Development of Sensor Array towards the Cross-embodiment Pre-training

Rosh Ho

Introduction

One of the goals of the CPT project is to achieve multi-fingered dexterous manipulation by examining multi-modal human demonstrations. Such demonstrations should include a wrist view in addition to an ego-centric view. The primary goal of my project is to build hardware and develop the setup for such data collection. The key modalities to collect with setup are an RGB and depth image stream from the wrist position, finger-tip tactile sensing, and potentially motion capture if the timeline permits.

Hardware

Objective

The objective is to obtain a visual RGB and depth data stream from both the wrist and head point-of-view; the most cost-effective and time-efficient option would be to acquire one off-the-shelf. As such, off-the-shelf gloves were used for the tactile sensing gloves, head mount, and wrist mount, while the remaining sensors and adapters were custom-made. *Fig 1* illustrates the current setup worn on a person for visualization purposes. This differs from Wang et al., 2024's *DexCap* ('mocap to robotic manipulation') includes the force exertion from the finger-tips during data collection to allow for further dextrality during object manipulation as opposed to a binary outcome of contact or no contact.

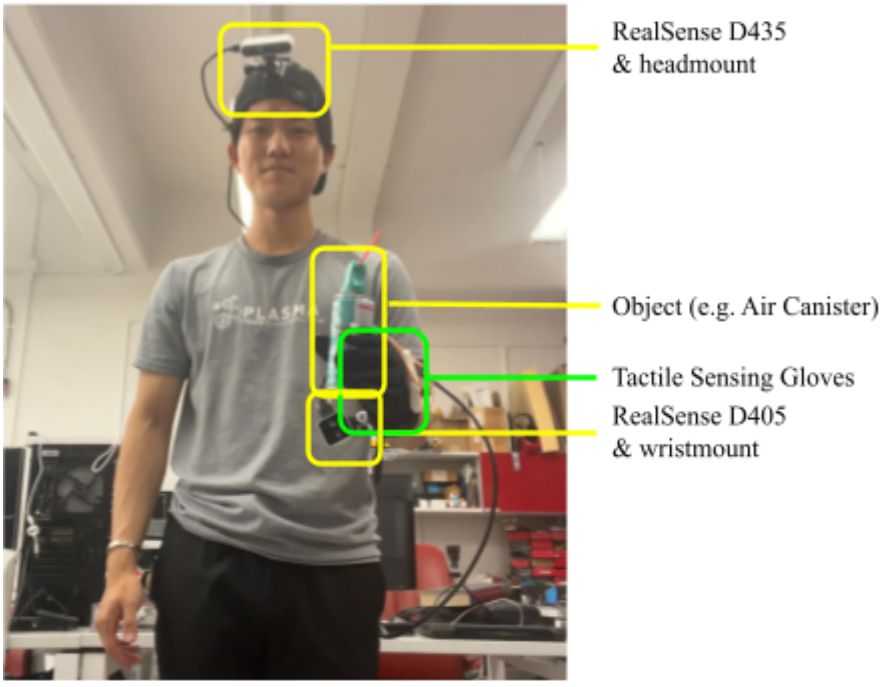

Figure 1: Head mount, wrist mount, and glove setup demo picture

RGBD Sensor Criteria & Sensor Choice

As mentioned, we aim to collect RGBD data from both egocentric and wrist views. Depth information can be achieved explicitly with a depth camera or include such information implicitly with the help of mirrors (Chi et al.)(*Fig 3*). While such a setup would accomplish including depth information with a much lower cost to the experiment, it likely causes incompatibility due to representation learning used in the CPT project. Hence, we chose to collect depth information explicitly with RGBD cameras.

Numerous RGBD cameras exist - Microsoft Kinect, Persee 2, Intel RealSense D435, D405, Leap Motion 2, etc.,. Below, we will discuss our choice of egocentric and wrist-view cameras.

