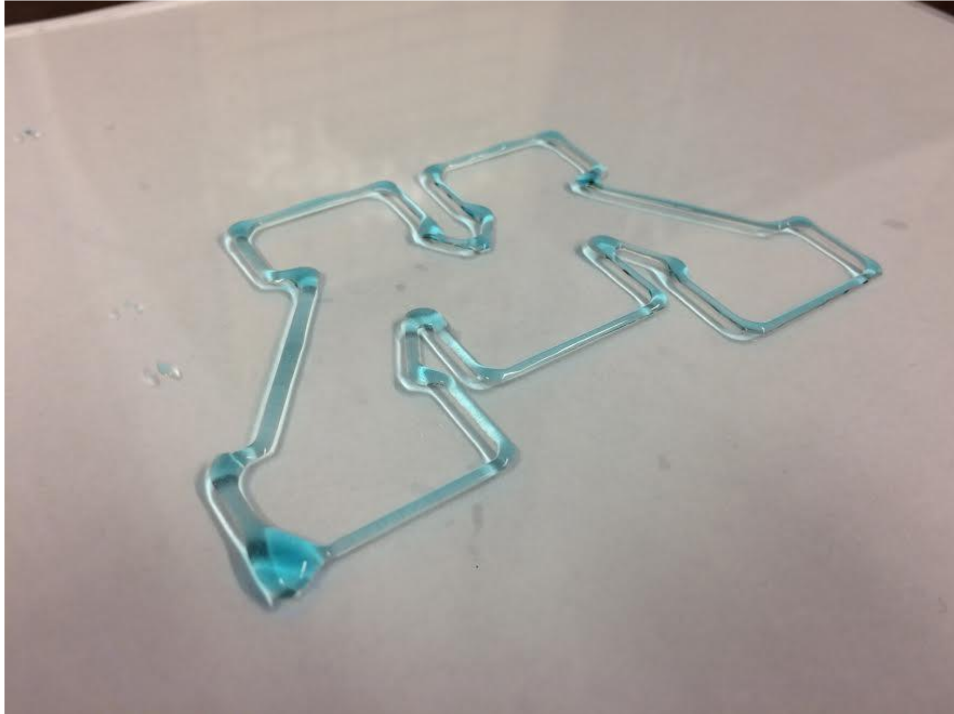
3D Printing onto a Moving Target



A MECHANICAL ENGINEERING SENIOR DESIGN PROJECT COMPLETED BY:

Reed Johnson, Galen Helgemo, Jun Liu, Joe Viavattine, Shiping Yi, Thomas Young

May 2, 2017

Executive Summary

Problem Definition

Conventional 3D printers print on a static baseplate or object in an open-loop control scheme, meaning there is no sensory feedback to the printer controller regarding the base plate's position or orientation. As a result, the baseplate must be stationary during printing to achieve acceptable print accuracy. By actively monitoring the position and orientation of the object being printed on, a controller can correct for movement of the target object, allowing for printing to occur on an object moving along an arbitrary, unknown planar path.

Design Description

Two endoscope cameras were placed next to extruder nozzle of the printer to provide a composite image of the target that was unobscured by the nozzle. This composite video feed was processed to find the center of an identifying object on the print target with respect to the extruder nozzle. A position-based visual servoing controller combined the X and Y position of the target, desired print geometry data, and the extruder's current position to generate the appropriate extruder velocity commands in the X and Y directions. These velocity command were passed to a microcontroller on the main print gantry, which was responsible for interpreting the velocity commands and running the low-level hardware components.

Evaluation

A physical prototype was constructed to validate the control strategy and computer vision tracking algorithms. The controller was validated by feeding in the current X and Y position of the target's center from the path generator gantry, a device that moved the target through a random path. The system was found to print on a moving surface with a speed of 2 cm/s, with an accuracy of 1 mm of the desired geometry. The computer vision program was validated by following an identifying mark on the target as it moved along a random path. While both components were validated separately and found to work, further development is required to combine both the computer vision and controller aspects of the project into an integrated computer vision and control program.

Table of Contents

Executive Summary ... **2**

1 Problem Definition ... **5**

1.1 Problem Scope ... 5

1.2 Technical Review ... 5

1.2.1 Applications of 3D Printed Circuits ... 5

1.2.2 Applications of 3D Printing on Moving Objects ... 6

1.2.3 Previous Work: Closed-loop Control for Static 3D Printers ... 6

1.2.4 Previous Work: Additive Manufacturing onto Human Anatomy ... 6

1.2.5 Previous Work: Visual Servoing ... 7

1.2.6 Previous Work: Common Computer Vision Sensors ... 8

1.3 Design Requirements ... 9

2 Design Description ... **11**

2.1 Summary of Design ... 11

2.2 Detailed Description ... 11

2.2.1 Print Geometry Array ... 12

2.2.2 Motion Controller ... 13

2.2.3 Plant: Primary Gantry, Structure, Hardware, Motors, and Extruder(s) ... 14

2.2.4 Moving Target System ... 16

2.2.5 Camera Configuration ... 18

2.2.6 Image Processing Algorithms ... 20

2.2.7 Serial Communication ... 20

3 Design Evaluation ... **21**

3.1 Test Plan and Results ... 21

3.2 Design Assessment ... 26

3.3 Future Work ... 27

Bibliography ... **29**

Appendices ... **30**

Appendix A: List of Terms ... 30

Appendix B: Data Packet Structure ... 31

Appendix C: Extruder Latency Characterization 31
Appendix D: Confirmation of Unobstructed Endoscope FoV with 1.5” Needle....... 33
Appendix E: Extruder Relay Board.. 34
Appendix F: Open CV Frame Rate... 35
Appendix G: Computer Vision Detection Methods Explored 37
Appendix H: Finalized Source Code for Circle Detection 44
Appendix I: Comparison of Extruder Head Manipulator Systems 49
Appendix J: Sensing Hardware Selection 50
Appendix K: Field of View Fusion of Two Cameras.. 51
Appendix L: Main Gantry Print Algorithm 52

1 Problem Definition

1.1 Problem Scope

In recent years, 3D printing and other additive manufacturing techniques have become commonplace in engineering and manufacturing fields where prototype or customized parts must be developed quickly and at low cost. While printing with plastics is a well-established technology, current research has now turned to printing with more advanced materials, such as soft materials, biologics, and utilizing more sophisticated control techniques [1]. Most 3D printers operate in an open-loop control scheme and as a result they cannot correct for position errors in the printer gantry as well as movement of the target object to be printed on. The ability to print on a moving 3D target opens the door to printing tissues and medical devices directly onto a body during surgery or printing circuits and other structures on a moving assembly line.

1.2 Technical Review

The design problem of 3D printing on a moving target draws on many disparate areas of engineering such as control theory, computer vision, mechatronics, and material science. The following section offers a primer on several of these topics.

1.2.1 Applications of 3D Printed Circuits

3D printing and other additive manufacturing processes have been used to create electronic circuits and antennas embedded directly in printed material [2]. This one-step process has the potential to reduce the number of operations required to create a product with embedded electronics and allow for denser circuits to be integrated within a given space. If perfected, low cost embedded sensors and circuits could be used to monitor the performance of various components within a system, such as directly printing strain gauges onto a machine component or RFID tags into parts for inventory tracking.

Additive manufacturing of circuits also has several advantages over current circuit fabrication methods, which utilize a photoetching process and potentially harmful chemicals. By using additive processes, circuits can be produced directly on substances that could be damaged by these chemicals [3].

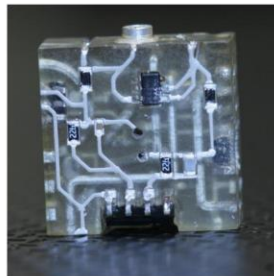


Figure 1: Signal conditioning circuit implemented in cube with additive manufacturing processes [3]

1.2.2 Applications of 3D Printing on Moving Objects

Recent advances in tissue engineering have yielded many man-made, biological structures including artificial hearts, bladders, and kidneys [1]. Many of these structures were produced in part by 3D printing a scaffold for cells to grow on. Currently, these structures are printed separately, grown, and implanted into a patient. If perfected, a 3D printer with visual feedback could directly print tissue scaffolds, such as bone grafts, directly onto a fractured bone while correcting for slight patient movement and allowing precision placement of the necessary tissues. As both the tissue engineering and bioprinting fields mature, advances could be made in emergency medicine and allow for more precise deposition of man-made tissue structures.

1.2.3 Previous Work: Closed-loop Control for Static 3D Printers

Closed-loop control refers to a system's ability to sense and correct for disturbances in the system that is being controlled. A common example of a closed-loop controller implementation is the cruise control system in a car which keeps the car at the same speed even as the car travels up and down hills. The analogous open-loop system to a cruise controller would be a device that keeps the throttle at a certain point, without respect to the car's current speed. Most commercial 3D printers operate in an open-loop configuration; the controller sends position commands to the motors, but does not receive any information back from the system as to the actual position of the print head. Because no feedback is provided to the controller, the system cannot correct for missed steps in the stepper motor, nor adjust for inaccuracies in print position. Recent work [4] has investigated the potential benefits of utilizing a closed-loop P-I controller to correct for errors in print position. With a rudimentary P-I controller, print accuracy was shown to increase by over 75%. While the control strategy was used on a static print, similar strategies will likely be implemented to continually correct for the changing position of the moving object to be printed on.

