GLOVELET

by

ARNAV GARG

Presented to the Faculty of the Honors College of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

HONORS BACHELOR OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2018

ACKNOWLEDGMENTS

ABSTRACT

GLOVELET

Arnav Garg, B.S. Computer Science

The University of Texas at Arlington, 2018

Faculty Mentor: Christopher Conly

The traditional computer mouse is unable to provide as high a level of productivity and efficiency in workflow for three-dimensional virtual objects as it provides for two-dimensional virtual objects. With the rise in three-dimensional virtual objects in Augmented and Virtual Reality technologies, the need for a more user-friendly and convenient user interface device is now an inescapable necessity. 'GloveLet' is a wearable glove that is designed to be used as a user interface device for three-dimensional as well as two-dimensional object movements. It is conceptualized to be lightweight, comfortable and able to be used in work for long periods of time. It has a minimalistic yet robust design. To achieve maximum reliability and simplicity, it will normalize the data to best suit the user and apply multiple filters to catch false negatives and false positives. This will allow fewer possible points of failure in the design.

TABLE OF CONTENTS

Appendix

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

Douglas Engelbart of Oregon State University developed the first computer mouse in 1964 [1]. Since its invention, the mouse has been an integral part of the computer. With time, several alternate forms of computer mouse have been created, adding additional buttons or scrolls wheels, but no variations have been successful in providing an intuitive, natural method of control at a low cost. During prolonged use of a standard computer mouse, the user's hand experiences stress. By allowing natural hand movements to control the computer's cursor action, 'GloveLet' has the potential to significantly reduce the stress on an individual's hand. 'GloveLet' allows a person to wirelessly control the mouse cursor on the computer screen via a hand-tracking glove. It allows users to interface with a computer from afar, without the need of a flat table-like surface. The traditional computer mouse requires a flat surface where the optical sensors can depict the changes to determine mouse movement on the screen [2]. 'GloveLet' can be used from a distance of four feet from the user's computer, where the hand is suspended in a floating environment, removing excessive stress on the wrist. With GloveLet, users will be able to control user interactions normally performed by a computer mouse, as well as provide new, convenient use cases not previously possible with standard user interface devices.

The development team, named 'Pied Pipers', which consisted of Arnav Garg, Joseph Tompkins, Prasoon Gautam, Sushil Bista and Ravindra Javadekar, aimed to create a wireless two-dimensional and three-dimensional object tracking glove as a part of the

Senior Design Project for the Bachelors of Science in Computer Science Program. The motivation for Pied Pipers was to build a wearable glove that can replace the traditional mouse with a more user-interactive and easy-to-use device. This would greatly enhance the user experience and would allow users to use their hands to move objects, both two-dimensional and three-dimensional, on the screen, thereby making the experience more intuitive for younger and older adults. GloveLet was built to provide one-to-one mapping between the movement of the cursor and the rotation of the virtual three-dimensional objects on the screen to the movement and rotation of the wireless glove. The current available solutions for computer mouse devices are not very intuitive for three-dimensional data on the computer. We aim to get rid of the bulkiness and difficulty in use involved in the devices currently available.

This design was developed from the ground up by the team and was designed to be lightweight, comfortable, and able to be worn for long periods without creating any discomfort. It was also ensured that the users should be able to switch freely between using GloveLet and typing on the keyboard or even switching back to a computer mouse.

GloveLet is aimed at users who use computer technology on a regular basis and wish to increase productivity and improve work flow. It will also provide a foundation for a unique gaming interface device. My team and I aimed to make the glove affordable, user-friendly, convenient and usable for a wide range of applications.

CHAPTER 2

PRELIMINARY RESEARCH

The device we built aimed to replace the traditional computer mouse; therefore, comfort, affordability and usability for a wide range of applications were accorded prime importance. To achieve error-free operation of the device, research on known algorithms for object(s) tracking in Computer Vision, Bluetooth communication for wireless data transmission and Inertial Measurement Unit (IMU) sensor data analysis was conducted. Bista and Javadekar researched on Bluetooth data transmission, Tompkins and Gautam researched the analysis and interpretation of IMU sensor data, and I surveyed published papers. Two relevant papers, titled "Tracking Color Objects in Real Time" and "A Noise Reduction Method For IMU and Its Application on Handwriting Trajectory Reconstruction", gave us significant information about color-based object(s) tracking and analysis and performance evaluation of data collected from the IMU sensor.

The paper "Tracking Color Objects in Real Time" by Vladimir Kravtchenko from the University of British Columbia discussed a clustering method that used density and spatial cues to cluster object pixels into separate objects. He proposed methods for identifying objects from the neighboring video frame and predicting their future movement. It provided details of a practical implementation of a tracking system based on the proposed techniques. The tracking system implemented in the paper tracked colored blobs. The notion of blobs as a representation for image features has a long history in computer vision and has had many different mathematical definitions. The algorithm

proposed in the paper used the pixel distribution information, namely density and spatial

cues, to cluster object pixels into separate objects/blobs. These separate objects were then

identified based on their previous history, i.e. position in the neighboring frames.

