EEL4924C Senior Design - January 15, 2020

Electrical and Computer Engineering Department

Final Design Report:

Air Canvas

Two Guys Three Dimensions

Made by:
Caleb Robey, Evan Gruda

Contents

1 Abstract

For our project we designed a 3 dimensional real time drawing application. We designed our own stereo vision camera that we use to track an object in 3D space. We then us the coordinates to create real time 3D renderings. The stereo camera is made by combining the real time capture of two cameras and combining them into a stereo image. The camera feed will be processed by a broadcom system on module board connected via PCIE to our custom PCB. It will then go through image recognition via a Python program utilizing an OpenCV library. This will then display the drawing as a 3D object. We worked with an open-source software company Dust3D to help with our cloud meshing algorithm that produced the .OBJ file. We have created a controlled space so that the user's hand will be the only visual to the camera feed. The user also has a control board that allows them to make customization's to their drawing.

2 Introduction

3D drawing is a practical tool in both industry and academia. In the industry space, a user could draw and highlight their surroundings as they show broken machinery to remote engineers. In academia, 3D drawing can be used for a whole variety of visualization assignments that help students understand multidimensional math, art, and science.
Our design will allow for 3D drawing in a controlled environment. The user will be able to use their hand as a pencil in the drawing space that will be detected by 3D cameras and let the space serve as the canvas. The user will also be able to erase and rotate using a variety of controls that we will implement. If time permits, we will also implement some gesture recognition to allow more fluid user control.

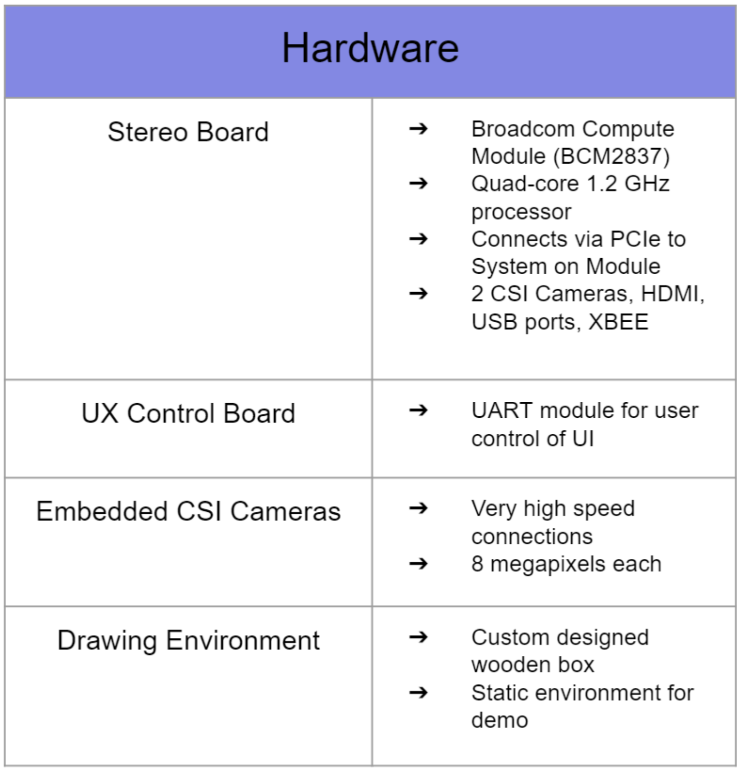
3 Technical Objectives

The primary technical objective of this project is to remotely sense and track an object in 3D space to create real time 3D drawings.
From the hardware perspective, we will use two cameras to create a stereo vision system. To enable a sufficient streaming rate, these will be attached using CSI connectors. The PCB will have differential routing to ensure timing and entegrity of signals. CSI connections allow for frame rates far higher than those of USB cameras. The CSI connections will then port to a PCIE DDR2 connection to the broadcom SOM that we will be using. The SOM will begin processing the captures in GPU and send rendered images via DMA to our CPU.
Once we have both of the images, they need to be rectified using a least-squares algorithm, which will subtract both images down to their common subsets. Then the algorithm will have both of the angles to each pixel and the distance between the two cameras. This allows for a distance to be determined using a triangulation algorithm.
Furthermore, color patterns in the HSV colorspace will be exploited to detect a specifically colored glove in a fairly static environment. Once the correct color contour is detected, it's centroid will be used as our 2D touch point for analysis.
Once this disparity map is rendered and the point is found, we will use this information to create a 3D

point to send to the cloud meshing algorithm. This will be done by referencing the 2D point against the disparity map resulting in a 3D point for use.
After receiving the stream of 3D coordinates we wait for the user to interact with the Control Board to decide what to do with the current data. If the user decided to draw, the coordinates will be fed over TCP/IP to a local machine running Dust3D 's cloud meshing algorithm. Dust3D will write back over TCP/IP the .OBJ file. Another process is used to to display and rotate the .OBJ file for the user to see and manipulate.

3.1 Hardware

The Hardware we plan to use is described in the table and sections below:



Hardware	
Stereo Board	➔ Broadcom Compute Module (BCM2837) ➔ Quad-core 1.2 GHz processor ➔ Connects via PCIe to System on Module ➔ 2 CSI Cameras, HDMI, USB ports, XBEE
UX Control Board	➔ UART module for user control of UI
Embedded CSI Cameras	➔ Very high speed connections ➔ 8 megapixels each
Drawing Environment	➔ Custom designed wooden box ➔ Static environment for demo

Figure 1: Hardware Summary

3.1.1 Stereo Board

The primary processor selected for our project is the Broadcom Compute Module. We chose this due to its fast processing ability and its ability to stream dual cameras. The processing ability will be help us in our computer vision aspect of our project. The dual cameras are needed for the stereo vision application of our project.

- **Stereo Board**

  - Broadcom Compute Module (BCM2837)
    - Quad-core 1.2 GHz processor
    - 1Gbyte LPDDR2 RAM
    - 32GB onboard eMMC flash
    - Latest Linux Kernel support
    - 54 accessible GPIO via PCIe
    - 2x I2C
    - 2x SPI
    - 2x UART
    - 2x SD/SDIO
    - 1x HDMI 1.3a
    - 1x USB2 HOST/OTG
    - 1x DPI (Parallel RGB Display)
    - 1x NAND interface (SMI)
    - 1x 4-lane CSI Camera Interface (up to 1Gbps per lane)
    - 1x 2-lane CSI Camera Interface (up to 1Gbps per lane)
    - 1x 4-lane DSI Display Interface (up to 1Gbps per lane)
    - 1x 2-lane DSI Display Interface (up to 1Gbps per lane)
  - High-speed CSI connectors

    2 Sony Exmor IMX219 CSI Cameras
  - Xbee Backpack
  - USB 2.0 ports

    FSUSB42UMX controller IC
  - HDMI out

    ESD5384 IC

    AP2331W Regulator
  - Low-dropout regulator

    AP7115-25SEG-7
  - Voltage regulator (5V to 3.3V and 1.8V)

    RT8020AGQW
  - GPIO breakout

3.1.2 UX Control Board

The UX control board is designed to allow the user to have more control over the application experience. The UX board will consist of an analog joystick and several push buttons to allow the user to switch between different drawing modes. The board will communicate with the stereo board via the Xbee RF module.

- UX Control Board

  - Atmel ATmega16A
  - Xbee backpack
  - COM-09032 Analog Joystick
  - Digital push buttons
  - Battery Power Supply (9v)
    3.3v regulator

3.1.3 Embedded CSI Cameras

CSI cameras allow us to stream at much higher frame rates compared to USB cameras. The SOM allows us to connect the CSI cameras directly to the GPU on chip to allow for faster processing without taking up CPU time.