1.2.4 Previous Work: Additive Manufacturing onto Human Anatomy

Previous research has been conducted on additive manufacturing onto moving human anatomy. It has been shown that a two-linked robotic arm, as seen in Figure 2, can be kinematically optimized to track a human hand and potentially apply a substance in a controlled manner [5]. The robot arm in this case only allowed for 2D hand tracking. In another study, a six degree of freedom, commercial, robotic arm was used in conjunction with a Leap Motion sensor to track a hand in 3D and draw a straight line onto the hand [6]. In both cases, visual servoing was used to allow closed loop control to track the hand..

Figure 2: Kinematically optimized robot arm for additive manufacturing onto human anatomy [5]

1.2.5 Previous Work: Visual Servoing

In the mid-90's, Peter Corke developed a vision-based control technique of robots called visual servoing. The objective of position-based visual servoing is to control the position and orientation of a robot's end-effector relative to some target by estimating the global position of the target using details extracted from images collected from a camera most commonly placed on the robot's end effector [7]. The important coordinate frames involved in visual servoing are from the robot base frame to the end-effector, the end-effector to the camera, the camera to the target, and the robot base frame to the target. Figure 3 illustrates the transformations between these frames.

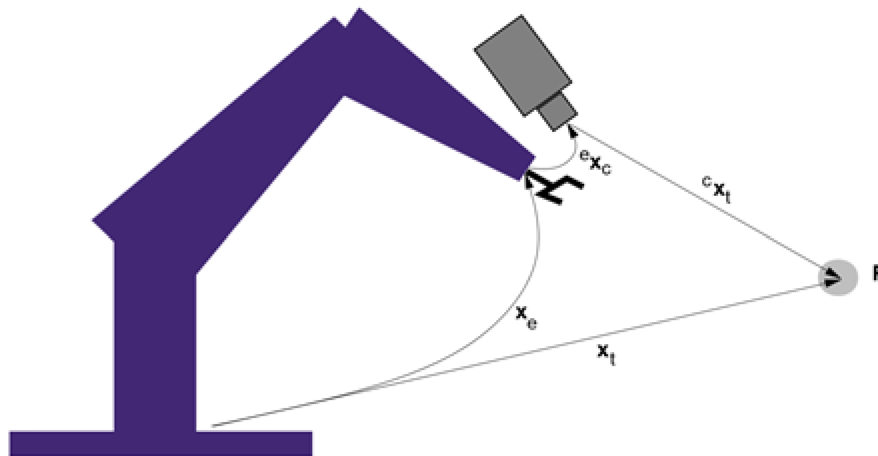


Figure 3: Relevant Reference Frames in Visual Servoing [7]

Knowing the position of the end-effector and the current orientation of the camera relative to the end-effector, the position of the target in the robot's base frame can be estimated using features extracted from the camera. This then enables the end-effector to be controlled along a desired trajectory relative to the target. Figure 4, a simple control system block diagram

illustrates this control technique with the required control law (commonly PD) and low level robot hardware and controllers.

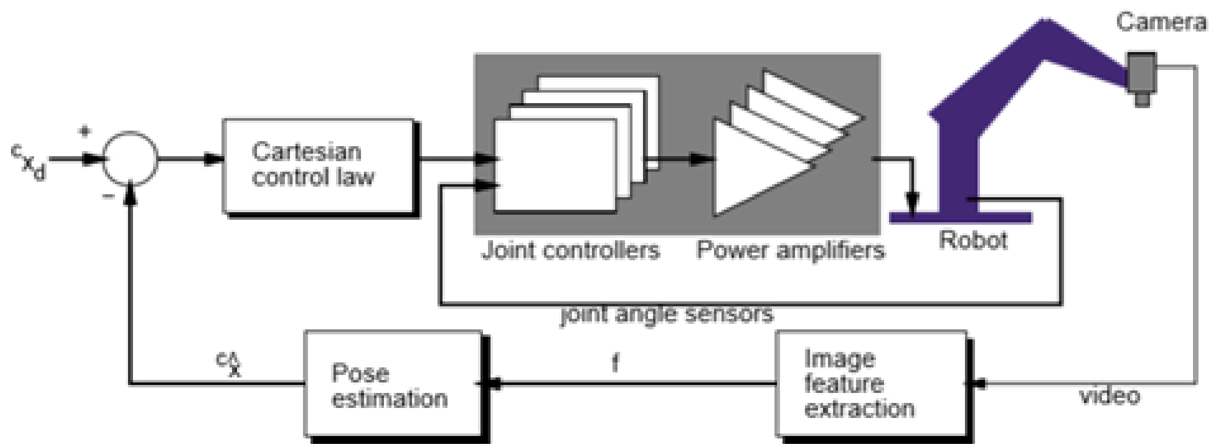


Figure 4: Position-based visual servoing control system block diagram [7]

1.2.6 Previous Work: Common Computer Vision Sensors

A wide range of image sensing technologies such as stereo vision, and time of flight sensors have been used to track motion of a moving object. Stereo vision provides information in the third dimension with the sensing of depth by using binocular vision like the human eyes. Multiple images are acquired simultaneously and then combined continuously [9]. Time of flight sensors detect object depth by projecting lines of light and calculating the distance from the source to the sensed object. Compared with stereo vision, images obtained using the time of flight method are not affected by shading and other illumination problems but lack the resolution provided by Stereo Vision. By closing the loop and offering feedback through such sensors, tasks have the potential to become autonomous by rejecting disturbances from the environment such as non-uniform motion. For real time operation, it is critical to have fast communication between sensors, local stations, and a central processor which is quickly becoming less of a barrier as time passes and computational power becomes more powerful all while becoming cheaper. With the combination of sensor options plus the decreased cost and increased performance of computer processing, computer vision is becoming a more viable option for many different markets to pursue.

1.3 Design Requirements

The following list of needs were identified through interviews with the customer. Needs were assigned an importance of 1-5, with 5 being most important to the design.

Table 1: Summary of design needs identified through customer interviews

#	Need	Importance
1	2 Degrees of Freedom of Motion Printer	5
2	Ability to print functional materials (electronics, biologics)	4
3	Capability of directly printing materials onto moving object	5
4	Visual tracking of target object	5
5	Design and Construct Printer Target Object	5
6	Design and Construct Target Movement System	4
7	Software can accept (sliced) GCODE file from an open source slicing program such as CURA	1
8	Printer is self-calibrating	2

Table 2: Quantitative requirements from customer Needs. The 'Need #s' refer back to a specific row on Table 1

Need #	Metric	Importance	Units	Marginal Value	Ideal Value
1	Degrees of Freedom of Motion of Motion	5	integer	2	3
1	Printer resolution	2	mm	.5	.1
1	Maximum Gantry Speed	2	mm/s	10	50
2	Print Material Viscosity	4	Pa-s	70	70
2	Extruder Latency	4	S	1	.1
2	Minimum Extrudable Material Volume	3	cm^3	3	30
2	Number of extrusion heads	5	integer	1	2

4	Computer Vision Tracking Error	4	mm	3	.5
5	Build Area	2	cm^2	400	1600
6	Maximum Target Speed	5	mm/s	25	25
4	Image Resolution	3	pixels	640x440	2048x1080
4	Image Processing Frame Rate	5	fps	15	30
7	Program accepts GCODE file	1	Binary	Yes	Yes
8	Printer Self Calibrating	3	Binary	Yes	Yes

2 Design Description

2.1 Summary of Design

The proposed design employs a silicone extruder nozzle affixed to a two-stage gantry system. A sensing system employing two endoscope cameras mounted close to the extruder head was chosen for tracking the moving object and facilitating closed-loop feedback control. The extruder driver unit mounts to the superstructure, along with the endoscope camera cables and the pressure line. The feed from the cameras are processed by a computer, which also runs the printer motion control algorithm. This algorithm outputs position commands to the gantry microcontroller, which runs the low-level motor control processes to move the extruder head to the correct location. Lastly, a path generator gantry moves the target in a repeatable fashion for characterizing the performance of the system. It is important to note that the target's path through the workspace is not known to the motion controller.