The paper titled "A Noise Reduction Method for IMU and Its Application on

Handwriting Trajectory Reconstruction" by the researchers at National Cheng Kung

University proposed a trajectory reconstruction method based on a low-cost IMU (Inertial

Measurement Unit), which is generally used in smartphones. In order to improve the

accuracy and precision of the IMU data, filtering methods were proposed to reduce high

and low frequency noises of the signal. Techniques for Attitude Estimation and Coordinate

Transformation were also discussed in the paper. Since the coordinate system of the IMU

is different from the coordinate system of the real world, the acceleration measured by the

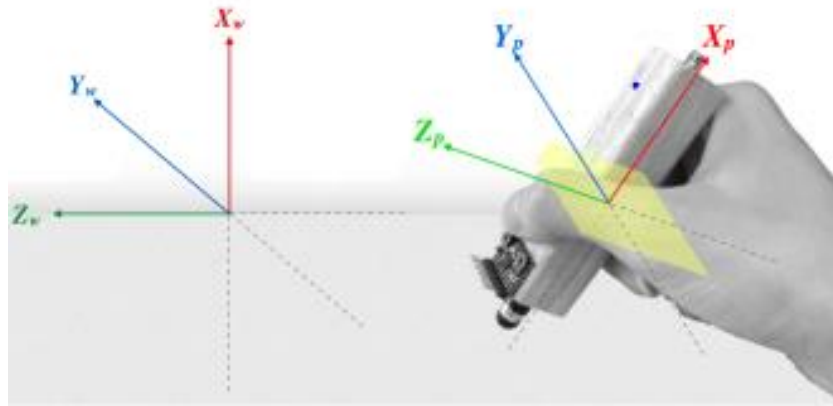accelerometer would be inconsistent with the acceleration of the real movement.

Figure 2.1: The Coordinate System of the Real World and the IMU [3]

CHAPTER 3

DESIGN

GloveLet is designed to provide a one-to-one mapping of the movement of the cursor and rotation of a virtual three-dimensional object to the movement and rotation of the wireless glove. Minimal computation is done on the microcontroller on board the glove unit. The majority of processing is performed on the computer running the user-end application. The data from the glove is transferred to the user's computer via Bluetooth transmission.

Users can perform simple gestures in order to attain the same functionality provided by a standard computer mouse as shown in Figure 2. Starting from left, the first two images show, from two different angles, the gesture needed to perform the left click functionality of a computer mouse and the next two images show the gesture required to perform the right click functionality.

Figure 3.1: Gestures to Perform Standard Computer Mouse Functionality

3.1 System Overview

The high-level structure of the GloveLet software system is divided into four layers: The Hardware/Sensor Layer, the Computer Vision Layer, Event API Layer and Application Layer. The Hardware and Sensor Layer is the physical layer, which will provide us with the IMU and flex sensor data via the Bluetooth module. The Computer Vision Layer are responsible for providing us with the user's hand tracking co-ordinates. The data from the Hardware and Sensor Layer and the Computer Vision Layer is processed in the Event API Layer where the Kalman Filter Algorithm is applied to compute the user's hand gestures and track their hand movement. The Application Layer is the top-level layer that utilizes the API functionality of the Event API layer. This layer would be responsible for directing the data to third-party applications that would use the rotational and translational data in their applications.
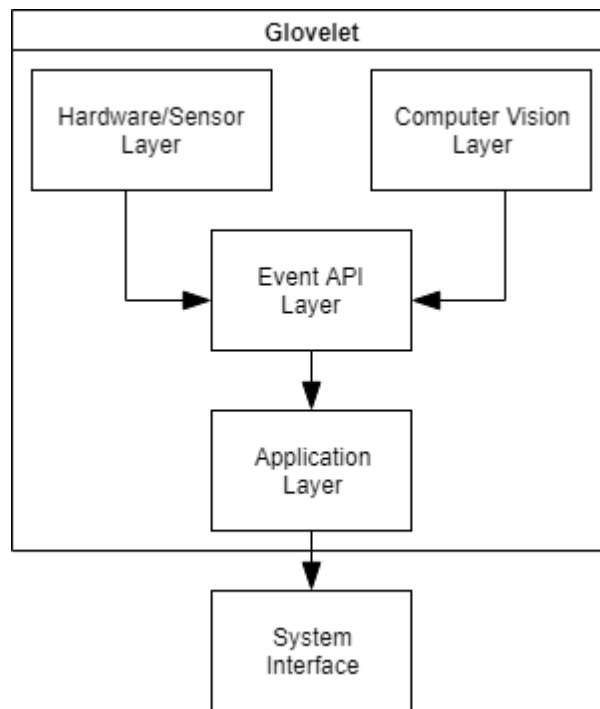
Figure 3.2: GloveLet High Level System Architecture Diagram