- Sony Exmor IMX219 CSI Cameras

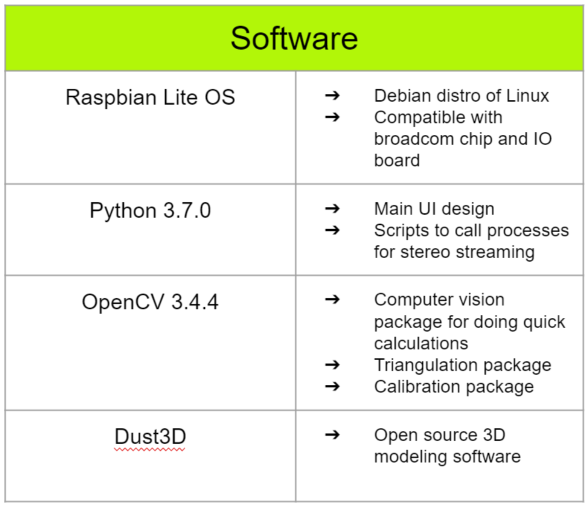
  8 Megapixels

  4Kp @30 FPS, 1080p @60FPS, 720p @180FPS

3.1.4 Demo Environment

Since the end goal of this project is to draw in 3D space, we will also be constructing a static demo environment to keep the cameras at the correct distance apart and give a background without excessive noise. This will be constructed as a 2'x2'x2' box, and we will do the custom woodwork ourselves.

3.2 Software



Software	
Raspbian Lite OS	➔ Debian distro of Linux ➔ Compatible with broadcom chip and IO board
Python 3.7.0	➔ Main UI design ➔ Scripts to call processes for stereo streaming
OpenCV 3.4.4	➔ Computer vision package for doing quick calculations ➔ Triangulation package ➔ Calibration package
Dust3D	➔ Open source 3D modeling software

Figure 2: Software Summary

3.2.1 Linux Raspian Lite Os

- Strengths

  Built by Raspberry Pi foundation specifically for embedded broadcom processors

  Supports protocol for dual cameras

  Consumes less than 2GB of storage

- Weaknesses

  No desktop interface

  No cloud integration

We also considered an RTOS and the Windows IOT OS. While Windows IOT had more features than Raspbian out of the box, it was not as easily customizable and bulkier than we would prefer. We did not use an RTOS because the real-time requirements of our project were not too stringent for a normal OS, and still wanted the drivers and multi-tasking associated with Raspbian.

3.2.2 Python 3.7.0

- Strengths

  Library support for user interfaces

  Support for drawing applications

- Weaknesses

  Processing time can be exponentially slower than a lower level language

3.2.3 OpenCV 3.4.4

- Strengths

  Best open source library for using a camera

  Strong analytical libraries for stereo calculations

3.2.4 Dust3D

- Strengths

  Open Source

  Creators willing to work with us to help create new API

- Weaknesses

  Cloud meshing requires heavy processing

  Requires us to run on a separate device

4 Implementation

4.1 Hardware

4.1.1 Stereo Board

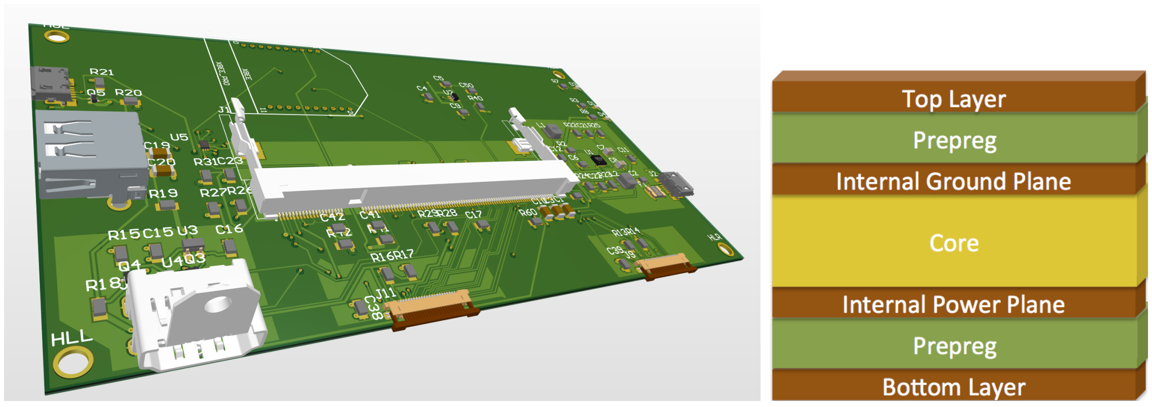


Figure 3: Final Stereo Board

- **4 Layer Board**

  Layer 1 : Top Signal Layer (100Ω R Traces)

  Layer 2 : Internal Ground Plane

  Layer 3 : Internal Power Plane

  Layer 4 : Bottom Signal Layer (100Ω R Traces)

  We decided to go with a four layer board to maximize our ability to control power stability throughout the board. The adding of the Ground and Power planes help us with crucial timing during the boot sequence

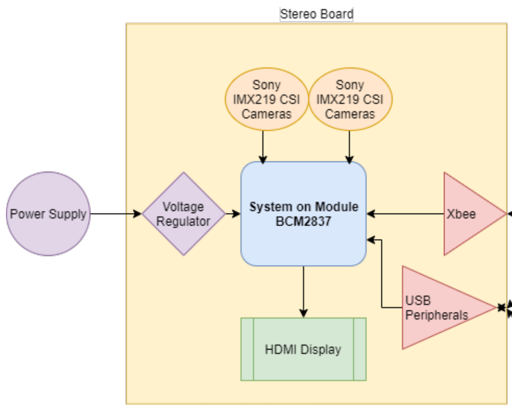
- **Differential Routing**

  The In order to keep signal integrity for the CSI Cameras and the HDMI we needed to use differential routing. This helps with timing issues that occur during high speed signal transfer.

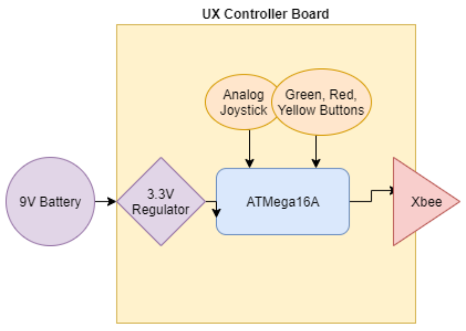
- **USB**

  We elected for an external USB HUB connected to our single USB on board peripheral. This will allow for a user to add additional devices without needing to upgrade hardware.

4.1.2 Diagrams



(a) Stereo Board Diagram



(b) UX Board Diagram

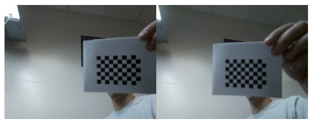
Figure 4: Hardware Diagrams

4.2 Software

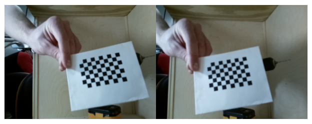
4.2.1 Stereo Vision Algorithm

- **Calibration**

  In order to implement stereo vision, one must calibrate the two cameras to each other. This is necessary due to minor manufacturing defaults in every camera.



