needed to laboriously create manual annotations on video frames in order to establish ground truths. While the team was able to completely annotate a 15-second, 30 FPS video of a living room with 4 people, this still resulted in limited data for model evaluations. Consequently, the precision and recall metrics were restricted to whether an object was a person or not in order to prevent numerical instability in these calculations. A sample instance of the numerical instability problem is shown in Figure 7.8 below.

![Figure 7.8: Numerical instability arises from analyzing multi-class results.](image)

The team performed evaluations on the Unity and Python vision models by collecting data on all of the above metrics, yielding Table 7.1 below.

<table>
<thead>
<tr>
<th></th>
<th>IoU</th>
<th>Precision</th>
<th>Recall</th>
<th>FPS</th>
<th>Inference Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unity Model</td>
<td>0.0</td>
<td>0.9682</td>
<td>0.2185</td>
<td>11.408</td>
<td>73.5</td>
</tr>
<tr>
<td>Python Model</td>
<td>0.5417</td>
<td>0.9325</td>
<td>0.7014</td>
<td>15.067</td>
<td>61.0</td>
</tr>
</tbody>
</table>

The most notable results from Table 7.1 are the IoU, precision, and recall metrics. The Unity vision model yielded average IoU, precision, and recall scores of 0.0, 0.9682, and 0.2185 respectively. This was unsurprising as the performance of the Unity vision model was severely hindered by currently unknown issues with the Unity implementation. The low IoU and recall scores signify that the model could not detect all people within a video frame, nor could it accurately locate the positions of each person. Yet, the high precision score means the model was able to make accurate predictions whenever it could.

On the other hand, the Python vision model yielded average IoU, precision, and recall scores of 0.5417, 0.9325, and 0.7014 respectively. The higher IoU and recall scores indicate stronger performance compared to the Unity model.
scores mean that not only was the Python model more capable of detecting all people within a video frame, but it could also more accurately locate each person. Similarly to the Unity model, the Python model’s high precision score signifies that the model made accurate predictions whenever it could. While both models have high precision scores, this is explainable by the fact that the annotated video clip contained mostly “people” labels and very few “non-people” labels.

The FPS and inference time statistics provide further insight into the performance of the models. The Unity model yielded average FPS and inference times of 11.408 and 73.5 ms respectively, while the Python model yielded average FPS and inference times of 15.067 and 61.0 ms respectively. In other words, the Python model is able to process each frame on average 12.5 ms faster, resulting in the Python model being able to process 3-4 more frames each second. Therefore, utilizing all the metrics and statistics above, the Python model has an overall significantly better performance than the Unity model.
8

Known Issues

8.1 Stream Delay

As detailed in Section 7.3.2, the delay between displaying the received camera feed and the actual camera feed is about 670 milliseconds, just under the known problematic threshold. As a result, the user’s driving ability is negatively affected. Ideally, this delay could be reduced as much as possible. The human recognition reaction time is most relevant to this project, so a good target latency would be about 350-450 milliseconds.

8.2 Unity

While the vast majority of the Unity app’s features and code are functional and well-tested, our team did not have the time to test and verify every detail of the Unity app’s code. The first issue relates to the menu, specifically the connections page which allows users to connect to the web server and MJPEG stream. Currently, the Meta Quest system keyboard does not display what is entered by the user, and while the menu does show the input as it is being typed, the system keyboard sometimes blocks it, making it hard for users to input long, complicated strings such as websocket or video URLs. Second, because Unity-specific methods cannot run in async methods or in any code that runs on a separate thread, the WebEvent class does not always properly record debug information coming from the WebSocket and MJPEG classes, both of which use multithreading; however, this is a very minor issue that will likely never be noticed by end users. The third and final issue stems from the rapid implementation of the evaluation measurement system, which is not covered by unit tests or by very robust error handling.
8.3 Vision Model

The vision model was implemented using a third-party Unity package, which unfortunately makes the model prone to Unity-related errors. The model creates seemingly nonsensical predictions with bounding boxes that are offset. Figure 8.1 depicts an example prediction in which the Unity vision model misclassifies a wheel of a car as a clock. Interestingly, this may simply be explained by the fact that the car wheels have spokes which make them very similar to the hands of a clock. In addition to the misclassification, the bounding box generated by the vision model is significantly offset from the car wheel’s center. In an attempt to identify the cause of this issue, an experiment on the quality of the camera feed using a separate Python YOLO model was performed. Unfortunately, the team concluded that the video quality was not the issue’s culprit.