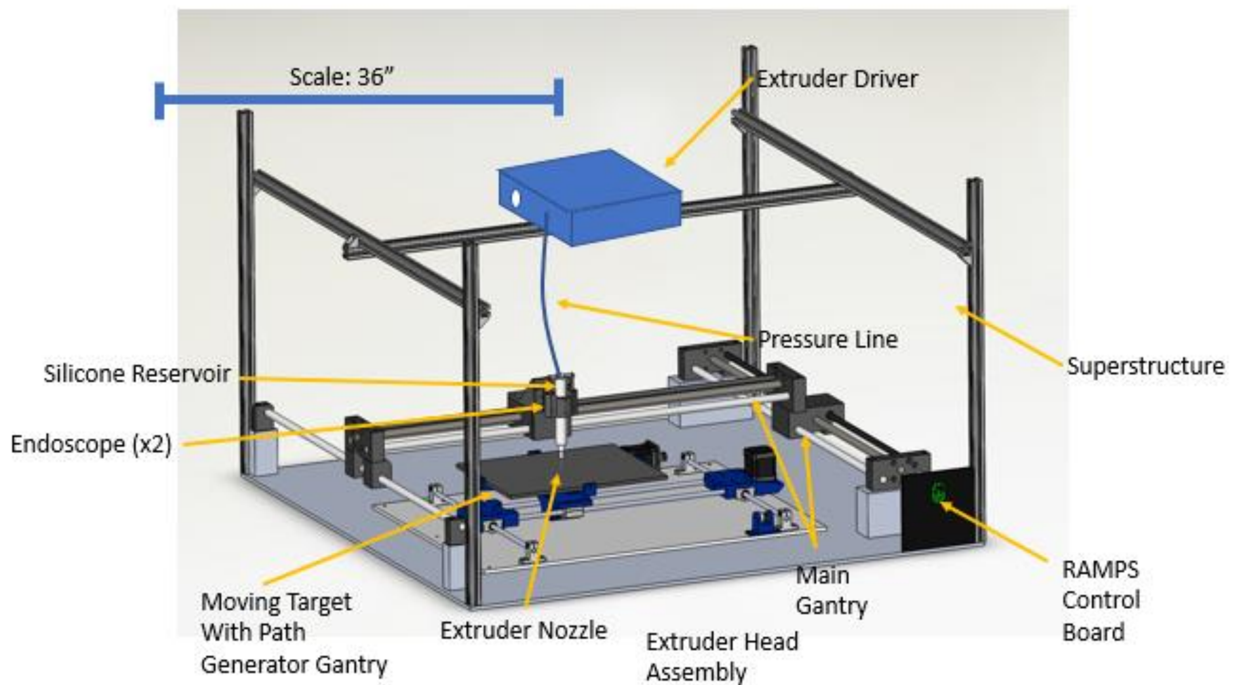


Figure 5: Moving target assembly. Not shown is the main processing computer, RAMPS board, and wiring. The second path generator gantry was constructed as well to provide a repeatable, consistent motion for testing different control system strategies.

2.2 Detailed Description

The various subsystems of the design can be organized into a system block diagram, as seen in Figure 6. On the left of the image is the print geometry array, also known as the reference. This is the input to the system that is user-defined, while the motion of the target can best be modeled as a disturbance to the system. Moving to the right is the motion controller, which uses the X-Y

points from the reference as well as processed data from the image processing algorithms to calculate the next expected position of the target and move the extruder head to the appropriate location to print. The desired extruder velocity commands from the controller are sent to the plant, which consists of all the mechanical components making up the main gantry system, as well as the low-level microcontroller used to drive the stepper motors. The output of the system is the extruder head's position with respect to the moving target's position. As the name implies, the target will not remain stationary. To correct for the target's continuously changing position, the target is watched by cameras mounted near the extruder head. The feeds from these cameras are combined into one composite feed, which is then processed with a computer vision algorithm to identify the new position and orientation of the target object. In this way, feedback is realized back to the controller.

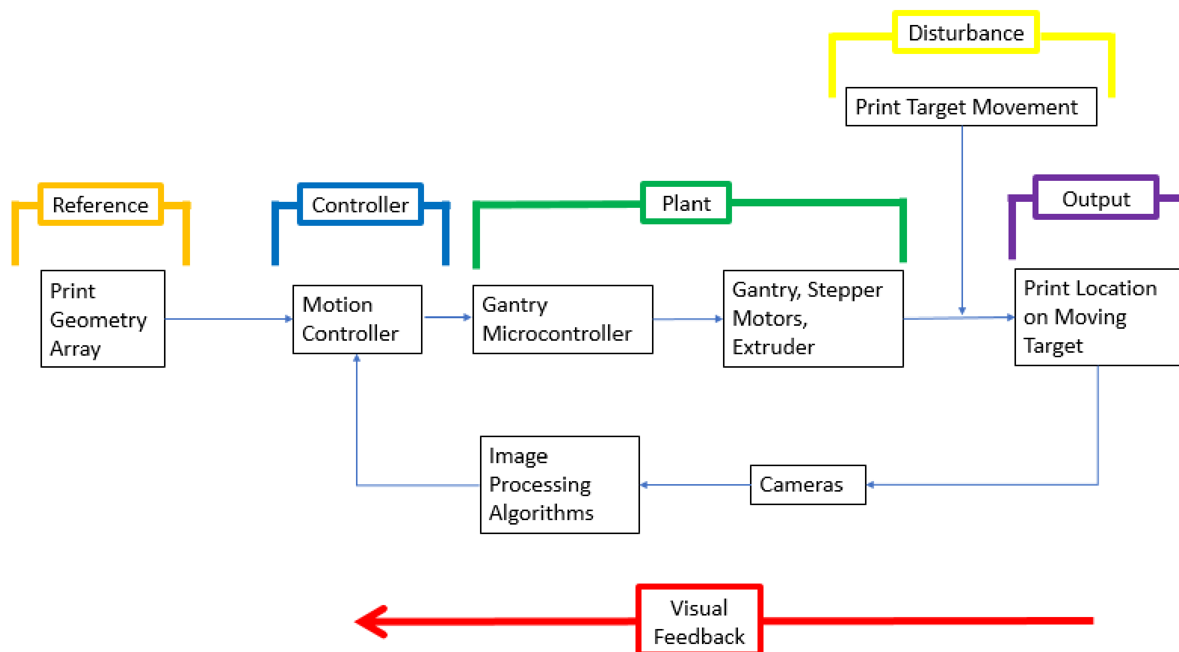


Figure 6: System Functional Block Diagram

2.2.1 Print Geometry Array

The print geometry array contains the X-Y data points that define the layer to be printed. For demonstration purposes, geometries such as squares, circles, and triangles will be printed. The X-Y points defining the vertices that make up the geometry are input as a 2D array in the controller Visual Studio Project.

2.2.2 Motion Controller

The motion controller is responsible for computing the required velocity of the extruder head from print geometry and target motion captured by the cameras. Equation 1 shows the X and Y-axis velocity commands that are output to the microcontroller at the camera's frame rate (30 Hz).

$$\bar{V}_E = \underbrace{\omega|\bar{r}_{PL}|\begin{bmatrix}-\sin(\alpha-\theta)\\ \cos(\alpha-\theta)\end{bmatrix} + \bar{V}_{T,G}}_{\text{Target Motion Compensation or Feedforward Control}} + \underbrace{S_P\begin{bmatrix}cos(\theta) & \sin(\theta)\\ -\sin(\theta) & \cos(\theta)\end{bmatrix}\frac{\bar{r}_{PD}}{|\bar{r}_{PD}|}}_{\text{Print Velocity}} \underbrace{- K_V\dot{\bar{r}}_{Error} - K_P\bar{\mathrm{r}}_{\mathrm{Error}}}_{\text{PD Feedback Control}} \quad (1)$$

where ω is the target angular velocity, $\bar{r}_{PL}$ is the desired print location relative to the target, θ is the angle of rotation of the target, α is the angle to the desired print location in target frame, $\bar{V}_{T,G}$ = target velocity with respect to the main gantry, S_P is the linear print speed, $\bar{r}_{PD}$ is the current print segment direction, $\bar{\mathrm{r}}_{\mathrm{Error}}$ is the current print displacement error, and K_V and K_P are positive definite matrices. In Equation 1, the first two terms compensate for random target rotational and linear motion, respectively, the third term is the instantaneous print trajectory velocity, and the final term corrects for the error in the actual and desired extruder positions, closing the loop. θ is found by taking the dot product between observed and nominal locations of the identifying features

$$\theta = \cos^{-1}(\bar{r}_{IF,T} \bullet \bar{r}_{IF,C}) = \cos^{-1}((\bar{r}_{IF2,T} - \bar{r}_{IF1,T}) \bullet (\bar{r}_{IF2,C} - \bar{r}_{IF1,C})) \quad (2)$$

where $\bar{r}_{IF,T}$ is the displacement vector between the centers of two unique identifying features in the target frame and $\bar{r}_{IF,C}$ is the displacement between these two identifying features in the camera's frame. The identifying feature vector is found using the endoscopes with Equation 3

$$\bar{r}_{IF1,C} = \begin{bmatrix}(Py_{IF1} - 240)FOV_X/480\\ (Px_{IF1} - 320)FOV_Y/640\end{bmatrix} \quad (3)$$

where (Px_{IF1}, Py_{IF1}) are the pixel coordinates of the centroid of identifying feature 1 and FOV_X and FOV_Y are the camera x and y field of view lengths, respectively. The equations for angular velocity, desired print location, and print direction are listed below.

$$\omega = \frac{\theta(t) - \theta(t - dt)}{dt} \quad (4)$$

$$\bar{r}_{PL} = \bar{r}_{n-1} + S_P\frac{\bar{r}_{PD}}{|\bar{r}_{PD}|}\left(t - t(\bar{r}_{n-1})\right) \quad (5)$$

$$\bar{r}_{PD} = \bar{r}_n - \bar{r}_{n-1} = \begin{bmatrix}x_n - x_{n-1}\\ y_n - y_{n-1}\end{bmatrix} \quad (6)$$

In Equation 6, (x_n, y_n) and (x_{n-1}, y_{n-1}) are two consecutive print coordinates.