(a) Scene in Noisy Environment



(b) Scene in Controlled Environment

Figure 5: Scene images

These defects form a matrix that can be calculated using an OpenCV algorithm run on a collection of images of checkerboards as shown below:

These images of checkerboards were used to determine the intrinsic parameters. These were the radial and tangential distortion coefficients, focal length, and image centers. The extrinsic parameters determined were the translation, rotation, fundamental, and essential matrices used to determine the position of a pixel in the right camera relative to the left camera. The following are the resulting images after stereo camera rectification:

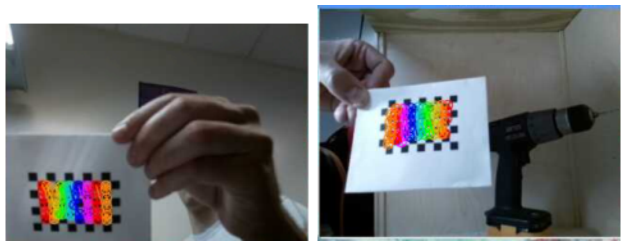

(a) Captured in Noisy Environment (b) Captured in Controlled Environment

Figure 6: Captured images

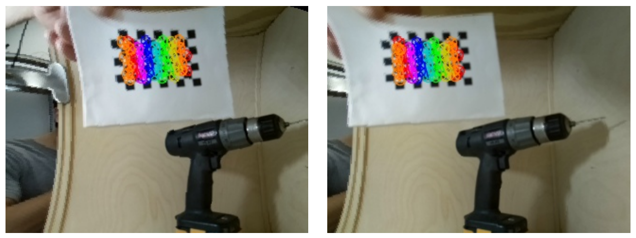

(a) Rectified Left (b) Rectified Right

Figure 7: Rectified images

- **Parameter Adjustment**

  Based on the distortion parameters received in the calibration step, the stereo correspondence algorithm could still be adjusted using certain parameters: The minimum number of disparities to be found in the stereo map, the expected number of disparities, and the uniqueness ratio.

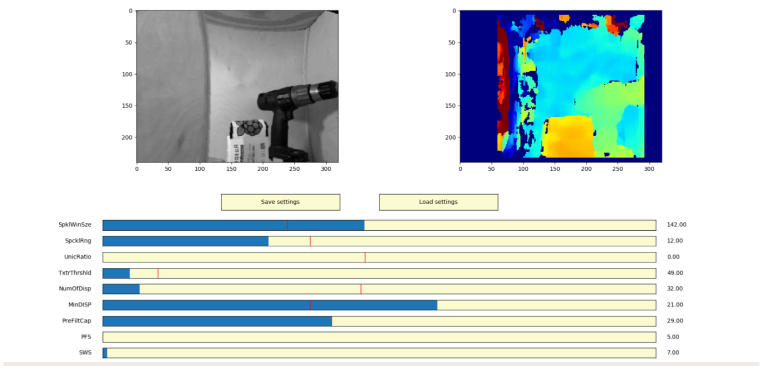
  The number of disparities dictates the resolution of your depth map. For example, if you assume 30 disparities in the image, there are 30 different depths at which a point could reside.

  The minimum number of disparities is the "closest" that two points could be to the camera. This is proportional to the distance between the cameras. The further apart the cameras are, the higher this number needs to be.

  The uniqueness ratio dictates the threshold of confidence for a value to be rendered in the final disparity map. For this application, a uniqueness ratio of 1 was chosen (lowest possible value) because a densely populated map was more important than computational confidence due to the static environment.

- **Hand Detection**

  Hand detection was pre-calibrated for the demo with a blue glove. This was done by programming a script to focus on the center region of the camera stream. It then took a photo on the users command and extracted the color histogram data from the middle of the photo.



SpklWinSze 142.00
SpcklRng 12.00
UnicRatio 0.00
TxtrThrshld 49.00
NumOfDisp 32.00
MinDISP 21.00
PreFiltCap 29.00
PFS 5.00
SWS 7.00

Figure 8: Result of parameter adjustment

This histogram data in the case of the demo was a blue glove. Whenever hand detection was desired by the parent process, it would call an algorithm that grabbed each contour similar to the color histogram of the glove. Finding the maximum sized contour, it would then calculate centroid of that contour as the detection point.

- **Livestreaming**

  Though examples are plentiful of using stereo vision for a single shot camera, there are far fewer applications found in real-time streaming. This is largely due to the computational complexity of analyzing a single frame.

  In order to make the livestreaming work, we adjusted parameters so that the computation of the depth map takes approximately 0.2 seconds. Although this is not quite the desired speed for video streaming, it is good enough for a live demo.

4.2.2 Cloud Meshing with Dust3D

- **TCP/IP Protocol**

  We teamed up with the Dust3D team to create a remote API that allowed anyone to connect the application over port 53309. We created a list of protocols that would allow someone to use their cloud meshing algorithms by simply sending over (x,y,z) coordinates and how to connect nodes.

- **Python Library**

  After having this API we created a Python3 Library and Class that makes it easy for anyone to call and use the commands.

  In order to have solid communication we have a single Thread that is continuously listening for replies from Dust3D. This is how we are able to get valid Export Object files.

4.2.3 Parallel Processing

- The BCM 2837 is a quad core processor. To maximize our resources we used multiprocessing libraries to avoid Python's Global Lock to allow our program to access multiple cores and achieve real parallelism. For processes that needed better sharing of resources and can run in pseudo-parallelism we utilized Python's threading library.
  Our two biggest processes our the OpenCv scripts and the 3D modeling scripts. This situation is a classic Consume Producer Problem

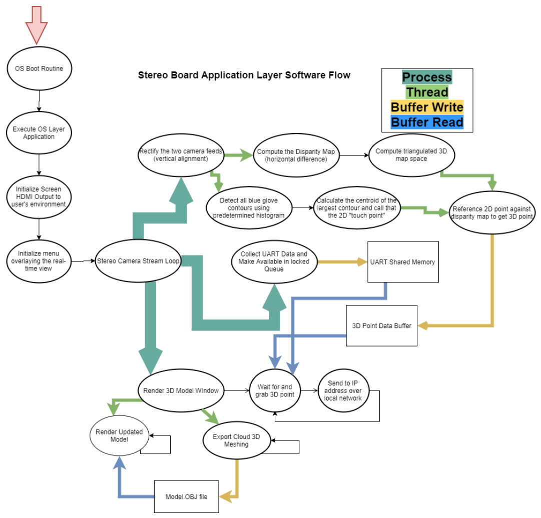
- **Producer**

  Our producer is the OpenCV scripts that are generating the (x,y,z) coordinates. This coordinates will be put into a shared Queue as they are generated.

- **Consumer**

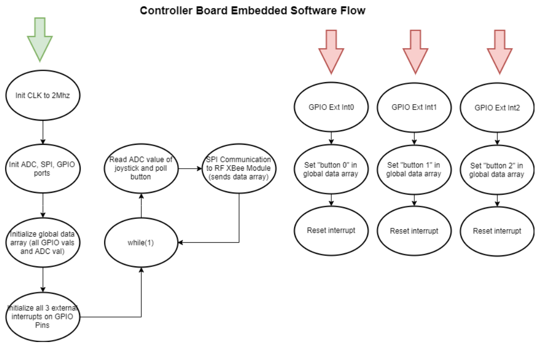
  Our consumer is the 3D modeling process that is taking the (x,y,z) coordinates and then possibly mapping them onto our current drawing. We are also utilizing threading to share our socket to the 3D modeling protocols.