3.1.1 Hardware and Sensor Layer

The Hardware/Sensor layer is composed of the hardware and sensor components. Each specific sensor and hardware component is considered to be a subsystem of this layer. Data from the sensors are handled by the micro-controller, which will in turn wirelessly stream sensor data to the Event API layer on the host system for pre-processing.
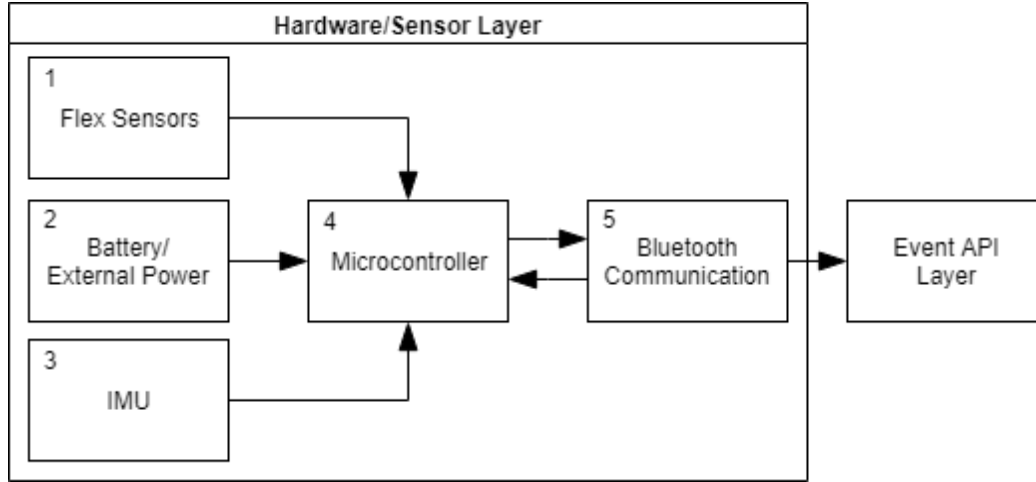
Figure 3.3: GloveLet Hardware Sensors Layer

The Hardware and Sensor Layer consists of a microcontroller, a wireless Bluetooth communication between the microcontroller and the PC, an IMU, flex sensors and a battery to power up the microcontroller.

3.1.2 Computer Vision Layer

The Computer Vision layer handles raw video stream data and runs the hand tracking algorithm on the video frames. Coordinates of the hand relative to the area of the captured video are its outputs.
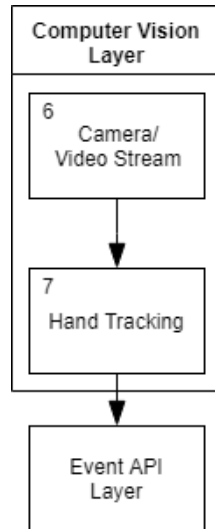
Figure 3.4: GloveLet Computer Vision Layer

The Camera/Video Stream subsystem will capture the frames from the computer's webcam and pass the data to the Hand Tracking subsystem, which will calculate the position of the hand relative to the two-dimensional coordinate frame of the computer screen. This data is then sent to the Event API Layer.

3.1.2.1 Hand Tracking Subsystem

```python
 1 def start_process(self):
 2         while True:
 3             self.read_webcam()
 4             self.threshold()
 5             self.extract_contours()
 6             if self.foundContour:
 7                 self.find_center()
 8                 self.normalize_center()
 9             x, y = self.move_cursor()
10             if cv2.waitKey(1) & 0xFF is ord('q'):
11                 break
12         cv2.destroyAllWindows()
```

Figure 3.5: Code Snippet for Hand Tracking

8

Figure 6 shows the entire processes life cycle of how a camera frame is analyzed and how the translational coordinates from the frame are extracted. Line 3 in Figure 6 calls a function to read the frame from the camera. Line 4-5 in Figure 6 is responsible for segregating the tracking color on the hand and outlining the shape of the color that is being tracked. This outlined shaped is called a contour [4]. If a contour is found, line 7-8 in Figure 6 is responsible for finding the center of the contour and normalizing it to the coordinate space of the computer. Line 9 in Figure 6 will then move the cursor on the screen to the position of the normalized center and output the x and y coordinate of the position it moved on the screen.