The requirements for an egocentric view camera are similar to the Intel RealSense cameras commonly used for robotic manipulation. Hence, we choose a widely used D435 camera with an ideal minimum operating distance of 30cm - 300cm, sufficiently encompassing the space from the head to the hands and beyond. As seen in *Fig 4*, the head-mounted sensor selected was the Intel RealSense D435.

However, the requirements for the wrist-view camera are different. First, we require the minimum operating distance to be well under 30 cm, the average distance between the camera position and the tip of the fingers. In addition, we require that the camera be as small as possible to be convenient for manipulation. A recently released model of the IntelRealSene camera - the D405 came to our rescue. As seen in Fig *2*, the Intel RealSense D405 was selected as the wrist sensor due to the small form factor, which prevents interference from potentially limiting wrist mobility, advertised high resolution, and short minimum operating distance of 7cm, the shortest amongst all the options. The Intel RealSense D405 was optimal for the wrist-view data collection.

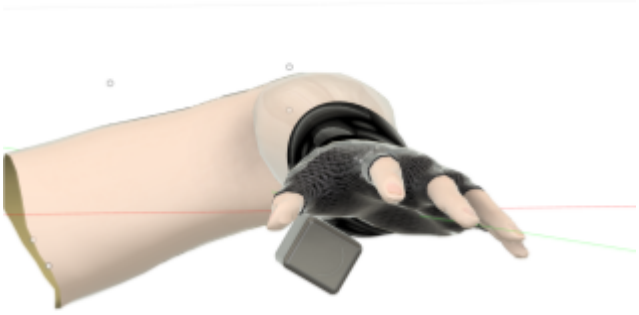

Figure 2: Wrist mount with Intel RealSense & light with glove

Figure 3: Wrist mount with Chi Et Al's stereo mirror with glove

Figure 4: Wrist mount with Intel RealSense, Chi Et Al's Stereo Mirror, glove, and head mounted Intel D435 positioning

Wearable mount

We use readily available cost-effective wearable mounts produced for a GoPro to mount the selected cameras to a human demonstrator. However, the included adapter mounts created an offset (*Fig 5*) between the wrist and the camera that was too large, posing potential clearance issues while conducting an action. To address this, I printed custom low-profile Gopro adapters and RGBD Intel Realsense sensors, as seen in *Fig 6* and *Fig 7*. *Fig* 6 shows the first version printed without accounting for the mass of the sensors leaning towards one side of the mount, causing imbalance. This was subsequently corrected in *Fig 7* by shifting the mounts with additional support pegs in between the compliant GoPro slotting mechanism.

Due to the RGBD Intel Realsense sensors operating on industry-standard M5 screws, additional GoPro-to-M5 metal screw adapters were acquired as 3D-printed plastic threads are not able to sustain high tension from the screws and are vulnerable to breakage.

Figure 5: Original GoPro mount

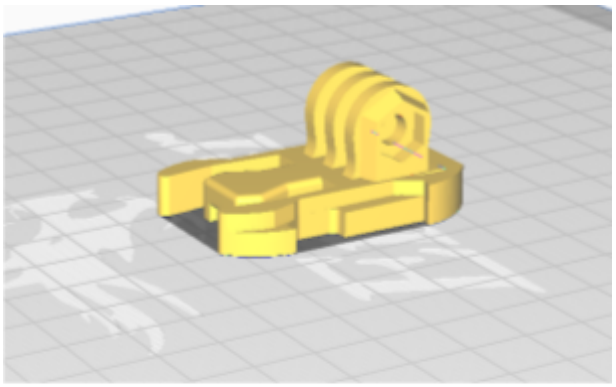

Figure 6: Gopro low profile mount v1

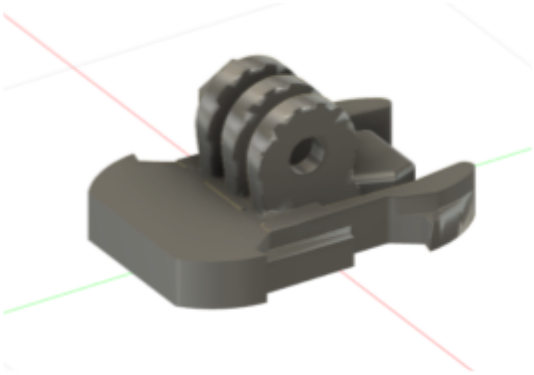

Figure 7: GoPro low profile mount v2

Cable management

As seen in *Fig 1*, another issue that may cause difficulties is the lack of cable management, causing potential disruption to the data collection. As such, custom 3D printable mount points for the velcro armbands are modeled as seen in *Fig 8*, *9*, and *10*.