4.2.4 Diagrams



Stereo Board Application Layer Software Flow
Process
Thread
Buffer Write
Buffer Read
OS Boot Routine
Execute OS Layer Application
Initialize Screen HDMI Output to user's environment
Initialize menu overlaying the real-time view
Stereo Camera Stream Loop
Rectify the two camera feeds (vertical alignment)
Compute the Disparity Map (horizontal difference)
Compute triangulated 3D map space
Detect all blue glove contours using predetermined histogram
Calculate the centroid of the largest contour and call that the 2D "touch point"
Reference 2D point against disparity map to get 3D point
Collect UART Data and Make Available in locked Queue
UART Shared Memory
3D Point Data Buffer
Render 3D Model Window
Wait for and grab 3D point
Send to IP address over local network
Render Updated Model
Export Cloud 3D Meshing
Model.OBJ file

(a) Stereo Board Software Diagram



Controller Board Embedded Software Flow
Init CLK to 2Mhz
Init ADC, SPI, GPIO ports
Initialize global data array (all GPIO vals and ADC val)
Initialize all 3 external interrupts on GPIO Pins
while(1)
Read ADC value of joystick and poll button
SPI Communication to RF XBee Module (sends data array)
GPIO Ext Int0
GPIO Ext Int1
GPIO Ext Int2
Set "button 0" in global data array
Set "button 1" in global data array
Set "button 2" in global data array
Reset interrupt
Reset interrupt
Reset interrupt

(b) UX Board Software Diagram

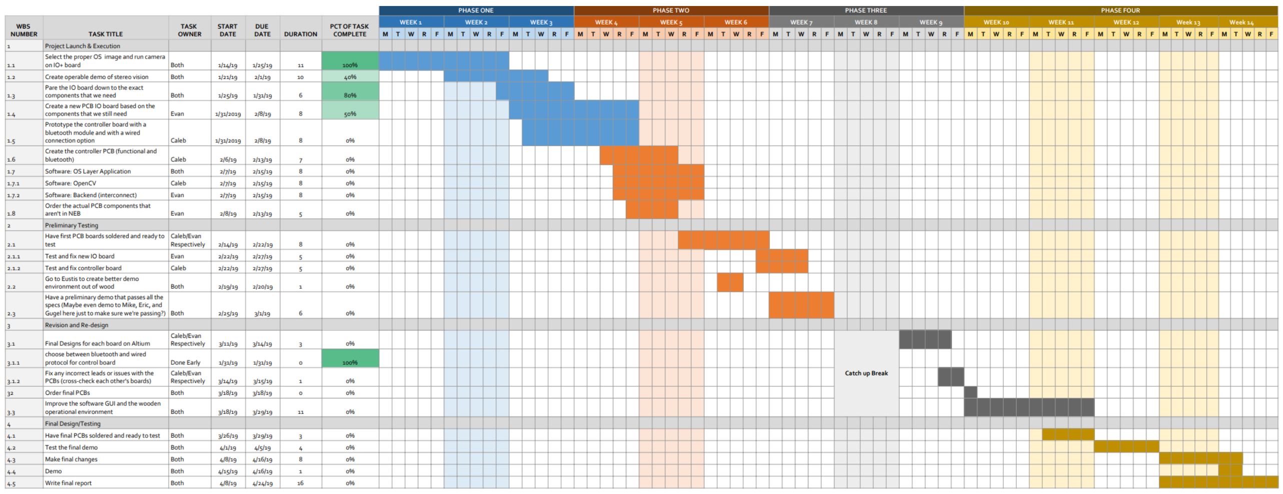
Figure 9: Software Diagrams

5 Team Work Division

5.1 Task Chart

Task	Member
Design Stereo Board	Evan
Design UX Board	Caleb
Create operable demo on evaluation boards	Team
Order parts	Team
Computer vision	Caleb
Stereo vision	Caleb
3D application	Evan
Multiprocessing and Threading	Evan
Modify and perfect Stereo Board	Evan
Modify and perfect UX board	Caleb
Create custom box for live demo	Team
Integrate software and hardware for Demo	Team
Implement backend software design	Team
Prepare for demo	Team
Final Report	Team

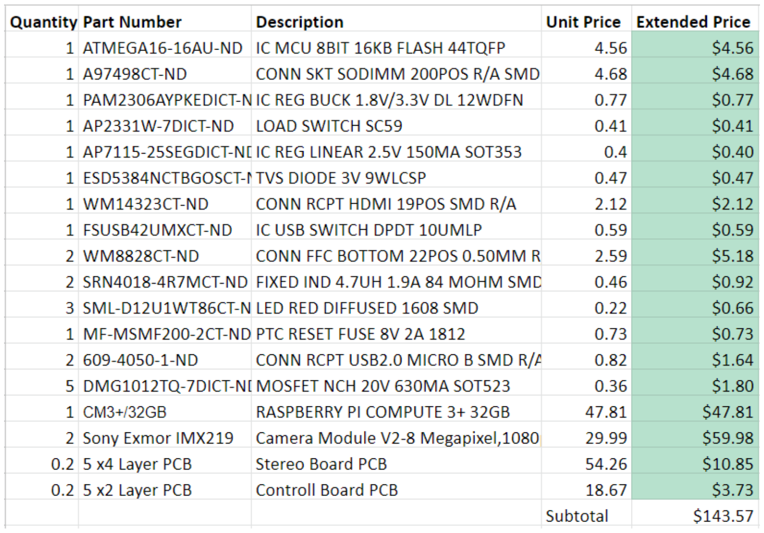
5.2 GANTT Chart



WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	Project Launch & Execution					
1.1	Select the proper OS image and run camera on IO+ board	Both	1/14/19	1/25/19	11	100%
1.2	Create operable demo of stereo vision	Both	1/21/19	2/1/19	10	40%
1.3	Pare the IO board down to the exact components that we need	Both	1/25/19	1/31/19	6	80%
1.4	Create a new PCB IO board based on the components that we still need	Evan	1/31/2019	2/8/19	8	50%
1.5	Prototype the controller board with a bluetooth module and with a wired connection option	Caleb	1/31/2019	2/8/19	8	0%
1.6	Create the controller PCB (functional and bluetooth)	Caleb	2/6/19	2/13/19	7	0%
1.7	Software: OS Layer Application	Both	2/7/19	2/15/19	8	0%
1.7.1	Software: OpenCV	Caleb	2/7/19	2/15/19	8	0%
1.7.2	Software: Backend (interconnect)	Evan	2/7/19	2/15/19	8	0%
1.8	Order the actual PCB components that aren't in NEB	Evan	2/8/19	2/13/19	5	0%
2	Preliminary Testing					
2.1	Have first PCB boards soldered and ready to test	Caleb/Evan Respectively	2/14/19	2/22/19	8	0%
2.1.1	Test and fix new IO board	Evan	2/22/19	2/27/19	5	0%
2.1.2	Test and fix controller board	Caleb	2/22/19	2/27/19	5	0%
2.2	Go to Eustis to create better demo environment out of wood	Both	2/19/19	2/20/19	1	0%
2.3	Have a preliminary demo that passes all the specs (Maybe even demo to Mike, Eric, and Gugel here just to make sure we're passing?)	Both	2/25/19	3/1/19	6	0%
3	Revision and Re-design					
3.1	Final Designs for each board on Altium	Caleb/Evan Respectively	3/11/19	3/14/19	3	0%
3.1.1	choose between bluetooth and wired protocol for control board	Done Early	1/31/19	1/31/19	0	100%
3.1.2	Fix any incorrect leads or issues with the PCBs (cross-check each other's boards)	Caleb/Evan Respectively	3/14/19	3/15/19	1	0%
32	Order final PCBs	Both	3/18/19	3/18/19	0	0%
3.3	Improve the software GUI and the wooden operational environment	Both	3/18/19	3/29/19	11	0%
4	Final Design/Testing					
4.1	Have final PCBs soldered and ready to test	Both	3/26/19	3/29/19	3	0%
4.2	Test the final demo	Both	4/1/19	4/5/19	4	0%
4.3	Make final changes	Both	4/8/19	4/16/19	8	0%
4.4	Demo	Both	4/15/19	4/16/19	1	0%
4.5	Write final report	Both	4/8/19	4/24/19	16	0%