While the team was unable to determine the exact cause of these issues, there are a few possible causes. The potential cause of the misclassification problem could simply be the fact that similar-looking objects will result in similar classifications. A solution to this issue would be that the weights of the Unity vision model would need to be fine-tuned on similar objects to help the model better distinguish between them. In terms of the box offset issue, the cause likely lies within the implementation of the Unity vision setup. Within Unity, a complex configuration of cameras, textures, and 2D rectangular canvases was implemented to create the illusion of object detection algorithms in front of the user. Due to the complexity of such a vision setup, the object detection process within Unity is highly prone to both major and minor errors. Further investigation into the vision setup would be necessary in order to pinpoint the cause of the box offset problem.
8.4 Unity and Vision Integration

While the vision model was functional when tested in a desktop environment, our team could not successfully build the project for usage on the headset. Moreover, when running the unity-vision model on a desktop, it suffered from subpar performance and maxed out the CPU running on desktop hardware, which is much more robust than that on the Meta Quest 2. As a result, our team did not further pursue debugging or attempting to run the vision model with the rest of our project pipelines. Our team suspects that, given the resource drain of the model, it would not be usable on the headset regardless.
9

Future Work

9.1 Stream File Type

The file format of the video stream from the car has been a major point of consideration for our team due to the inherent effect on performance and user experience. As mentioned previously, the original format of the video stream, MJPEG, is not natively supported by Unity, requiring an additional library to be transcoded. Our team considered the possibility of using the H.264 codec, as it is one of the most prevalent and compatible available. Furthermore, despite being computationally light, MJPEG does require a substantially higher bitrate than H.264 when streaming, meaning it demands a higher bandwidth to perform well [30]. This is primarily due to MJPEG only compressing individual frames while H.264 can compress across frames, meaning an overall smaller file size is streamed when subsequent frames are similar.

Although encoding an H.264 stream would theoretically make this option more hardware-intensive than MJPEG, the dedicated GPU on the Raspberry Pi should be able to maintain the stream quality while reducing overall lag. Unlike the current codec, H.264 also natively supports syncing audio and video, which could be a major boon to users. Our team attempted to implement this video format because of these benefits and found it fairly simple to do from the server end. However, Unity also does not natively support raw H.264 streams, and our team was unable to find a working solution that did not involve egregious spending on paid third-party packages or extremely slow CPU texture parsing. For these reasons, and because the MJPEG stream was already functional, we did not pursue this matter further.

9.2 Intermediate Web Server

Currently, all processing for the project is handled either on the PiCar-X’s Raspberry Pi or the Meta Quest 2 headset. While sufficient for the current pipeline, it could be extremely beneficial to introduce an intermediate web server between the camera
and the headset. This server could receive the raw video stream, apply any desired vision model, then re-stream to the user, allowing for more computationally intense operations than the old Raspberry Pi distribution. This would introduce a major increase to the potential processing power available to any vision or machine learning models, which could in turn accelerate the speed at which the stream is processed. Alternatively, a newer Raspberry Pi board could be swapped in place of the current Raspberry Pi 2, but the processing power of small, single-board computers is still limited compared to larger modern hardware.

The primary concerns involved with creating such a server would be additional latency between the headset receiving the live feed from the car, as well as the additional cost. Additionally, the intermediate server could perform video transcoding tasks that the PiCar-X’s Raspberry Pi or the Meta Quest headset could not. Should the stream file type be changed as recommended above, then converting the raw H.264 stream into an efficient format that Unity could recognize, such as .MP4, would be a practical usage. This would remove the need for an MJPEG or other video decoder, freeing up resources on the headset for other tasks and potentially enabling end users to perceive the car’s surroundings through sound, not just vision.

9.3 Auto Start

Currently, when the car is turned on it requires the operator to SSH into the Raspberry Pi to start the movement and camera servers. In the future, this should be done on boot via a script run on boot. Ideally, this script would also include means to get the car on the desired Wi-Fi network, potentially via environment variables stored on the file system. While a simple script currently exists to start both servers with default parameters, it is very basic in form and requires manually killing the processes via an external tool (TOP, et. al) when they need to be shut down or restarted.

9.4 OS Limitations

The PiCar-X’s integrated Raspberry Pi is currently running the Raspbian "Buster" operating system (OS), initially released in 2019. This OS has become deprecated by newer releases and consequently is subject to incompatibility with many modern libraries, as well as possible security flaws and bugs. These libraries could potentially improve performance in multiple different sectors, especially if more functionality is added to the Raspberry Pi itself. The drivers for the car’s motors are reliant on this OS version, and the team briefly but unsuccessfully attempted to run them after updating it. Therefore, we believe updating the PiCar-X’s built-in code to be compatible with a newer Raspberry Pi operating system would be a significant but worthwhile task.
9.5 Unity

Aside from the aforementioned known issues and the opportunity to implement large-scale optimizations in computer vision and stream parsing with or without an intermediate web server, opportunities for improvement on the Unity side are fairly small and focus mostly on ease of use.

9.5.1 In-App Menu

The main menu could also be improved upon by the implementation of a dragging feature common to many VR systems. Because the menu currently exists in a static location in front of the car, users may want to move it around their field of view or move it closer to them. Moreover, it may be difficult for first-time or less tech-savvy users to find and correctly format connection strings, a problem that could be fixed by a connection to a synced website of some kind.