3.1.3 Event API Layer

The Event API layer is divided into four subsystems. These systems are Sensor Data Pre-processing, Computer Vision Data Pre-processing, Motion Data Pre-processing, and the Front-Facing Event API. Data is received from both the Hardware and Sensor layer and the Computer Vision Layer. Pre-processing of the raw data is handled at this layer, and is relayed to the Application layer via the Front-Facing Event API.
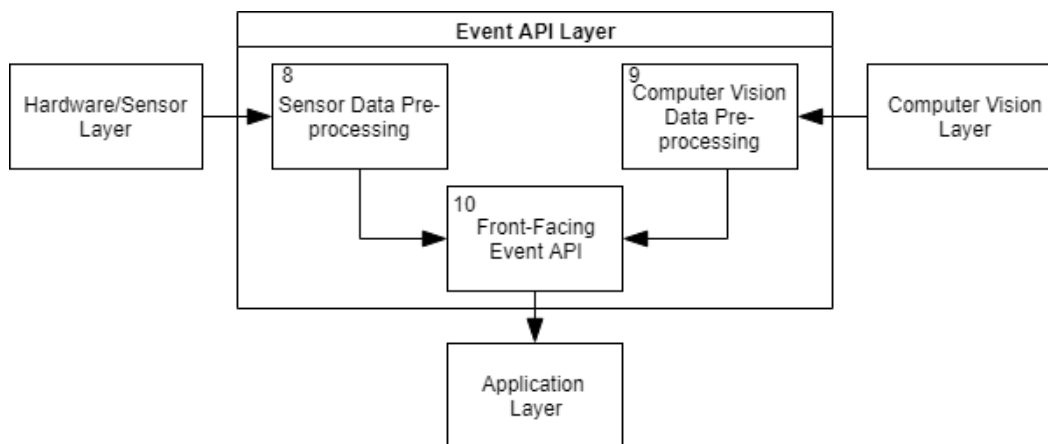
Figure 3.6: GloveLet Event API Layer

This layer is responsible for any necessary pre-processing of the raw data received from the Hardware and Sensor layer via wireless communication. The data processed here will be the IMU and flex sensor data. Pre-processing includes parsing and transforming raw data into timestamped data structures that can be used for further processing. The processed IMU data are sent to the Motion Data Pre-processing subsystem for further processing

This subsystem will also provide the system checks for whether or not the hardware has been connected to the host system. It will be responsible for establishing and maintaining wireless connection with the hardware, and ensuring that valid default data are passed if connectivity is lost.

3.1.4 Application Layer
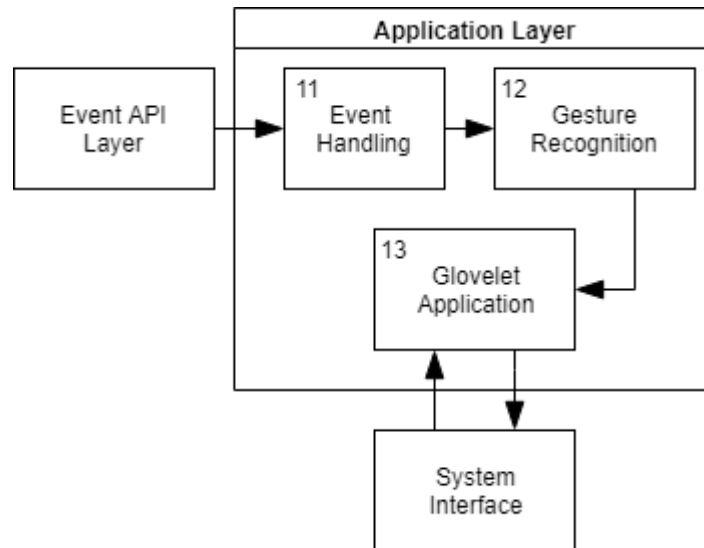
Figure 3.7: GloveLet Application Layer

The Application Layer Subsystem is the user-facing application, which receives tracking data from the Event API subsystem and translates that into computer mouse pointer or three-dimensional object movement and rotation data. It provides a user-

interface for the user to talk to the hardware and configure the application for gesture recognition and a more customized tracking.
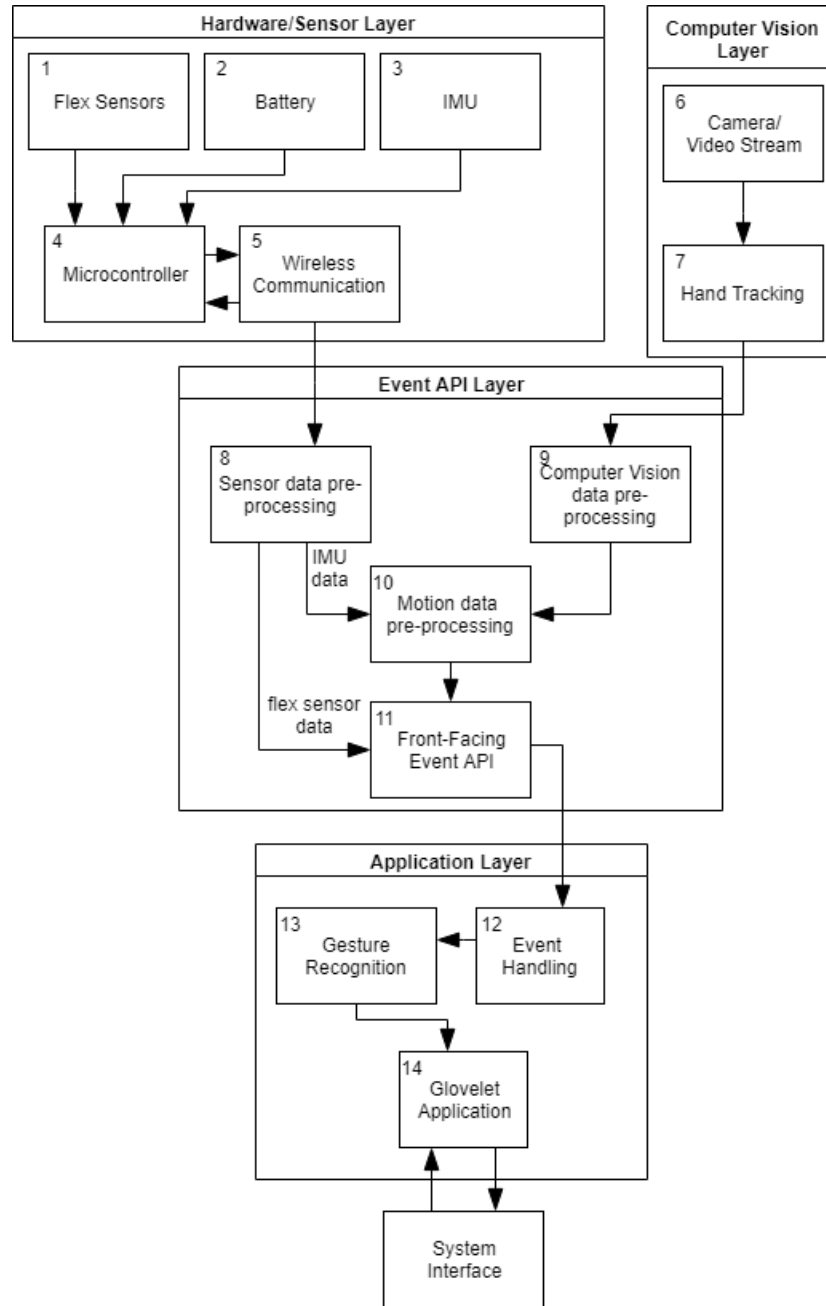
Figure 3.8: GloveLet System Data Flow