Figure 10: GANTT Chart

6 Bill of Materials



Quantity	Part Number	Description	Unit Price	Extended Price
1	ATMEGA16-16AU-ND	IC MCU 8BIT 16KB FLASH 44TQFP	4.56	$4.56
1	A97498CT-ND	CONN SKT SODIMM 200POS R/A SMD	4.68	$4.68
1	PAM2306AYPKEDICT-N	IC REG BUCK 1.8V/3.3V DL 12WDFN	0.77	$0.77
1	AP2331W-7DICT-ND	LOAD SWITCH SC59	0.41	$0.41
1	AP7115-25SEGDICT-NI	IC REG LINEAR 2.5V 150MA SOT353	0.4	$0.40
1	ESD5384NCTBGOSCT-I	TVS DIODE 3V 9WLCSP	0.47	$0.47
1	WM14323CT-ND	CONN RCPT HDMI 19POS SMD R/A	2.12	$2.12
1	FSUSB42UMXCT-ND	IC USB SWITCH DPDT 10UMLP	0.59	$0.59
2	WM8828CT-ND	CONN FFC BOTTOM 22POS 0.50MM R	2.59	$5.18
2	SRN4018-4R7MCT-ND	FIXED IND 4.7UH 1.9A 84 MOHM SMD	0.46	$0.92
3	SML-D12U1WT86CT-N	LED RED DIFFUSED 1608 SMD	0.22	$0.66
1	MF-MSMF200-2CT-ND	PTC RESET FUSE 8V 2A 1812	0.73	$0.73
2	609-4050-1-ND	CONN RCPT USB2.0 MICRO B SMD R/A	0.82	$1.64
5	DMG1012TQ-7DICT-NI	MOSFET NCH 20V 630MA SOT523	0.36	$1.80
1	CM3+/32GB	RASPBERRY PI COMPUTE 3+ 32GB	47.81	$47.81
2	Sony Exmor IMX219	Camera Module V2-8 Megapixel,1080	29.99	$59.98
0.2	5 x4 Layer PCB	Stereo Board PCB	54.26	$10.85
0.2	5 x2 Layer PCB	Controll Board PCB	18.67	$3.73
			Subtotal	$143.57

Figure 11: Bill of Materials

7 Future Applications

- Industry

  This application can be used for quick 3D modeling in a meeting or conference when ideas need to be rendered without complicated software packages.

  Furthermore, depth maps work great in noisy environments, so this could be extended to actually drawing in the 3D space of the user as long as some sort of drawing point detection could still be obtained.

- Academia

  Explaining and understanding 3D spaces is difficult for many students. Being able to render a drawing and see its curvature in real-time could be very helpful for mathematical and artistic applications.

8 Appendix A: Electrical Specifications Standards Used with Expected Constraints

8.1 HDMI 1.3 Communication

- **Description:** We will have an HDMI bus interface from the Broadcom Compute Module to an output display screen at a maximum pixel clock rate of 680Mhz for a dual-link connection.
- **Standard:** EIA/CEA-861, TMDS High Speed Serial Data Communication

  Constraints: Our cameras can only stream at 60fps, which is far below the speed that HDMI is capable of projecting, so this protocol is sufficient.

8.2 CSI Communication

- **Description:** We will be using high-speed CSI lanes to connect our cameras to the primary processor, the Broadcom Compute Module. These are faster than any serial communication protocol due to the parallel nature of it.
- **Standard:** MIPI CSI-3-v1.1 Camera Serial Interface communication standard with I2C standard communication from BCM2837
- **Constraints:** At 720p, the CSI lanes allow streaming at 60FPS from a single camera stream. We need an frame rate of at least 20FPS for the application to run smoothly, which will be easily met.

8.3 RF Communication

- **Description:**Our hand held controller will wirelessly communicate with our main base unit via (2) XBEE Series 1, 2.4 GHz, 250 Kbit/sec, wire antenna RF modules. One module will be interfaced to the UARTs on the hand held controller microprocessor while the other will be interfaced to the UART on the main stereo board.
- **Standard:** IEEE 802.15.4 low rate wireless personal area network standard (RF transmission) and IEEE RS232 (digital microprocessor interface).
- **Constraints:** Our hand held controller is battery operated and is expected to consume 30 mA per hour. The XBEE Series 1 indoor distance range with free line of site is typically around 100 ft. However, with obstacles, like walls and other objected in the line of sight path, it can be greatly reduced. For our application, we require a 1 foot open line of sight path and therefore should be okay with this module.

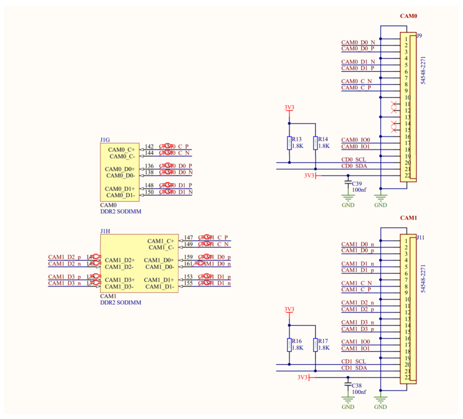
8.4 PCIe Serial Communication

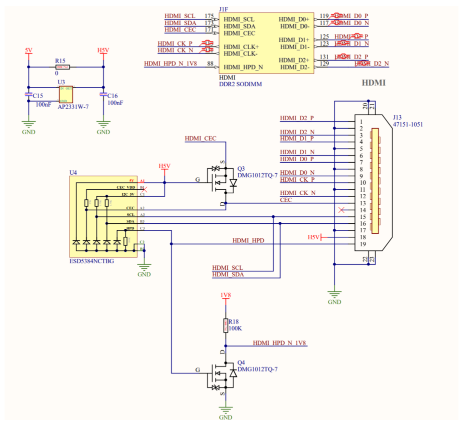
- **Description:** Our compute module will connect to the stereo vision board using a PCIe interface. This allows for communication to the stereo vision board at 1.969 GB/s for each individual lane. The connector we will be using is a DDR2 SODIMM, which can support data rates of 800 MT/s.

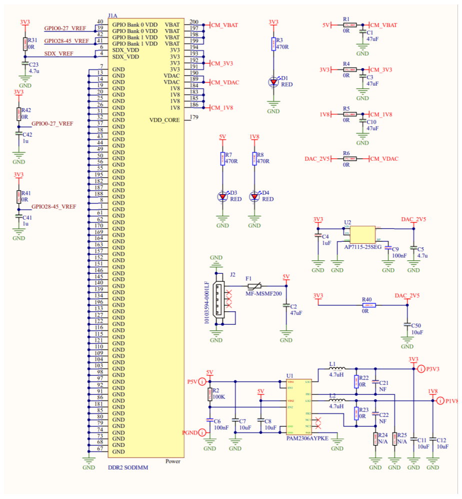
- **Standard:** DDR2-800C Double data rate synchronous dynamic random-access memory interface
- **Constraints:** The PCIe interface is incredibly fast and we do not need the RF information to reach the BCM2837 any faster than 5-6 MT/s, so the 800 MT/s should be totally sufficient.