The velocity of the target with respect to the global reference frame is calculated in equation 7. The position of the target with respect to the global reference frame must account for the motion of the camera and is calculated using Equation 8,

$$\bar{V}_{T,G} = \frac{\bar{r}_{T,G}(t) - \bar{r}_{T,G}(t-dt)}{dt} \tag{7}$$

$$\bar{r}_{T,G} = \bar{r}_{T,C} + \bar{r}_{C,E} + \bar{r}_{E,G} \tag{8}$$

where $\bar{r}_{T,G}$ is the position of the target with respect to the print gantry, $\bar{r}_{T,C}$ is the target position relative to a camera, $\bar{r}_{C,E}$ is the camera position relative to the extruder which is constant, and $\bar{r}_{E,G}$ is the extruder position relative to the print gantry determined from gantry encoder values.

$$\bar{r}_{T,C} = \bar{r}_{IF1,C} + \begin{bmatrix} cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \bar{r}_{T,IF1} \tag{9}$$

$$\bar{r}_{Error} = \bar{r}_{E,G} - \left(\bar{r}_{PL} + \bar{r}_{T,G}\right) = -\left(\bar{r}_{PL} + \bar{r}_{T,C} + \bar{r}_{C,E}\right) = \bar{r}_{E,T} - \bar{r}_{PL} \tag{10}$$

2.2.3 Plant: Primary Gantry, Structure, Hardware, Motors, and Extruder(s)

The main gantry structure is composed of the 3D printer machine components, base plate, extruder hardware, and camera mounting system. A Newmark Systems ETL X-Y gantry system was used to track the moving object. Two endoscope cameras and an extruder head were mounted to the carriage of the gantry system. An existing gantry system was chosen to streamline the development time by not having to design and construct an all-new precision gantry system. The Newmark system also features integrated stepper motors and encoders, reducing complexity, cost and procurement time.

The base plate consists of a .25” thick aluminum plate, which is used to mount the main gantry system, path generator gantry system, and superstructure. The superstructure is used to hold the extruder pressure regulator, pressure lines, and endoscope camera cables. The superstructure is made from an extruded aluminum T-slot channel, which allows for quick and easy assembly, and does not require any specialized tools beyond an Allen wrench.

The main gantry structure also contains a Teensy 3.2 microcontroller and Allegro A4988 stepper drivers to control the stepper motors. The Teensy accepts commands from the motion controller over serial bus from the processing computer. The data sent over serial is in the form of standard packets, which contain velocity and extruder command data. For a complete breakdown of the command packet structure, see Appendix B. The Teensy parses these packets and outputs commands to the A4988 stepper drivers and extruder relay, which control the stepper motors and extruder driver. Figure 8 shows a flow diagram of the information as it passes from the processing computer, through the Teensy, to the stepper drivers and extruder driver.

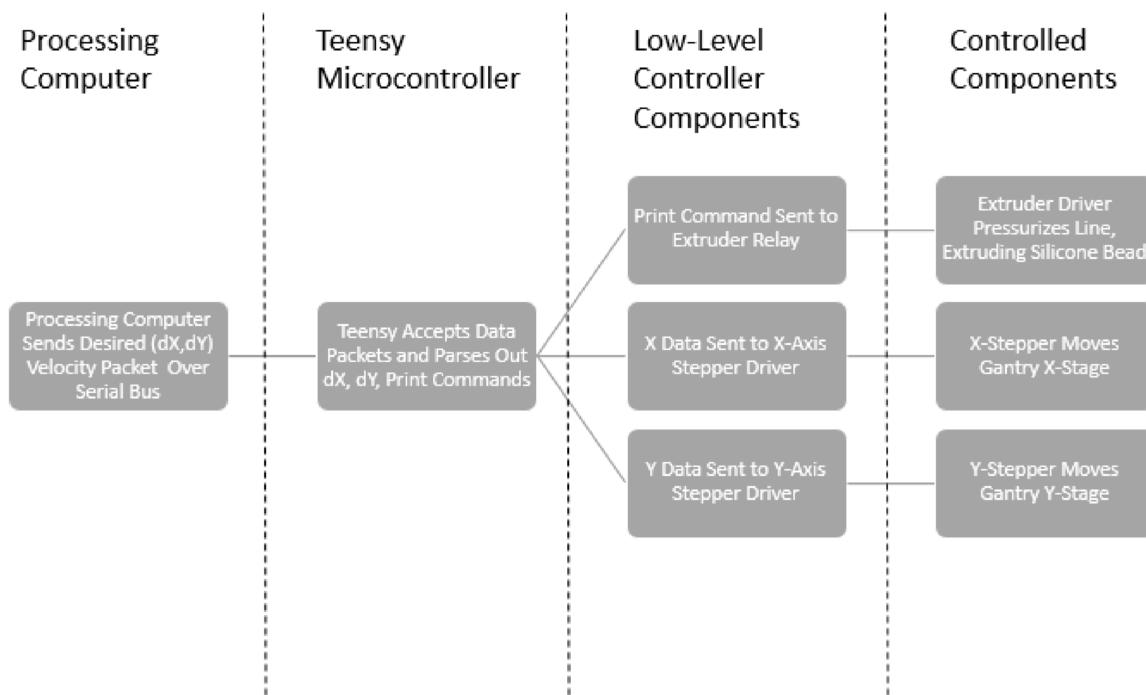


Figure 7: Control signal flow from processing computer to output components

The extruder head consists of the X-axis carriage, silicone syringe reservoir and nozzle, two endoscopes, and a mounting bracket. As seen in Figure 1, the silicone reservoir is connected to the pressure line, which is connected to the extruder driver that is attached to the superstructure. While the pressure line could have been run through the cable management trays, the overhead solution was chosen to reduce the chance of pinching the line and reduce the extruder latency by minimizing the volume of line that needs to be pressurized and depressurized. Figure 8 shows a close-up of the single and twin-material extruder designs. Initially a single material will be printed, but designing for a second extruder early in the design process allows for added flexibility without requiring major changes to the overall mechanical design.

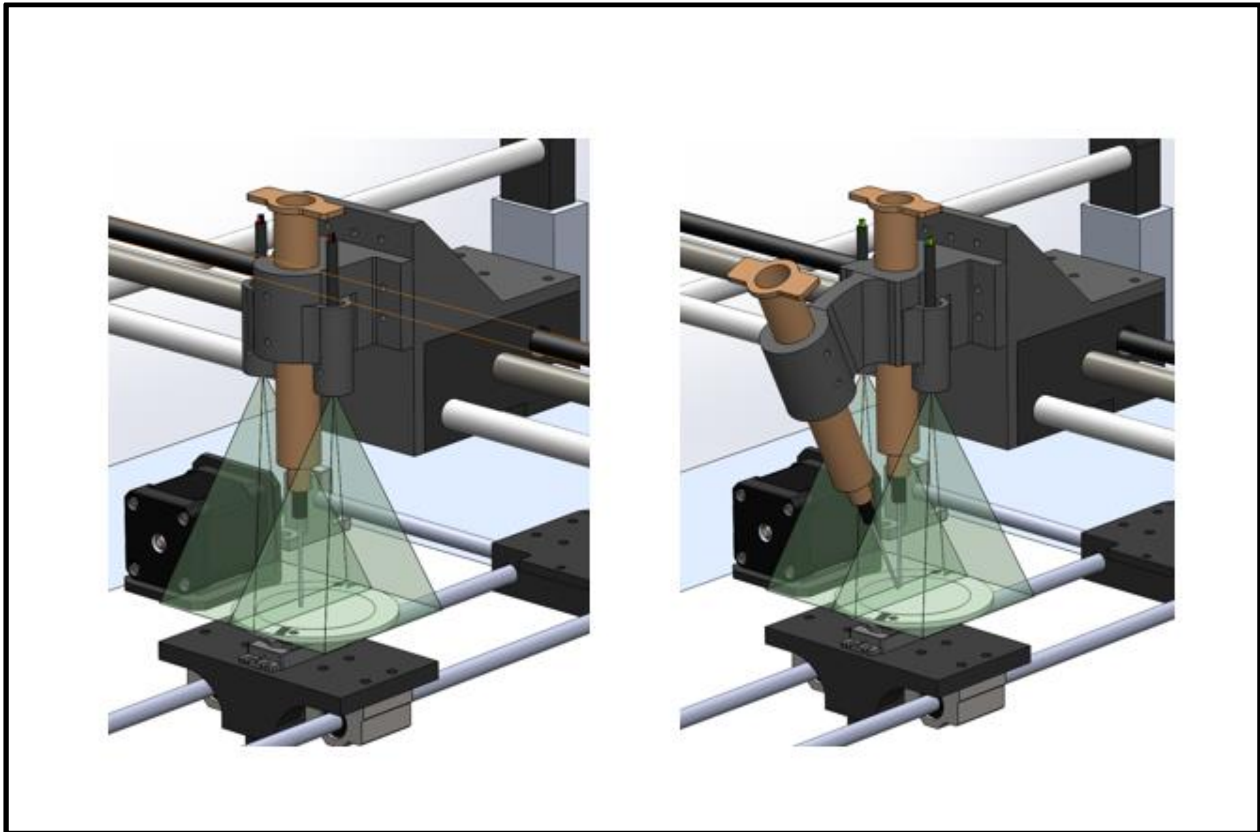


Figure 8: Design for one and two-material extrusion heads, respectively

While the print command is processed quickly, there is a lag time between when the command is sent and when silicone starts flowing from the nozzle. This can be attributed to the need to pressurize the entire pressure line and the resistance to flow of the viscous silicone compound. Experimental data found in Appendix C has shown this lag to be approximately 38 milliseconds at 80 psi. While redesigning the extruder system is outside the scope of this project, the controller algorithm compensates for this delay when beginning to print.