The Hardware and Sensor layer and the Computer Vision layer are the low-level layers of our system. The microcontroller in the Hardware and Sensor layer is responsible for collecting the flex sensor data for finger movement, and the IMU data for the hand rotation, and sending it to the Event API Layer via the Bluetooth Wireless Communication subsystem within the Hardware and Sensor layer. The Computer Vision layer is responsible for reading the frames from the users webcam and analyzing them to find the hands x and y coordinate. These coordinates are then sent to the Event API Layer.

The Event API Layer lies between the low-level layers (i.e. Computer Vision and Hardware and Sensor layer) and top-level layer (i.e. the Application layer) and is responsible for the data processing and communication between the two different layers. The Event API receives the data from the Hardware and Sensor layer and the Computer Vision layer in two different processes running in parallel. The two sets of data are then pre-processed in their respective processes and sent to the Application layer. The sensor data, i.e. the IMU data and the flex sensor data, are sent to the Sensor Data Pre-Processing submodule in the Event API Layer for parsing, transforming raw data into timestamped data structures and also for signal noise reduction. After the data are processed from the Sensor Data Pre-Processing subsystem, the IMU data are then sent to the Motion Data Pre-Processing subsystem, where the co-ordinate transform between the IMU's coordinate frame and Worlds coordinate frame is computed in order to get accurate glove to three-dimensional virtual object rotation mapping. The final computed flex sensor and IMU data are sent to the Front-Facing Event API, which sends the data to the Application Layer.

The Application layer is the high-level layer of our system. It collects the processed data from the Event API Layer and translates that into computer cursor movement or three-

dimensional virtual object movement and rotation. It also provides a user interface for the user to talk to the hardware and configure the application for gesture recognition and a more customized tracking.

CHAPTER 4

DISCUSSION

The integration of computer vision and sensor data was one of the primary challenges of this project. Data pre-processing methods for Computer Vision and sensor data were developed separately as subsystems. For this reason, an event-based data flow method was designed and implemented. The concept of this system was derived from commonly used mouse event APIs, where applications implement their own listener interface, which receives mouse event information via callback functions.

To increase the efficiency and performance of computation of both Computer Vision and sensor data, each was done on a separate process so that computation for both could be done in parallel on the host device. This means that GloveLet applications have a minimum of three processes running in parallel at a given time. Using multiprocessing techniques, the Computer Vision and sensor processes dispatch events in a constant stream, which are then distributed to listeners that execute a callback sub process on the main program process.

Once the GloveLet application starts, the Computer Vision and sensor sub-processes begin pre-processing data and dispatching events in parallel with the main application process. The application will then enter a main loop, where on each execution of the loop, the listeners will respond to any dispatched events. Within the listener callbacks is where mouse functionality is implemented.

In future implementations, the sub-processing and event dispatching would occur on a separate background service which would begin when GloveLet is powered on. Event APIs for GloveLet would be provided for applications to override and implement their own functionality for GloveLet. This way, Computer Vision tracking and sensor data processing is abstracted in full from functionality implementation.