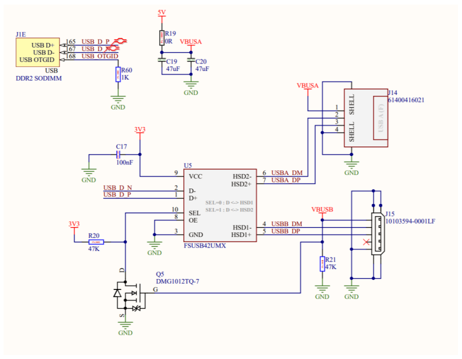
9 Appendix B: Schematics

9.1 Stereo Board


(a) Camera Schematic


(b) HDMI Schematic


(c) Power Schematic


(d) USB Schematic

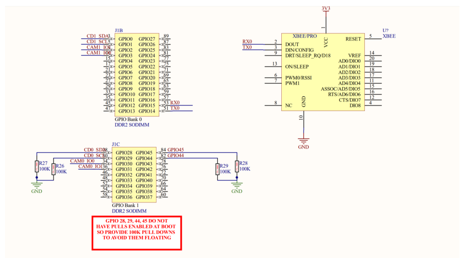
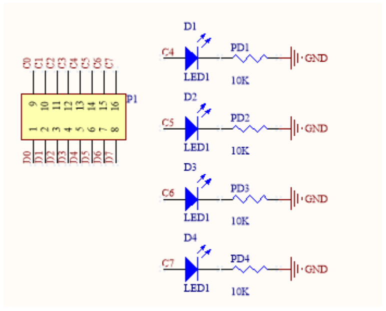
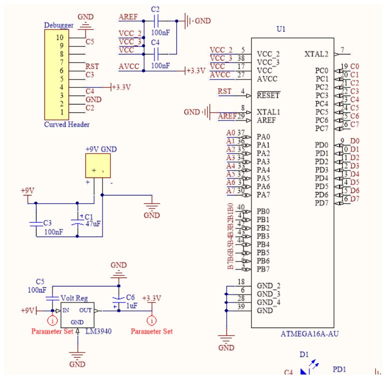

(e) Xbee and GPIO Schematic

Figure 12: Stereo Board Schematics

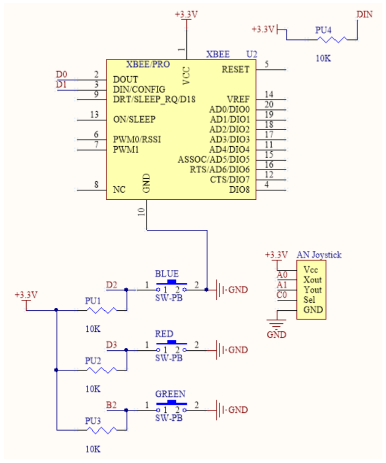
9.2 UX Control Board



(a) Header and LED Schematic



(b) Processor, Power, Debugger Schematic



(c) Buttons, Joy stick, Xbee

Figure 13: UX Control Board Schematics

10 Appendix C: Sample Code

Full code can be found on https://github.com/E-Gruda/AirCanvas

10.1 Computer Vision: Finger/Hand Tracking

```
"""
The following code is a library that we used to detect a blue glove
in open space. Thank you to
   https://dev.to/amarlearning/finger-detection-and-tracking-using-opencv-and-python-586m
and
https://picoledelimao.github.io/blog/2015/11/15/fingertip-detection-on-opencv/
for resources and explanations of how to accomplish this.
"""

import cv2
import numpy as np
import matplotlib.pyplot as plt
hand_hist = None
traverse_point = []
painting = []
total_rectangle = 9
```

```
hand_rect_one_x = None
hand_rect_one_y = None

hand_rect_two_x = None
hand_rect_two_y = None



def rescale_frame(frame, wpercent=130, hpercent=130):
    width = int(frame.shape[1] * wpercent / 100)
    height = int(frame.shape[0] * hpercent / 100)
    return cv2.resize(frame, (width, height), interpolation=cv2.INTER_AREA)


def contours(hist_mask_image):
    gray_hist_mask_image = cv2.cvtColor(hist_mask_image, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(gray_hist_mask_image, 0, 255, 0)
    _, cont, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
        cv2.CHAIN_APPROX_SIMPLE)
    return cont


def max_contour(contour_list):
    max_i = 0
    max_area = 0

    for i in range(len(contour_list)):
        cnt = contour_list[i]

        area_cnt = cv2.contourArea(cnt)

        if area_cnt > max_area:
            max_area = area_cnt
            max_i = i

        return contour_list[max_i]

def hist_masking(frame, hist):
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    dst = cv2.calcBackProject([hsv], [0, 1], hist, [0, 180, 0, 256], 1)

    disc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (31, 31))
    cv2.filter2D(dst, -1, disc, dst)
```

```python
    ret, thresh = cv2.threshold(dst, 150, 255, cv2.THRESH_BINARY)
    # thresh = cv2.dilate(thresh, None, iterations=5)
    thresh = cv2.merge((thresh, thresh, thresh, thresh))
    return cv2.bitwise_and(frame, thresh)


def centroid(max_contour):
    moment = cv2.moments(max_contour)
    if moment['m00'] != 0:
        cx = int(moment['m10'] / moment['m00'])
        cy = int(moment['m01'] / moment['m00'])
        return cx, cy
    else:
        return None


def farthest_point(defects, contour, centroid):
    if defects is not None and centroid is not None:
        s = defects[:, 0][:, 0]
        cx, cy = centroid

        x = np.array(contour[s][:, 0][:, 0], dtype=np.float)
        y = np.array(contour[s][:, 0][:, 1], dtype=np.float)

        xp = cv2.pow(cv2.subtract(x, cx), 2)
        yp = cv2.pow(cv2.subtract(y, cy), 2)
        dist = cv2.sqrt(cv2.add(xp, yp))

        dist_max_i = np.argmax(dist)

        if dist_max_i < len(s):
            farthest_defect = s[dist_max_i]
            farthest_point = contour[farthest_defect][0]


            # squeeze the farthest point in the direction of the centroid
            # define the percentage of the centroid to squeeze
            pct_sq = 0.05

            if (farthest_point[0] > cx+(cx*pct_sq)):
                farthest_point[0] = farthest_point[0] - cx*pct_sq
            elif (farthest_point[0] < cx-(cx*pct_sq)):
                farthest_point[0] = farthest_point[0]+cx*pct_sq
            if (farthest_point[1] > cy+(cy*pct_sq)):
                farthest_point[1] = farthest_point[1]-cy*pct_sq
            elif (farthest_point[1] < cy-(cy*pct_sq)):
                farthest_point[1] = farthest_point[1]+cy*pct_sq

            return tuple(farthest_point)
        else:
            return None

def manage_image_opr(frame, hand_hist):
    hist_mask_image = hist_masking(frame, hand_hist)
    contour_list = contours(hist_mask_image)
    max_cont = max_contour(contour_list)
    cnt_centroid = centroid(max_cont)
    cv2.circle(frame, cnt_centroid, 5, [255, 0, 255], -1)
    return cnt_centroid
```