Figure 8: Frontal view of cable management armband

Figure 9: Top view of cable management armband

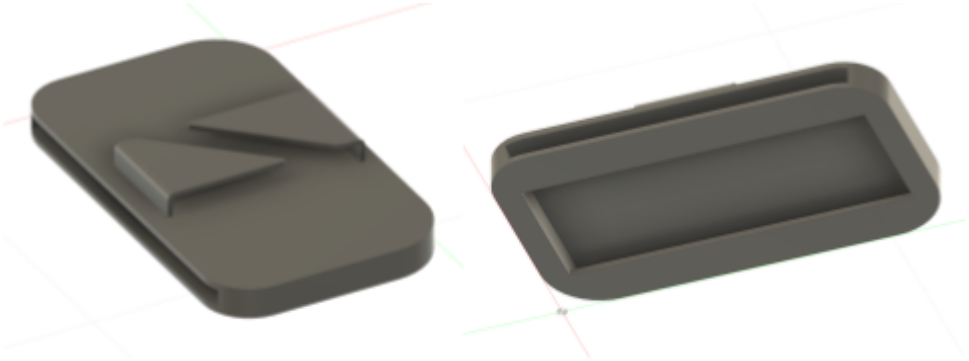

Figure 10: Cable management holder

Tactile Gloves

The tactile finger-tip gloves are intended to capture fingertip force exerted while manipulating an object for more difficult manipulation with the 2-finger and 5-finger robots. The tactile gloves capture fingertip forces exerted on the object for a better understanding of object manipulation. As seen in *Fig 11*, the glove is constructed with a teensy 4.0 and single-tact sensor due to equipment availability and ease of use. As the project continues, we seek to upgrade to a single-tact sensor with a larger surface area to better register the entire fingertip for engagement instead of a potential lack of inputs. The current issues with this glove are the need for more accommodation for different hand sizes and limited positioning for the sensor pads, which vary between hand sizes. The following steps are to sew on strips of velcro pads under the fingertips on more stretchable gloves to allow easy repositioning of the sensor pads between data collection runs.

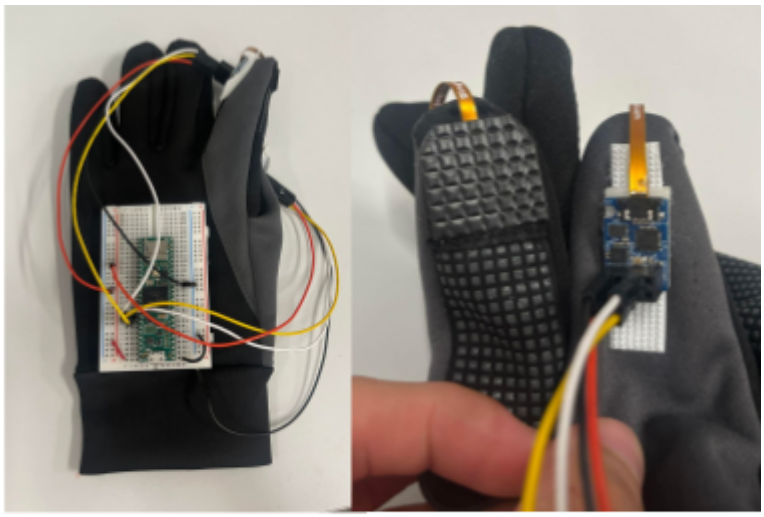

Figure 11: Tactile Fingertip Gloves (back, front)

Software

In this section, I will discuss the progress in developing the software setup for data collection.