2.2.4 Moving Target System

The moving target subsystem consists of the path generator gantry and the target object that is to be printed onto. As seen in Figure 9, the target object is a circle with a 2 unique sets of 2 identifying shapes around the circumference to enable localization of the target. The target was designed to have a 5 cm diameter print workspace and an outer 1 cm thick annulus to place the sets of identifying features which each fit in 1 cm x 1cm square areas. The identifying features are different shapes which can be differentiated using a shape recognition and centroid algorithm detailed later in the report.

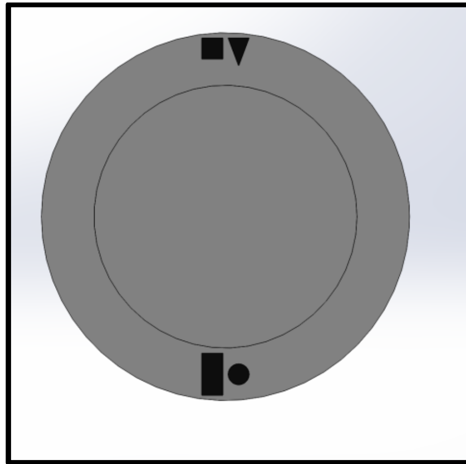


Figure 9: Moving target with two pairs of identifying features

The target is moved through the workspace by the path generator gantry system. The path generator gantry allows for development and validation of the control system against a controlled, repeatable motion. As seen in Figure 10, the path generator gantry subsystem consists of two linear stages driven by timing belts and stepper motors, which move the target carriage in the X and Y directions (global coordinate system). Inside the target carriage is the rotational servo, as seen in figure 11, which changes the target's orientation angle, φ. It is important to note that the path generated by the path generator gantry is not known by the motion controller. Rather, the motion of the moving target is only relayed back to the motion controller through feedback provided by the cameras. The path generator gantry is controlled by an Arduino Mega microcontroller. A RAMPS 1.4 circuit board is used as a shield on the Arduino for interfacing to the stepper drivers.

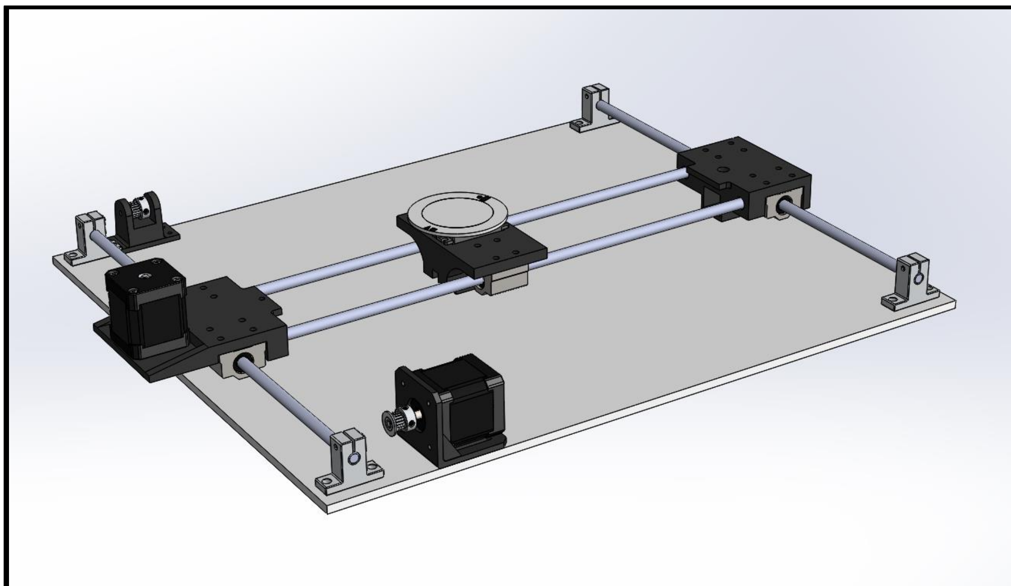


Figure 10: Moving Path generator gantry

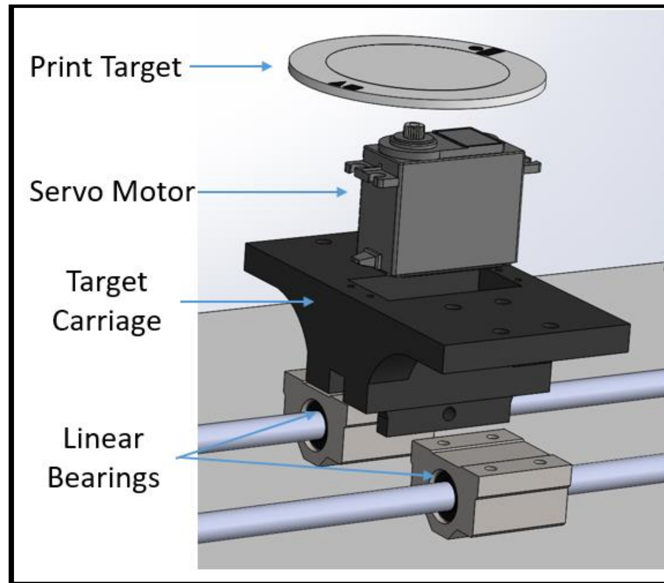


Figure 11: Exploded View of Moving Gantry Carriage

2.2.5 Camera Configuration

A DBPOWER endoscope with a 640x480 pixel array and marketed frame rate of 30 frames per second was chosen to meet the image resolution and sampling rate requirements. Additional detail regarding sensor selection can be found in Appendix 10. Two of these are used on opposite sides of the extruder to achieve the desired 7 cm x 7 cm field of view required about the center of the extruder to track the target. The required displacement from the target plane to achieve this field of view knowing the camera's view angle was determined to be 10 cm, a distance within the camera's focal range. The spacing between the two cameras was then designed to be 4.0 cm to prevent obstruction of the image field of view by the syringe body. Throughout this design process, the length of needle was determined to be 1.5 inches long.

The spacing of endoscopes also sets the range of overlap in their fields of view. It was desired to have an area of overlap of at least 1.4 cm, so that one pair of the target identifying features which fit in 1 cm by 1 cm areas are fully visible in at least one camera's image. Important (and some redundant) dimensions of the camera field of view are shown in Figure 12. Two possible positions of the target in the image field of view while printing are shown in Figure 13, illustrating the need of at least two unique sets of identifying features located on opposite sides of the target to continuously track the target while printing within the 5 cm diameter print workspace.