10.2 Stereo Vision: Mapping 3D space

```
"""
The following code is a library that we created to render the depth map
necessary to render the 3d model. A huge thanks to www.stereopi.com and
the people who work there for the mentorship on this part.
"""
from picamera import PiCamera
import time
import cv2
import numpy as np
import threading
import json
from stereovision.calibration import StereoCalibrator
from stereovision.calibration import StereoCalibration
from datetime import datetime
from detect_finger import *
from mpl_toolkits.mplot3d import Axes3D
from dust import *
import gui_coords_lib
import uart
import loadOBJ
import os
global camera
# Depth map default preset
```

```
SWS = 5
PFS = 5
PFC = 29
MDS = -30
NOD = 160
TTH = 100
UR = 10
SR = 14
SPWS = 100

# Camera settimgs
cam_width = 1280
cam_height = 480
#cam_width = 2560
#cam_height = 720

# Final image capture settings
scale_ratio = 0.5

# Camera resolution height must be dividable by 16, and width by 32
cam_width = int((cam_width+31)/32)*32
cam_height = int((cam_height+15)/16)*16
print ("DM: Used camera resolution: "+str(cam_width)+" x "+str(cam_height))

# Buffer for captured image settings
img_width = int (cam_width * scale_ratio)
img_height = int (cam_height * scale_ratio)
capture = np.zeros((img_height, img_width, 4), dtype=np.uint8)
print ("DM: Scaled image resolution: "+str(img_width)+" x "+str(img_height))

# Initialize the camera
#camera = PiCamera(stereo_mode='side-by-side',stereo_decimate=False)
#camera.resolution=(cam_width, cam_height)
#camera.framerate = 20
#camera.hflip = True

in_folder = 'newcams_calib_result'

# Implementing calibration data
print('DM: Read calibration data and rectifying stereo pair...')
calibration = StereoCalibration(input_folder=in_folder)

disparity = np.zeros((img_width, img_height), np.uint8)
sbm = cv2.StereoBM_create(numDisparities=0, blockSize=21)
```

```
def stereo_depth_map(rectified_pair):
    dmLeft = rectified_pair[0]
    dmRight = rectified_pair[1]
    disparity = sbm.compute(dmLeft, dmRight)
    local_max = disparity.max()
    local_min = disparity.min()
    disparity_grayscale = (disparity-local_min)*(65535.0/(local_max-local_min))
    disparity_fixtype = cv2.convertScaleAbs(disparity_grayscale,
        alpha=(255.0/65535.0))
    disparity_color = cv2.applyColorMap(disparity_fixtype, cv2.COLORMAP_JET)
    key = cv2.waitKey(1) & 0xFF
    if key == ord("q"):
        quit();
    return disparity_color, disparity_fixtype,
        (disparity-local_min)/(local_max-local_min)

def load_map_settings(fName):
    global SWS, PFS, PFC, MDS, NOD, TTH, UR, SR, SPWS, loading_settings
    print('Loading parameters from file...')
    f=open(fName, 'r')
    data = json.load(f)
    SWS=data['SADWindowSize']
    PFS=data['preFilterSize']
    PFC=data['preFilterCap']
    MDS=data['minDisparity']
    NOD=data['numberOfDisparities']
    TTH=data['textureThreshold']
    UR=data['uniquenessRatio']
    SR=data['speckleRange']
    SPWS=data['speckleWindowSize']
    #sbm.setSADWindowSize(SWS)
    sbm.setPreFilterType(1)
    sbm.setPreFilterSize(PFS)
    sbm.setPreFilterCap(PFC)
    sbm.setMinDisparity(MDS)
    sbm.setNumDisparities(NOD)
    sbm.setTextureThreshold(TTH)
    sbm.setUniquenessRatio(UR)
    sbm.setSpeckleRange(SR)
    sbm.setSpeckleWindowSize(SPWS)
    f.close()
    print('Parameters loaded from file '+fName)
```

```
def get_grayscale_lr_images(frame, img_height, img_width):
    # get the left and right images in the context of the frame
    pair_img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    imgLeft_gr = pair_img [0:img_height,0:int(img_width/2)] #Y+H and X+W
    imgRight_gr = pair_img [0:img_height,int(img_width/2):img_width] #Y+H and X+W
    return imgLeft_gr, imgRight_gr

def get_grayscale_lr_crop(imgLeft, imgRight, boundaries):
    b=boundaries
    crop_l = imgLeft[b[0]:b[1], b[2]:b[3]]
    crop_r = imgRight[b[0]:b[1], b[2]:b[3]]
    return crop_l, crop_r

def get_local_dm(frame, img_height, img_width, calibration, touch_point, pad_val =
    100):
    # figure out the boundaries
    boundaries = np.zeros(4)
    boundaries[0] = touch_point[0] - pad_val
    boundaries[1] = touch_point[0] + pad_val
    boundaries[2] = touch_point[1] - pad_val
    boundaries[3] = touch_point[1] + pad_val
    boundaries[boundaries < 0] = 0
    for num in range(2):
        if boundaries[num] >= img_width:
            boundaries[num] = img_width-1
    for num in range(2,4):
        if boundaries[num] >= img_height:
            boundaries[num] = img_height-1
    left_im, right_im = get_grayscale_lr_images(frame, img_height, img_width)
    l_crop, r_crop = get_grayscale_lr_crop(left_im, right_im,
       boundaries.astype(int))

    # create a relative touch point to the new window
    rel_tp = np.zeros(2)
    rel_tp[0] = touch_point[0] - boundaries[0]
    rel_tp[1] = touch_point[1] - boundaries[2]

    rectified = rect_images(l_crop, r_crop, calibration)
    disparity, ft, gs = stereo_depth_map(rectified)
    return gs, rel_tp.astype(int)

def get_grayscale_lr_images(frame, img_height, img_width):
    # get the left and right images in the context of the frame
    pair_img = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    imgLeft_gr = pair_img [0:img_height,0:int(img_width/2)] #Y+H and X+W
    imgRight_gr = pair_img [0:img_height,int(img_width/2):img_width] #Y+H and X+W
    return imgLeft_gr, imgRight_gr

def get_rgb_lr_images(frame, img_height, img_width):
    imgLeft = frame[0:img_height,0:int(img_width/2)]
    imgRight = frame[0:img_height,int(img_width/2):img_width] #Y+H and X+W
    return imgLeft, imgRight

def rect_images(im_l, im_r, calibration):
    # rectify the left and right pairs (this can be in grayscale or RGB)
    return calibration.rectify((im_l, im_r))

def get_dm(frame, img_height, img_width, calibration):
    imgLeft_gr, imgRight_gr = get_grayscale_lr_images(frame, img_height, img_width)
    rectified = rect_images(imgLeft_gr, imgRight_gr, calibration)
    disparity, ft, gs = stereo_depth_map(rectified)
    return disparity, ft, gs

uart_val =[0]
loadOBJ.uart_val[:] = uart_val
```