Troubleshooting

While Intel Realsense Viewer can visualize and record the raw data, the criteria to capture the data in tandem with the tactile gloves renders the data collection to be done through ROS2.

The Intel RealSense D400 series sensors are listed as compatible with ROS1. Despite this compatibility list and Intel RealSense D405 falling under the D400 series, the D405 was introduced later, rendering library support only available through ROS2. This minor detail caused a significant delay in the data collection process.

Data Collection

Fig 12 & 13 shows Rviz2's RGB and depth stream visualization. While the depth may seem noisy in the visualization, the distance data will remain consistent with the object and will be better interpreted with a plain background. The current renderings are based on arbitrary minimum and maximum depth values and await further optimization to find the ideal range. *Fig 14* illustrates the intended computation graph with an Intel node publishing Intel RealSense D405 and D435 depth, RGB image stream, while a separate node publishes the tactile input from the glove. This is then all subscribed by an individual node. As of the status quo, the RGB and depth visual streams are captured at 35hz through ROS2 as images (*Appendix I & II*).

The follow-up actions are to capture the depth data and put it into the point cloud. Then create a subscriber logger node to record all the data points. Also convert the data points into actual measurements as the default raw data is not recorded in metric or imperial measurements.

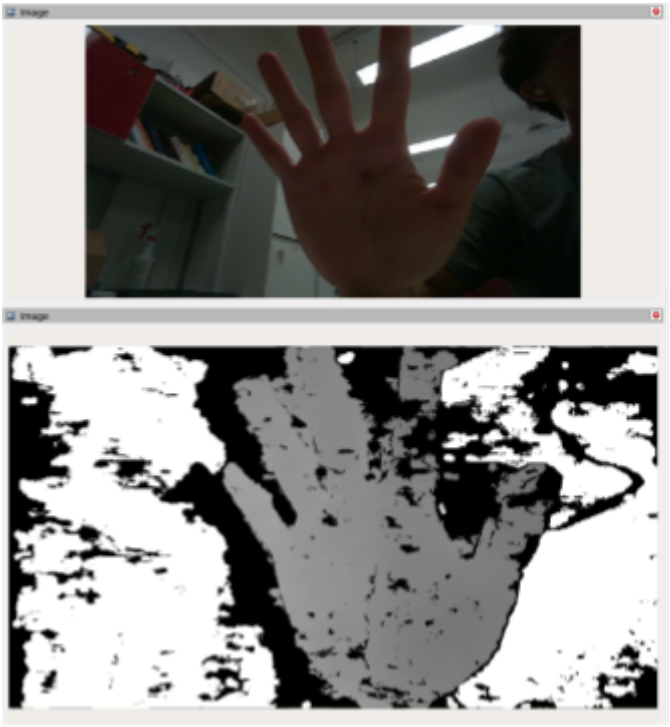

Figure 12: Intel RealSense D405 RGB & Depth Stream

Figure 13: Intel RealSense D435 RGB & Depth Stream

Figure 14: ROS Computation graph logic

Next Actions

The ROS2 subscriber can capture RGB and depth streams from the sensors. However, the method for capturing the raw depth data is still being explored. While ROSbag2 is a standard method of capturing raw data and analysis, there may be other options for deep learning analysis. Alternatively, we're exploring methods to capture the depth data into PointCloud2 - a ros topic list that can be subscribed to.

Sarah currently programs the software side of the tactile gloves in ROS1. It has yet to be bridged into ROS2 to capture the data simultaneously with the data collection from both Intel Realsense sensors - recall, Intel Realsense D405 only functions with ROS2, hence the logistical complication.