9.5.2 Controls

The Unity app’s control schemes pose another area for improvement in the Unity app. Our team did not have the time to extensively test the best possible control scheme, and while we did create a number of options, further playtesting may be required to find the best solution. Moreover, that solution may not even need to rely on Meta Quest controllers because the Meta Quest supports hand gestures, which could be used to control the car by interacting with a virtual steering wheel, virtual buttons, levers, etc.

9.6 Vision

Object detection tasks form an essential part of this project, thus an effective and reliable computer vision model is crucial. While the team made significant progress in a Unity implementation of an object detection model, there were many difficulties that hindered the completion of the project. The primary difficulty with implementing vision models within Unity was that the Unity platform had little support for computer vision applications. The Unity platform is a software framework that is primarily used for video game development, rather than for purely computer vision-based applications. Although there does exist a few examples of object detection models done within Unity by other Unity practitioners, these examples provided little documentation and were not flexible enough for our project’s needs. Additionally, Unity’s user interface is necessarily complex for video game development purposes but is too complex for standard machine learning and computer vision applications. Consequently, the Unity vision model approach became overly reliant on trial-and-error development, and the model was significantly limited in behavior by the Unity application itself. Therefore, any future work into the vision component
of this project is highly recommended to be done in Python for simplicity and easier management.

The Python programming language is more capable of handling the computer vision needs of our project. As previously mentioned in Section 5.3.9, Python supports many data science and machine learning libraries that aid in the development of a vision model such as Pandas, PyTorch, and TensorFlow. The Pandas library is useful for data pre-processing purposes such as gathering, organizing, and processing data to be compatible with an object detection model. Similarly, the PyTorch and TensorFlow libraries are useful for creating and training detection models from scratch or fine-tuning a pre-trained detection model to fit the project. A Python approach to the project’s computer vision needs is much more natural because of how much support and documentation there is for machine learning in Python. Python also allows for the significant separation of computer vision tasks from the limitations of Unity. Any future work in the project’s vision component should be done in some development environment dedicated to Python. A separate development environment helps ensure that the code development experience is simpler, but it also reduces the overhead of the vision model, making the vision model’s behavior more predictable and flexible.

A few notable details regarding a python-based implementation still need to be addressed. The team recommends the development of a web server dedicated to the vision model. This web server would need to not only house the computer vision algorithm, but also handle communication with the Camera on the RC car and the Unity server. Similarly, any future work will also require the development of a Python script that is dedicated to handling the inputs and outputs of the Python-based vision model. Once the web server receives camera footage from the RC car, the Python script is executed, and any model outputs, e.g. bounding box information, are sent to the VR headset by the web server.

The team also recommends any future work to involve obtaining a fully-annotated dataset for training, testing, or fine-tuning purposes. An appropriate dataset for this project would most likely consist of not only images and videos of everyday handheld objects such as bottles, books, and cereal boxes, but also people. While these kinds of data already exist on the Internet, it is preferred that the data is obtained through the use of a 360-degree camera, which severely limits available annotated datasets. Thus, future work on this project would require the acquisition of 360 camera footage, either from manual annotation or a thorough investigation of annotated datasets on the Internet. Yet, the incorporation of 360 camera footage data introduces the issue of the content within the raw camera footage being warped. The team recommends any future work to include the research and investigation of footage de-warping algorithms in order to reduce the effects of warping. However, footage de-warping algorithms as part of the data pre-processing step carry the risk of significantly increasing the latency of the overall end-to-end system. Thus, further investigation into the advantages and disadvantages of footage de-warping algorithms should also be done.
Conclusion

The intent of our project was to create a viable, expandable framework for a remotely-controlled miniature car that can be easily and comfortably driven by a user wearing a VR headset. Although we were unable to completely integrate all of the desired functions, we believe we have created a suitable baseline for future endeavors. We have successfully modified the PiCar-X to function with a camera capable of 360-degree vision, developed a pipeline that can stream the camera view to a Meta Quest 2 headset, and created prototype vision models that can run within Unity. Our team also identified numerous aspects for improvement and future work such as separate servers for processing, vision model refinement, and potentially a change in the engine for the digital assets. We are excited about the future opportunities this project can fulfill for future development in the field of virtual reality.
11

References


Appendix A

User Stories

- As a user, I want to see a reasonably responsive live video feed from the point-of-view of the car, so that I can experience its environment in an immersive way.

- As a user, I want to drive (steer/accelerate/decelerate) the PiCar-X using the Oculus Touch Controllers, so that I can control the car from the Unity app on the Meta Quest 2.

- As a user, I want to have a clear and intuitive graphically-based menu from within the Unity app, so that I can easily customize my experience.

- As a user, I want to be able to connect to the web server on the car using the menu, so that I can use it with different networks/settings without needing to modify and rebuild the application.

- As a user, I want to be able to change my controls using the menu, so that I can choose the most natural feeling interface for myself to interact with the application/car.

- As a developer, I want a virtual version of the car within the app, so that the user can have a more immediate indicator of how the controls are being registered.

- As a developer, I want an easily visible debug panel within the VR panel, to monitor console output and debug messages while testing the Unity application.