10.3 3D Object/ Dust3D

```
"""
The following code is a section of the class used to communicate with
the Dust3D Software over TCP/IP socket protocols
"""
import socket
import binascii
import numpy as np
import time
import random
import uuid

class dust:
    global s
    def __init__(self, TCP_IP = '127.0.0.1'):
        #TCP_IP = '127.0.0.1'
        TCP_PORT = 53309
        self.BUFFER_SIZE = 4096
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.connect((TCP_IP, TCP_PORT))
        TCP_PORT = 53309
        BUFFER_SIZE = 4096
    def start(self):
        global s
        TCP_IP = '127.0.0.1'
        TCP_PORT = 53309
        BUFFER_SIZE = 4096
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((TCP_IP, TCP_PORT))
        TCP_PORT = 53309
        BUFFER_SIZE = 4096
        return s
    def add_n_node(self,new_node,x,y,z, r=.01):
        #global node_count
        return self.s.send(binascii.hexlify("addnodewithid ".encode('utf8') +
            str(new_node).encode('utf8') +" ".encode('utf8')+str(x).encode('utf8') +
            " ".encode('utf8') +str(y).encode('utf8') + " ".encode('utf8') +
            str(z).encode('utf8') + " ".encode('utf8') +str(r).encode('utf8'))+
            "\0".encode('utf8'))
    def exportAsObj(self, write = 0, fname = "model.obj"):
        self.s.send(binascii.hexlify("exportAsObj".encode('utf8'))+
            "\0".encode('utf8'))
        count = 0
        while(write and count <5):
            reply_ = self.reply()
            if(reply_ != -1):
                f = open(fname, "w+")
                f.write((reply_))
                f.close()
                print(reply_)
                return 1
            count = count +1
            self.s.send(binascii.hexlify("exportAsObj".encode('utf8'))+
                "\0".encode('utf8'))
           # return reply_
        #return self.reply()
    def reply(self):
        #global reply_
        response = bytes()
        while True:
            oneEnd = response.find(chr(0).encode('utf8'))
            if (-1 == oneEnd):
                response += self.s.recv(self.BUFFER_SIZE)
                continue
            reply_ = response[:oneEnd]
            response = response[oneEnd + 1:]
            output =binascii.unhexlify(reply_)
            print(output[:14])
            if(output[:12] == "#exportready".encode('utf8')):
                print("test EXPORT")
                self.exportAsObj(0)
            #print (((reply_))[:14])
            if(((reply_))[:14] == b'2b4f4b0d0a2320'):
                f = open("model.obj", "w+")
                f.write((binascii.unhexlify(reply_))[14:].decode('utf8'))
                f.close()
        else:
            return -1
```

10.4 main() of AirCanvas

```
def main():
    uart_val = Array('i',[0])
    loadOBJ.uart_val[:] = uart_val
    is_hand_hist_created = False
    x = 0
    final_scale = 1
    local = False
    #Spwan UART Process
    uart_lock = Lock()
    uart_t = Process(target=uart.cont_get_value, args=(uart_val,uart_lock))
    uart_t.daemon = True
    uart_t.start()
    # Load 3D object as thread in parent process
    obj_t = threading.Thread(target=loadOBJ.start)
    obj_t.daemon = True
    obj_t.start()
    #Load UI in parent process
    img = np.zeros((340, 1280, 3), np.uint8)
    red_txt = "Press red to clear your drawing"
    yel_txt = "Press yellow to continue drawing"
    gre_txt = "Press green to create a new point"
    js_txt = "Press joystick to finish"
    js2_txt = "Move joystick left and right to rotate your canvas"
    js3_txt = "Move joystick up and down to adjust line width"
```

```
img = cv2.putText(img, yel_txt, (20, 50), cv2.FONT_HERSHEY_SIMPLEX, 1,
    (0,255,255), 2, cv2.LINE_AA)
img = cv2.putText(img, red_txt, (20, 100), cv2.FONT_HERSHEY_SIMPLEX, 1,
    (0,0,255), 2, cv2.LINE_AA)
img = cv2.putText(img, gre_txt, (20, 150), cv2.FONT_HERSHEY_SIMPLEX, 1,
    (0,255,0), 2, cv2.LINE_AA)
img = cv2.putText(img, js2_txt, (20, 200), cv2.FONT_HERSHEY_SIMPLEX, 1,
    (255,255,0), 2, cv2.LINE_AA)
img = cv2.putText(img, js3_txt, (20, 250), cv2.FONT_HERSHEY_SIMPLEX, 1,
    (255,0,255), 2, cv2.LINE_AA)
img = cv2.putText(img, js_txt, (20, 300), cv2.FONT_HERSHEY_SIMPLEX, 1,
    (255,0,0), 2, cv2.LINE_AA)
global instr_screen
instr_screen = img
#create shared Queue
q = Queue()
#Producer Process
p = Process(target=producer, args=(uart_val,q))
p.start()
#Consumer Process
c = Process(target=consumer, args=(uart_val, q, uart_lock))
c.start()
time.sleep(10)
while((uart_val[0] & 0x08) != 0x08):
    loadOBJ.uart_val[:] = uart_val
    if(not obj_t.is_alive()):
        print("restarting")
        obj_t = threading.Thread(target=loadOBJ.start)
        obj_t.daemon = True
        obj_t.start()
        print("restart")

#join/kill all processes
p.join(timeout = 1.0)
c.join(timeout = 1.0)
uart_t.join(timeout = 1.0)
q.close()
sys.exit()
```

10.5 Control Board

```
/*
```

```
 * UART-XBEE Test.c
 *
 * Created: 2/13/2019 12:02:27 PM
 * Author : Caleb and Evan
 */
#define F_CPU 1000000UL
#define USART_BAUDRATE 9600UL
#define BAUD_PRESCALE (((F_CPU / (USART_BAUDRATE * 8UL))) - 1)
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <util/delay.h>
#include "Joystick.h"
#include "USART.h"
void init_pins();
void init_interrupts();
volatile uint8_t packet;
int main(void)
{
   //MAX CLK INIT
   //bits 7:6 = vertical joystick (2 up, 1 down, 0 still)
   //bits 5:4 = horizontal joystick (2 right, 1 left, 0 still)
   //bits 3:0 = select, g, r, b buttons respectively
   packet &= 0;
   Joystick j = {0, 0};
   init_pins();
   init_interrupts();
   Init_ADC();
   USART_Init();
   PORTC = 0;
   PORTC |= 0xF0;
   while(1) {
      get_joystick_ternary(&j);
      if(j.vert == 2) packet |= 0x80;
      else if (j.vert == 1) packet|= 0x40;
      if(j.horiz == 2) packet |= 0x20;
      else if (j.horiz == 1) packet|= 0x10;

      if ((PINC & 0x01) == 0x00)
         packet |= 0x08;

      USART_SendByte(packet); // send value
      packet &= 0x00;
      PORTD = 0;
```

```
  }
}
void init_interrupts() {
     GICR = 1<<INT0 | 1 <<INT1 | 1 <<INT2; //Enable INT0, INT1, INT2
     MCUCSR = 0<<ISC2; // falling edge interrupt activates yellow button (INT2)
     MCUCR = 1<<ISC01 | 0<<ISC00; // The falling edge of INT0 generates an
        interrupt request.
     PORTD = 0xFF; // pull up
     PORTB |= 0x04; // pull up portb
     sei();
}
void init_pins() {
  //D3:2 are buttons
  //D0 is the RX line (input)
  //D1 is the TX line (output)
  DDRD = 0xF2;
  //C7:4 are the LED's
  //C0 is the input select line from the joystick
  DDRC = 0xFE;
  // set the ADC's from the joystick to inputs
  DDRA = 0xFC;
  //B1 is a button
  DDRB = 0xFD;
}
ISR(INT0_vect)
{
  //int0 = pd2 = yellow
  packet |= 0x01;
}
ISR(INT1_vect)
{
  //int1 = pd3 = red
  packet |= 0x02;
}
ISR(INT2_vect)
{
  //int2 = pb2 = green
  packet |= 0x04;
}
```