<div align="center">4.1 Sensors</div>

GloveLet uses two types of sensors to capture user hand gesture and motion data. For finger gestures, flex sensors are used to capture the degree of bend in each of the fingers. For rotational data, an IMU (Inertial Measurement Unit) with accelerometer, magnetometer, and gyroscope and on-board AHRS (Attitude and Heading Reference System) is used.

4.1.1 Flex Sensors
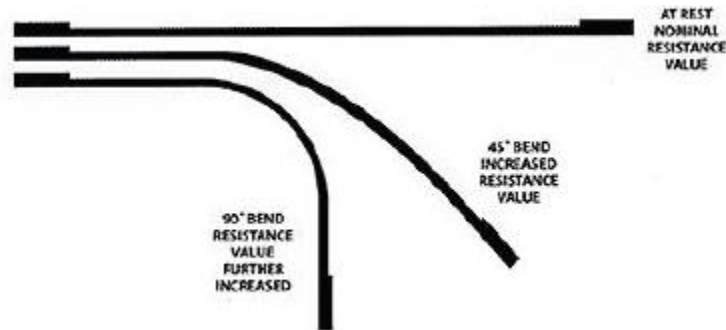
<div align="center">Figure 4.1: Flex Sensor Variable Resistance Readings [5]</div>

GloveLet uses a total of four flex sensors to capture data regarding the degree of bend of each finger. The index and middle fingers each have a 4-inch flex sensor. Two 2-inch flex sensors are attached to the thumb, one on the outer thumb and one on the inner thumb. This is due to the opposable nature of the thumb, which has a higher degree of freedom in bend motion. An analog signal is received from each of the flex sensors, and the microcontroller then executes an algorithm to clamp and normalize the readings before

<div align="center">15</div>

outputting each data sample. Flex sensor values read from the microcontroller are thus floating point numbers between zero and one, where zero is no--or minimal--bend and one is full--or maximum--bend.

For use as a mouse-type interface device, thresholds for bends are set and control flow statements are used to determine which action is being performed. For example, if the index finger exceeds a predetermined threshold, this is considered equivalent to the left mouse button being pressed down and the left-button down state is entered. Once the bend of the index finger decreases below a second predetermined threshold, this is considered equivalent to releasing the mouse button, and the left-button down state is exited.

Future methods could include machine learning for more nuanced gesture recognition, as well as training of new or custom gestures to expand the functionality of GloveLet.

4.1.2 Inertial Measurement Unit (IMU)
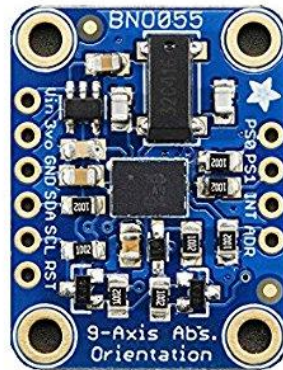
Figure 4.2: Fusion Breakout BNO055 IMU [6]

The Fusion Breakout BNO055 IMU was used for this project due to the robust embedded Sketch script libraries available for Arduino microcontrollers. These libraries provide calibrated orientation and linear acceleration data out of the box.

Orientation data provide the most use without further processing. Future applications of rotational data could be used to increase the number of possible gestures. For example, a 'thumbs-up' gesture that could be used to indicate scrolling up on a web page, versus the 'thumbs-down' gesture could be used to indicate scrolling down. Orientation of the hand can change the meaning of finger gestures.

Future methods that could be applied using accelerometer data could include machine learning over a moving time series of data samples in order to recognize some gestures, or simply to recognize whether the user is or is not moving their hand.

CHAPTER 5

CONCLUSION

The Pied Pipers have achieved the goal of creating a lightweight, affordable, comfortable and minimalistic yet robust device. The results of our project met our expectations outlined in the project proposal. Our final prototype is able to effectively control mouse movement and clicking with user hand gestures. The final prototype serves as an additional proof-of-concept to similar existing projects and we believe that this design has much room to grow.