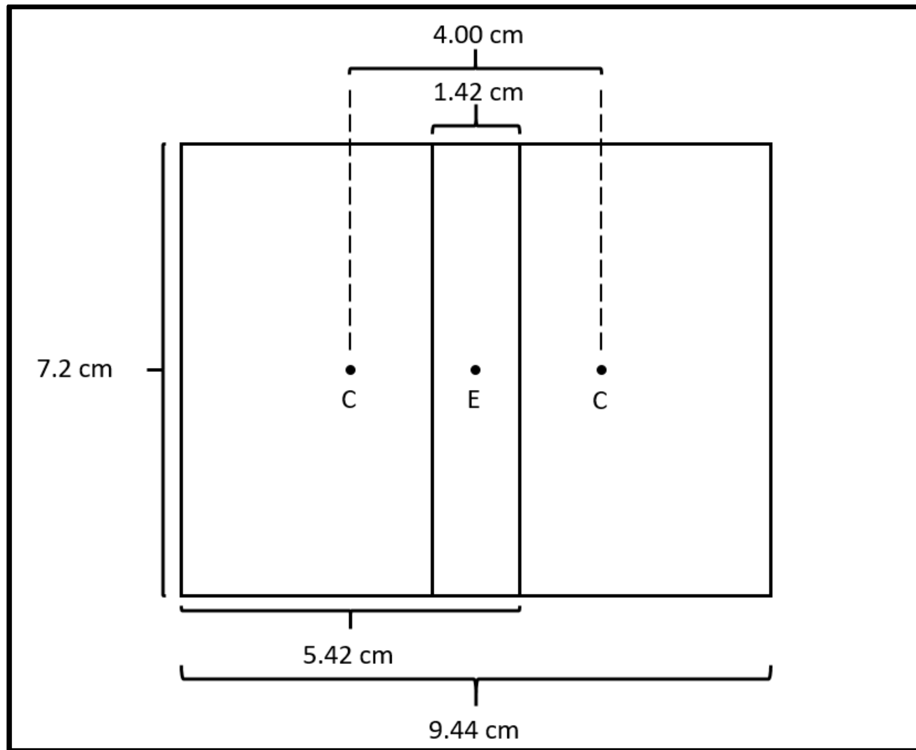


Figure 12: Endoscope field of view with endoscope (C) and extruder (E) positions

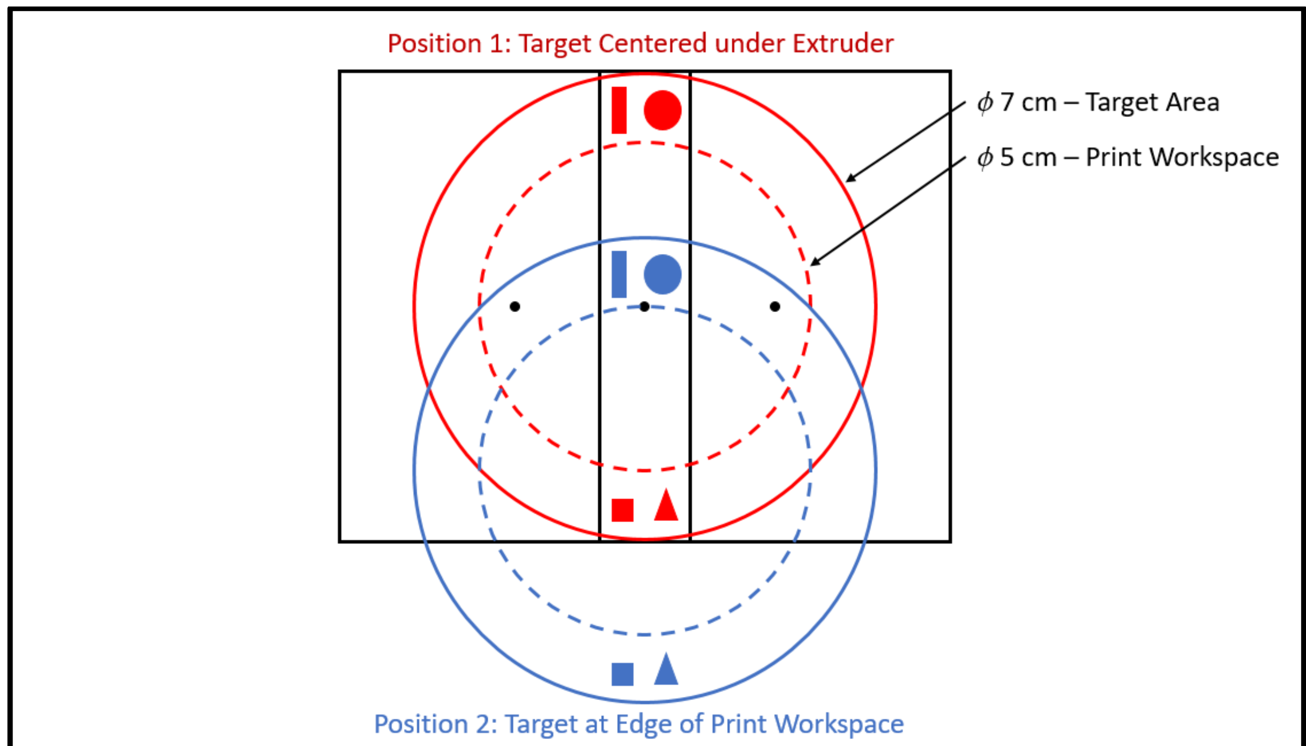


Figure 13: Target area, target print workspace, and unique target identifying feature pairs at two different positions relative to the extruder

2.2.6 Image Processing Algorithms

Various thresholding and tracking methods such as contour tracking, frame differencing, and shape approximating were explored to extract the desired contour or edge of the image. More on these methods can be found in Appendix G of this paper. Edge detection with Canny edge detection and a Hough Circle Transform algorithm for tracking objects was used for the final design. The Hough Circle Transform was used primarily due to its robustness in tracking circles even when part of it is out of the video frame or occluded. Also, the radius of the circle could be adjusted for tracking the desired size only. Figure 14 right shows the transformation from the original image to the grayscale thresholded image for edge detection. Pixel gradient was calculated and obtained to find the edge placements inside the image. This information is further used to find circles of desired range of radius from the Hough Circle function. More on this is found in Appendix G.

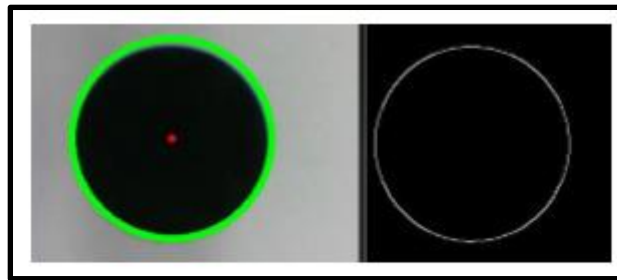


Figure 14: Hough Circle Transform with green indicating radius and red indicating center of the circle on featured object (left) and thresholded edge detection (right)

2.2.7 Serial Communication

After the velocity commands required to control the gantry are calculated, they need to be concatenated and sent to the Teensy microcontroller. The controller computer employed a serial communication protocol at 115200 baud (115200 bits per second) to communicate and facilitate control of the system over a USB cable between the computer and the Teensy microcontroller. This serial communication is established using the Boost C++ library which already has a serial communication class called Boost.asio and some error handling built into it. Per Appendix B, a packet structure is employed. Serial communication may have errors that occur while communicating. By using format characters to delineate one packet from another, most errors can be avoided. If there were no characters provided to dictate the start and stop of each command, there is potential that some numbers could unintentionally concatenate. If this were to occur with a controller with no error handling while parsing, excessive positions or velocities could be received by the Teensy microcontroller.

3 Design Evaluation

3.1 Test Plan and Results

The printer prototype was evaluated experimentally, with several geometries printed to characterize the accuracy of the combined controller and gantry system. Other requirements of the system, such as the camera resolution, image processing speed, and path generator gantry speed were measured through direct, standalone tests. Lastly, several of the requirements were evaluated using analytical solutions, such as the minimum extrudable material requirement.

Table 3: Test Plan

#	Requirement	Evaluation Method
1	Minimum extrudable material volume of 3 cc	Determined from reservoir piston stroke and bore area.
2	Track a randomly moving target with a maximum speed of 1.5 cm/s and achieve a print resolution of .5 mm	Matlab image processing of print specimen; comparison to desired print geometry
3	Extruder latency < .1 seconds	Determine the time between sending a signal to extruder and the time when material flows out of extruder
4	Image resolution < .2 mm	Calculated from manufacturer's specification and camera placement geometry
5	Image Processing Frame Rate >= 30 fps	The time to collect 120 image frames will be measured with a digital clock using system performance counts - see Appendix 7.
6	Move the target with a velocity > 5 cm/s in the x and y directions	Measure the max speed of each axis of each path generator gantry axis