References

Chi, C., Xu, Z., Pan, C., Cousineau, E., Burchfiel, B., Feng, S., Tedrake, R., & Song, S. (2024). Universal Manipulation Interface: In-The-Wild Robot Teaching Without In-The-Wild Robots. *ArXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2402.10329

Wang, C., Shi, H., Wang, W., Zhang, R., Fei-Fei, L., & Liu, C. K. (2024, March 12). *DexCap: Scalable and Portable Mocap Data Collection System for Dexterous Manipulation*. ArXiv.org. https://doi.org/10.48550/arXiv.2403.07788

Appendix I (publisher node):

```python
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from cv_bridge import CvBridge
import cv2
import pyrealsense2 as rs
import numpy as np

class IntelPublisher(Node):
    def __init__(self):
        super().__init__("intel_publisher")
        self.intel_publisher_rgb = self.create_publisher(Image, "rgb_frame", 10)
        self.intel_publisher_depth = self.create_publisher(Image, "depth_frame", 10)

        timer_period = 0.05 # run every 0.05 secs
        self.cv_convert = CvBridge() # converts cv2 to image

        try:
            self.pipe = rs.pipeline()
            self.cfg  = rs.config()
            self.cfg.enable_stream(rs.stream.color, 640,480, rs.format.bgr8, 30) # config RGB 480p 30fps
            self.cfg.enable_stream(rs.stream.depth, 640, 480, rs.format.z16, 30) # config depth 480p 30fps
            self.pipe.start(self.cfg) # connect cam footage
            self.timer = self.create_timer(timer_period, self.timer_callback) # ??? timer ends, hits callback and push frame to topic
        except Exception as e:
            print(e)
            self.get_logger().error("INTEL REALSENSE IS NOT CONNECTED")

    def timer_callback(self):
        frames = self.pipe.wait_for_frames()
        color_frame = frames.get_color_frame() # calling RGB stream
```

```
        depth_frame = frames.get_depth_frame() # calling depth stream
        color_image = np.asanyarray(color_frame.get_data()) # RGB stream accessesed & converts ndarry
        depth_image = np.asanyarray(depth_frame.get_data())

        self.intel_publisher_rgb.publish(self.cv_convert.cv2_to_imgmsg(color_image)) # cv2 format -> RGB conversion
        self.intel_publisher_depth.publish(self.cv_convert.cv2_to_imgmsg(depth_image)) # cv2 format -> depth conversion
        self.get_logger().info("Publishing RGB & depth frame")


def main(args=None):
    rclpy.init(args=None)
    intel_publisher = IntelPublisher()
    rclpy.spin(intel_publisher)
    intel_publisher.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Appendix II (subscriber node):

Python

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image #, PointCloud2
# from sensor_msgs import point_cloud2
from cv_bridge import CvBridge
import cv2

rgb_path = r'~/home/rosh/cpt_human_demo/ros2_ws/log/rgb'
depth_path = r'~/home/rosh/cpt_human_demo/ros2_ws/log/depth'

class IntelSubscriber(Node):
    def __init__(self):
        super().__init__("intel_subscriber")
        self.subscription_rgb = self.create_subscription(Image, "rgb_frame", self.rgb_frame_callback, 10)
        self.subscription_depth = self.create_subscription(Image, "depth_frame", self.depth_frame_callback, 10)
        self.subscription_depth
        self.cv_convert = CvBridge()


    def rgb_frame_callback(self, data):
        time = self.get_clock().now()
        self.get_logger().warning("Receiving rgb frame")
        current_frame = self.cv_convert.imgmsg_to_cv2(data)
        # img = cv2.imread(current_frame)
        cv2.imwrite('rgb'+str(time)+'.png',current_frame)

        cv2.waitKey(1)

    def depth_frame_callback(self, data): # test function
```

```
        time = self.get_clock().now()
        self.get_logger().warning("Receiving depth frame")
        # self.get_logger().warning(data.get_depth_frame())
        current_frame = self.cv_convert.imgmsg_to_cv2(data)
        # img = cv2.imread(current_frame)
        cv2.imwrite('depth'+str(time)+'.png',current_frame)
        cv2.waitKey(1)

def main(args = None):
    rclpy.init(args = args)
    intel_subscriber = IntelSubscriber()
    rclpy.spin(intel_subscriber)
    intel_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == "__main":
    main()
```