APPENDIX A

COMPUTER VISION TRACKING CODE

```python
import numpy as np
import tkinter
import pyautogui
import math
from GloveLet.utility.timeseries import DataTimeSeries
from GloveLet.utility.motion_multiplier import motion_multiplier
import logging
from ast import literal_eval
import sys
import cv2
from GloveLet.vision.gesture import Gesture
from GloveLet.vision.gestureAPI import PreDefinedGestures
from GloveLet.eventapi.event import EventAPIException


def callback(value):
    pass


class Vision:
    WINDOW_SIZE = 4  # The window size for calculating the average
    PREV_MEMORY = 2  # Previous points stored.

    def __init__(self, default_values):
        pyautogui.FAILSAFE = False
        root = tkinter.Tk()
        root.withdraw()
        # member variables
        self.webcam = cv2.VideoCapture(0)
        self.screen_width = root.winfo_screenwidth()
        self.screen_height = root.winfo_screenheight()
        self.cameraWidth = self.screen_width / 2
        self.cameraHeight = self.screen_height / 2
        self.webcam.set(cv2.CAP_PROP_FRAME_WIDTH, self.cameraWidth)
        self.webcam.set(cv2.CAP_PROP_FRAME_HEIGHT, self.cameraHeight)
        self.output = {}
        self.handContour = {}
        self.canvas = None
        self.handMoment = {}
        self.foundContour = {}
        self.realX = {}
        self.realY = {}
        self.stationary = {}
        self.mouseX = self.screen_width/2
        self.mouseY = self.screen_height/2
        self.queue = []
        self.clickThresh = 45
        self.pinched = False
```

```python
        self.window = {}
        self.movement_history = {}
        self.record = {}
        self.boundaries = {}
        self.init_mem_vars(default_values)
        self.handMoment = (0, 0)
        self.foundContour = True
        self.stationary = False
        self.record = False
        self.realX = 0
        self.realY = 0
        self.movement_history = []
        self.window = DataTimeSeries(
            self.WINDOW_SIZE, 2, auto_filter=True)
        self.init_gestures()

    def init_mem_vars(self, default_values):
        if not default_values:
            with open('.vision.config', 'w') as file:
                value = self.find_range()
                self.boundaries = value
                file.write('{}\n'.format(str(value)))
        else:
            try:
                with open('.vision.config', 'r') as file:
                    for line in file:
                        values = literal_eval(line)
                        self.boundaries = values
            except IOError:
                print('Config file not found. Run the program with -r flag.')
                sys, exit()
            except Exception:
                print('Not all the fingers have colors configured. Run with -r flag')
                sys.exit()

    def find_range(self):
        range_filter = 'HSV'
        cv2.namedWindow("Trackbar", 0)
        for i in ["MIN", "MAX"]:
            v = 0 if i == "MIN" else 255
            for j in range_filter:
                cv2.createTrackbar("%s_%s" %
                                   (j, i), "Trackbar", v, 255, callback)
        while True:
            _, image = self.webcam.read()
            image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

```python
        values = []
        for i in ["MIN", "MAX"]:
            for j in range_filter:
                v = cv2.getTrackbarPos("%s_%s" % (j, i), "Trackbar")
                values.append(v)
        lower = (values[:3])
        upper = (values[3:])
        thresh = cv2.inRange(image, tuple(lower), tuple(upper))
        # layout = np.vstack((thresh, image))
        cv2.imshow('Thresh', thresh)
        # cv2.imshow('Image', image)
        if cv2.waitKey(1) & 0xFF is ord('q'):
            cv2.destroyAllWindows()
            return tuple([lower, upper])

    def read_webcam(self):
        _, self.frame = self.webcam.read()
        self.frame = cv2.flip(self.frame, 1)
        # for finger in self.ACTIVE_FINGERS:
        self.canvas = np.zeros(self.frame.shape, np.uint8)
        self.frame = cv2.cvtColor(self.frame, cv2.COLOR_BGR2HSV)

    def threshold(self):
        (lower, upper) = self.boundaries
        lower = np.array(lower, dtype="uint8")
        upper = np.array(upper, dtype="uint8")
        mask = cv2.inRange(self.frame, lower, upper)
        kernel = np.ones((5, 5), np.uint8)
        self.output = cv2.bitwise_and(
            self.frame, self.frame, mask=mask)
        self.output = cv2.cvtColor(
            self.output, cv2.COLOR_BGR2GRAY)
        self.output = cv2.erode(
            self.output, kernel, iterations=1)
        self.output = cv2.dilate(
            self.output, kernel, iterations=3)

    def extract_contours(self):
        _, self.contours, _ = cv2.findContours(
            self.output.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
        maxArea, idx = 0, 0
        if len(self.contours) == 0:
            self.foundContour = False
            return
        else:
            self.foundContour = True
```

```python
        for i in range(len(self.contours)):
            area = cv2.contourArea(self.contours[i])
            if area > maxArea:
                maxArea = area
                idx = i
        self.realHandContour = self.contours[idx]
        self.realHandLength = cv2.arcLength(self.realHandContour, True)
        self.handContour = cv2.approxPolyDP(
            self.realHandContour, 0.001 * self.realHandLength, True)

    def __check_stationary(self):
        search_len = 3
        val = -1 * (search_len + 1)
        self.prev_record_state = self.record
        if self.can_do_gesture:
            xPoints = [pt[0] for pt in self.movement_history[val:-1]]
            yPoints = [pt[1] for pt in self.movement_history[val:-1]]
            xAvg = np.average(xPoints)
            yAvg = np.average(yPoints)
            factor = 0.04
            for [x, y] in self.movement_history[-(search_len + 1):-1]:
                if (x-xAvg)**2 + (y-yAvg) > factor * \
                                  min(self.cameraWidth, self.cameraHeight):
                    if self.stationary:
                        self.record = True
                    self.stationary = False
                    return
            if not self.stationary:
                self.record = False
            self.stationary = True

    def find_center(self):
        self.moments = cv2.moments(self.handContour)
        if self.moments["m00"] != 0:
            self.handX = int(self.moments["m10"] / self.moments["m00"])
            self.handY = int(self.moments["m01"] / self.moments["m00"])
            self.handMoment = (self.handX, self.handY)

    def normalize_center(self):
        self.window.add(self.handMoment)
        self.realX, self.realY = self.window[0]
        # print('{}'.format(self.window.timestamp[0]))
        self.movement_history += [(self.realX, self.realY)]
        self.__check_stationary()
```