Prototype

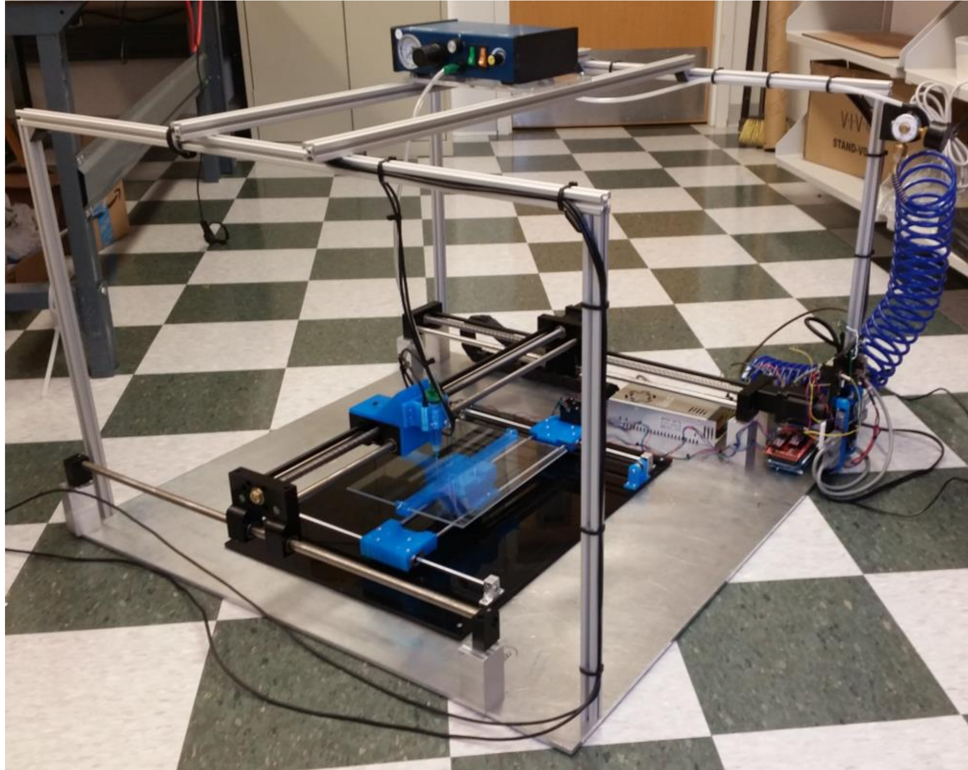


Figure 15: Complete system prototype

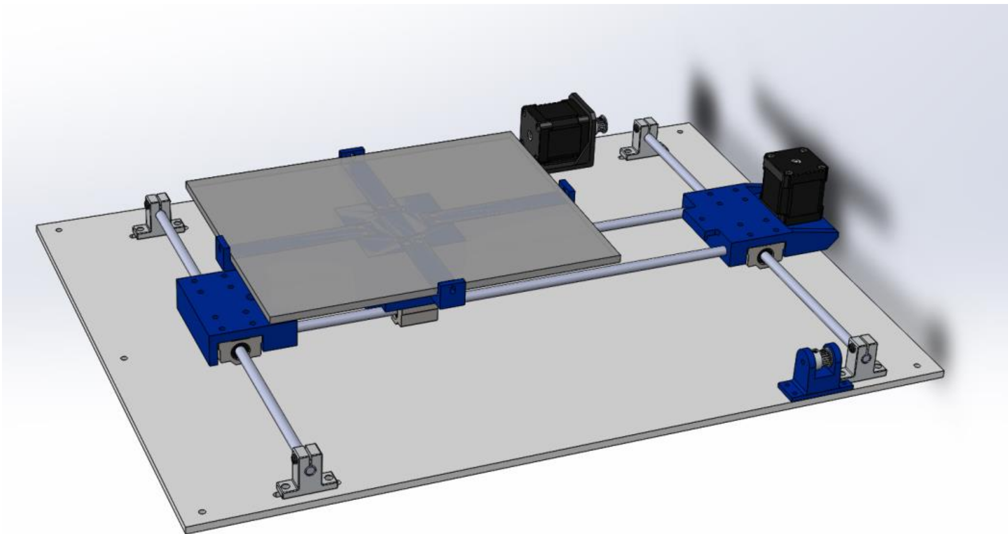


Figure 16: Target gantry prototype design

A physical prototype, seen in figure 15, was constructed to validate the overall sensing and control strategy, as well as to provide a physical testbed for further research on this topic. While a simulation could be applicable for modeling just the control strategy or computer vision algorithm,

a physical prototype can give insight into unforeseen problems or practical issues that can be difficult to account for on a subsystem level simulation.

In addition to constructing the main gantry and its related hardware, a path generator gantry was used to move the target through the workspace. The path generator gantry eliminates variability in the path and speed of the moving target, allowing for direct comparisons to be made from run to run. This allows for iterative tuning of the controller gains and comparisons between different controller strategies. The path generator gantry can also send the X-Y coordinates of the target directly to the controller, closing the loop with hardware (as opposed to cameras locating the object and passing the X-Y position). By passing the X-Y position directly to the controller, the accuracy of the controller program can be evaluated independent of the accuracy of the computer vision system.

Results

Extruder Latency Test (Appendix C)

While the extruder assembly was ruled out-of-scope for the project, the time required for silicone to flow out of the nozzle using an off-the-shelf pressure regulator and valve was assessed. While geometries with continuous beads were printed for evaluation, it may be advantageous to start and stop extruding throughout the print cycle. In order to provide an accurate print, the time required to start extruding the material must be known such that extrusion begins in the correct location.

A pulse signal of varying length was sent to the extruder relay board, causing the extruder driver to pressurize the syringe and extrude material. Pulse time was varied to find the shortest pulse required for material to begin to flow from the nozzle at several pressure settings. The minimum lag time when using the specified silicone compound was found to be 38 ms, meeting the extrusion time requirement. As expected, lag time decreased with increased line pressure. Experimental data and a detailed procedure can be found in appendix C.

Minimum extrudable volume of 3 cc

The minimum extrudable volume constraint was found from measuring the syringe diameter and stroke. The volume of the syringe was calculated to be 3.9 cc, meeting the requirement.

Dynamic Runout Test and Matlab Thresholding

To evaluate the accuracy of the printing approach, the resultant prints were scanned for digital analysis. For this experiment, two different geometries were printed. The geometries include a block ‘M’ and a square. The scanned image was then imported into MATLAB to quantify the performance characteristics. The scanned image was manually scaled and translated so that the image size and location matched that of the template image. Then an exhaustive search of the X,Y, and Theta was performed to finalize registration to the template.

Error was calculated at each pixel in the template image as the deviation in grayscale value between the template image and the corresponding pixel in the scanned image. The average error between the template and scanned image across the whole image was used as a performance metric.

Two primary metrics were analyzed, these include the number of pixels covered with material in the desired area and the number of pixels with material outside the desired area. A visual representation of the data for both prints is shown in Figure 17 and Figure 18. In the figures below, green represents the material deposited in the desired area, red represents the material deposited outside the desired area, and blue represents the areas where no material was deposited in a desired area.

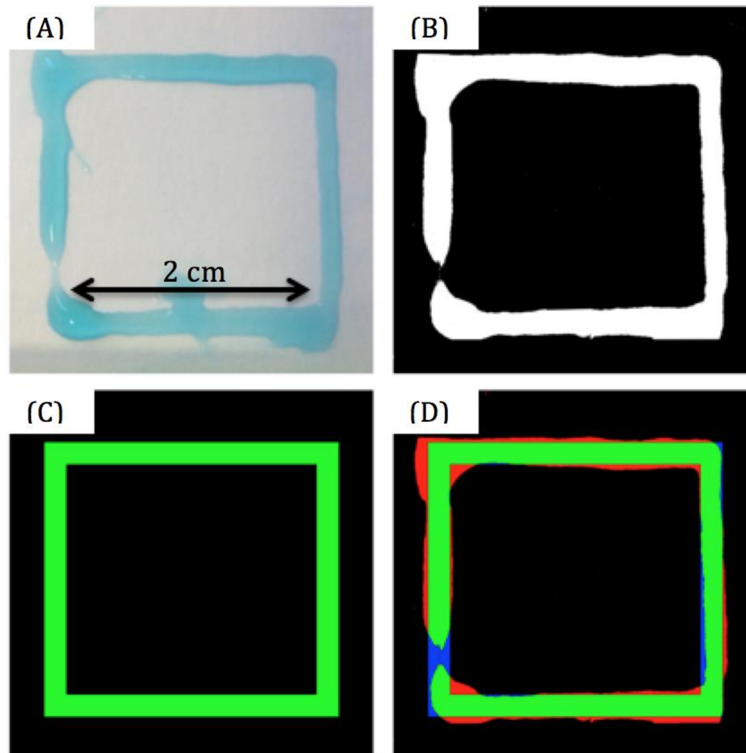


Figure 17: (A) Printed Scanned Square. (B) Printed Threshold Square. (C) Desired M (D) Printed Square Metrics (error in red)

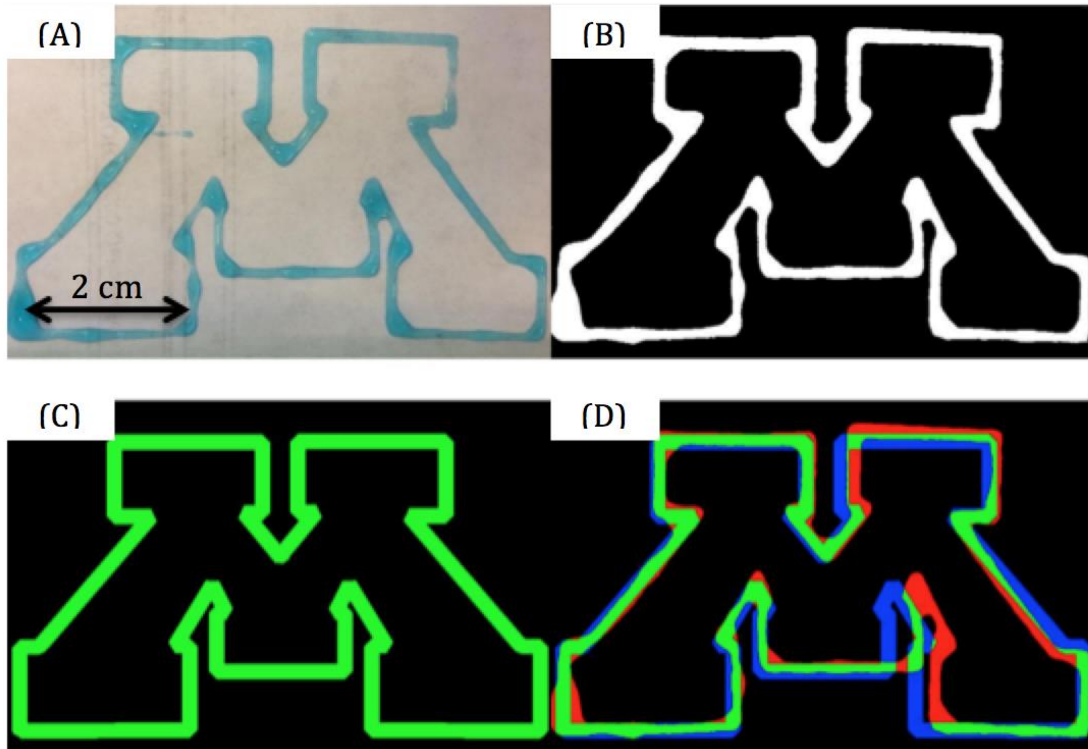


Figure 18: (A) Printed Scanned M. (B) Printed Threshold M. (C) Desired M (D) Printed M Metrics (error in red)

For the printed square, 93.2% of the desired area was covered in material with 79.2% of the total deposited ink. For the printed 'M', 60.7% of the desired area was covered in material with 71.0% of the total deposited ink.