```python
def move_cursor(self):
    x = self.realX * (self.screen_width / self.frame.shape[1])
    y = self.realY * (self.screen_height / self.frame.shape[0])
    return (x, y)

def start_process(self):
    """start_process
    This is where all the functions for tracking the fingers and
    movement are called. This is the master function.
    """
    while True:
        self.read_webcam()
        self.threshold()
        self.extract_contours()
        if self.foundContour:
            self.find_center()
            self.normalize_center()
        else:
            self.stationary = True
        self.draw()
        self.frame_outputs()
        self.check_can_perform_gesture()
        self.determine_if_gesture()
        x, y = self.move_cursor()
        # Exit out of this hell hole.
        if cv2.waitKey(1) & 0xFF is ord('q'):
            break
    cv2.destroyAllWindows()


vision = Vision()
vision.start_process()
```
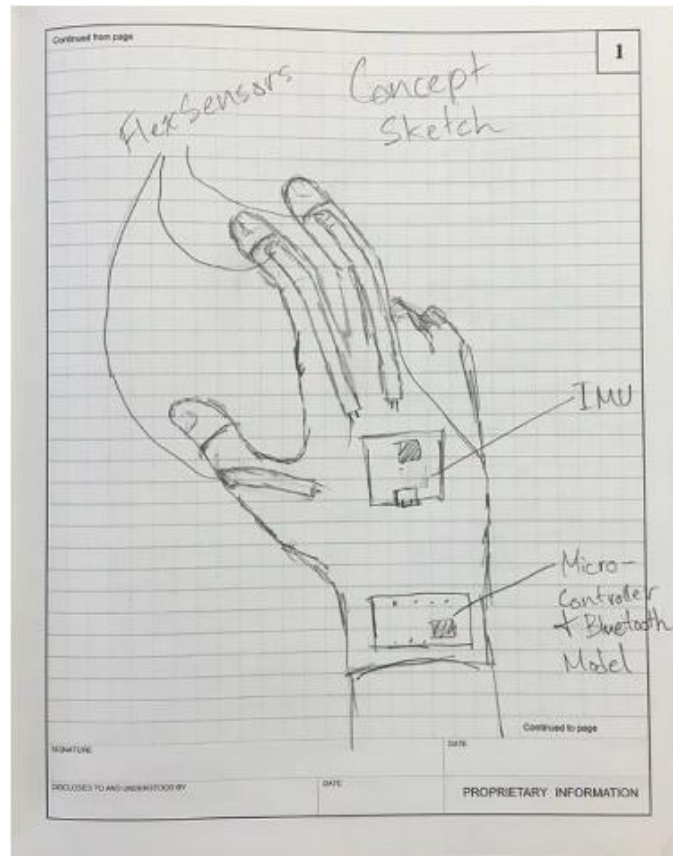
APPENDIX B

INITIAL AND FINAL DESIGN

Figure B.1: Initial GloveLet Conceptual Design

Figure B.2: Final GloveLet Design

REFERENCES

[1] *"*"The Centre for Computing History." *Centre For Computing History*,

www.computinghistory.org.uk/det/613/the-history-of-the-computer-mouse/.

[2] Binu. "Optical Mouse – Principle and Working." *IT Blogs*, 11 Aug. 2009,

itblogs.in/computers/hardware/optical-mouse-principle-and-working/.

[3] Pan, Tse-Yu, et al. "A Noise Reduction Method for IMU and Its Application on

Handwriting Trajectory Reconstruction." *2016 IEEE International Conference on*

Multimedia & Expo Workshops (ICMEW), 2016,

doi:10.1109/icmew.2016.7574685.

[4] "Contour Approximation Method." *OpenCV: Image Thresholding*,

docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html.

[5] "Method to Interface and Use Flexibend Sensor." *Tutorial by Cytron*, 10 Aug.

2010, tutorial.cytron.io/2012/08/10/method-to-interface-and-use-flexibend-

sensor/.

[6] Townsend, Kevin. "Adafruit BNO055 Absolute Orientation Sensor." *Power*

Usage | Adafruit Motor Shield | Adafruit Learning System, 22 Apr. 2015,

learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/overview.

BIOGRAPHICAL INFORMATION

Arnav Garg will graduate in May 2018 with an Honors Bachelor of Science in Computer Science from the University of Texas at Arlington (UTA). He plans to conduct research in graduate school, ideally in Computer Vision and Deep Learning.

Garg has worked as a Software Engineering Intern in two startups, Nod Labs in Mountain View, CA, and Cloud 9 Perception in Arlington, TX, where he assisted in devising various Computer Vision algorithms. He has also served as an Undergraduate Research Assistant at the Heracleia Human Interaction Lab at UTA.

.