Image Resolution <.2 mm

To evaluate the actual resolution of the image taken from the video feed, the distance between the object and one of the endoscope screens (d) was measured, as well as the camera field of view in the vertical direction (l). The final resolution was calculated to be .11 mm from system geometry.

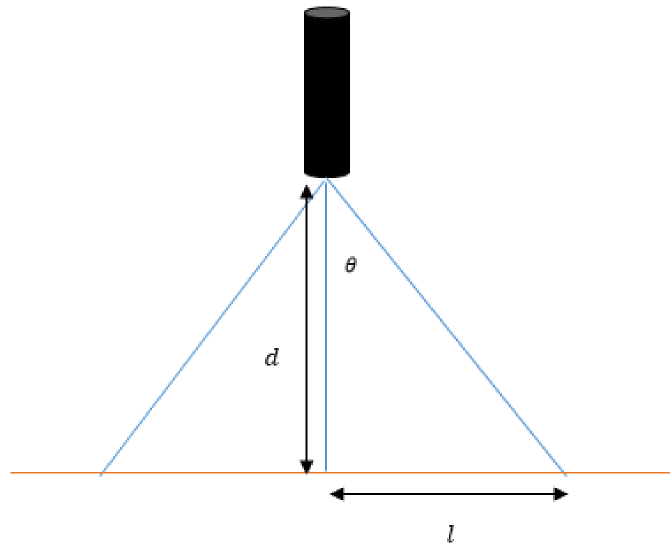


Figure 19: Drawing demonstrating the endoscope looking over its field of view.

Image Processing Rate Assessment

The maximum attainable frame rate using the DB Power endoscope was found to be 23 frames per second utilizing the C++ program outlined in Appendix F. The printing program with computer vision was not evaluated for frames per second, rather the loop time of the entire vision processing program was noted to be between 4-10 ms which demonstrates a minimum of 10 frames (processed) per second (if all loops run at 10 ms).

Target Velocity

The maximum target velocity was calculated from the stepper motor's maximum RPM specification. From this rotational speed and the pulley radius (which moves belts that move the target), one can calculate the maximum target speed to be 6.2 cm/s, meeting the maximum target speed requirement.

3.2 Design Assessment

Pros:

By using the current position data from the path generator gantry, the printer control strategy was validated on a randomly moving target. While originally envisioned for use with the endoscope cameras, one could use the developed control code to print on any planar moving object so long as a sensor can reliably locate the centroid of the target object. This enables further research opportunities looking at different sensing strategies without having to re-design a print controller.

In addition to a working print controller strategy, an operational 2D printer prototype was created, as detailed above. As with the controller, future researchers using this product can build from the current hardware and focus more on sensing and control strategies. Additionally, the path generator gantry can be used to perform repeatable tests.

Lastly, a sensing strategy was selected, designed, and analyzed. An image processing algorithm was created to construct a composite image of the workspace with no obstruction by the extruder nozzle. Initial computer vision algorithms were created to locate identifying features on the object, as well as determine the object velocity from a camera feed. As with the other components of the system, these algorithms give further researchers a "head start" on their project.

Cons:

While both the control and computer vision components could be validated individually, a successful integration of both parts was not completed. Another drawback with the current controller iteration is the print geometry input format. On a high level, the print geometry consists of a 2D array of X and Y points to be printed, with the printer "connecting the dots" to produce the print path. This method is acceptable for simple geometries like those used for the validation of the system, but is inconvenient for end-users who would like to print from a standard Gcode file. Further iterations should consider implementing a controller that can accept and interpret Gcode print files.

A final drawback of the current system is the limited target size that can be tracked, which is due to the relatively small field of view of the endoscope cameras. While acceptable for a proof-of-concept prototype, the limited field of view limits potential applications in the printer's current configuration.

3.3 Future Work

The current iteration of the printer validates the concept of printing on a moving object. Since the prototype has been designed as a research apparatus, the following topics are suggested areas of work that have been identified over the course of the initial system design.

Z-Stage Addition

A Z-stage was omitted from the design to reduce complexity and debugging time. To be a "true" 3D printer, a Z-stage should be added to build up layers and create a 3D structure. A consequence of raising and lowering the cameras is that the apparent size of the identifying objects on the target will change with the distance from the markings. In this way, the computer vision algorithm must scale the relative positions of the identifying marks in its viewing frame to account for the Z-position of the cameras with respect to the markings. Another vision-related complication of adding a Z-stage is the focal length off the cameras. A camera with a variable focal length could be employed to circumvent this issue.

Integration of Computer Vision and Controller Programs

The primary drawback of the current iteration is the lack of integration between the computer vision and motion controller programs. To truly realize the potential of this technology, the computer vision should be processed and used in real-time to provide the feedback to the motion

controller. While a camera-in-hand approach was used for the current design iteration, other work could focus on evaluating different sensing strategies, such as placing several cameras on the superstructure.

Alternate Sensing/Camera Configurations

The addition of more cameras also creates the potential to move from printing on an object moving in planar motion to one that moves in 6 DOF (degrees of freedom) motion. The addition of more cameras to resolve the target's complete position and pose is an important step towards printing on a moving 3D object, such as a biological structure in a body or a complex part moving on an assembly line.

OpenCV Based Algorithm Processing Speed Increase

Part of the OpenCV library includes Nvidia CUDA implementations of many of the functions the image processing program utilizes. If the design were to utilize a Nvidia graphics card, it would be a matter of setting up a system to utilize the CUDA-based algorithms rather than the standard machine versions. This allows the use of parallel computing inside the graphics card which assists the CPU using multithreaded processing and thus more computational power. This would allow the use of more of the baseline FPS reported in Appendix 7 than reported based on the loop times (as low as 10 FPS).

Bibliography

[1] Atala, A. (2011, March 8). *Printing a human kidney*. Retrieved from https://www.youtube.com/watch?v=9RMx31GnNXY

[2] MacDonald et al. (2014, March 13). 3D Printing for the Rapid Prototyping of Structural Electronics. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6766751

[3] Mohd Ghazali, M. I. (2014). 3D printed antennas : metalized plastic. Retrieved from https://etd.lib.msu.edu/islandora/object/etd%3A3194

[4] Weiss, B. K. (2014). Closed-Loop Control of a 3D Printer Gantry.

[5] French, A., O'Neill, J., & Kowalewski, T. M. (2016). Design of a Dynamic Additive Manufacturing System for Use on Free-Moving Human Anatomy. Journal of Medical Devices,10(2). Retrieved February 12, 2017, from http://medicaldevices.asmedigitalcollection.asme.org/article.aspx?articleid=2522804

[6] O'Neill, J., & Kowalewski, T. M. (2014). Online Free Anatomy Registration via Noncontact Skeletal Tracking for Collaborative Human/Robot Interaction in Surgical Robotics. Journal of Medical Devices,8(3). Retrieved February 12, 2017, from http://medicaldevices.asmedigitalcollection.asme.org/article.aspx?articleid=1842398

[7] Corke, P. I. (1996) Visual Control of Robots: High-Performance Visual Servoing. Retrieved from http://www.petercorke.com/bluebook/book.pdf

[8] Murray, D., Basu, A. (1994 May). Motion Tracking with an Active Camera. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 16, No. 5. Retrieved from http://ieeexplore.ieee.org.ezp3.lib.umn.edu/document/291452/?reload=true

[9] Otesteanu, M., Gui, V. (2008, March). 3D Image Sensors, an Overview. WSEAS Transactions on Electronics, Issue 3, Vol. 5. Retrieved from http://www.wseas.us/e-library/transactions/electronics/2008/Editor%20Paper%20OTESTEANU.PDF

[10] Young, E., Jargstorff, F. (2008). Image Processing & Video Algorithms with CUDA [PowerPoint slides]. Retrieved from http://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Image_Processing_and_Video_with_CUDA.pdf

Appendices

Appendix A: List of Terms

Bead-The line of silicone or sodium alginate extruded through the nozzle.

Extruder Head-The extruder head consists of the silicone nozzle, reservoir, pressure line attachment point, and associated mounting hardware.

Extruder Driver-The extruder driver is a modified Fisnar JB1113N pressure regulator. This device has been modified to interface with the extruder relay board, allowing the regulator to be controlled programmatically.

Extruder Relay Board-The extruder relay board uses the 3.3V logic signal from the Teensy microcontroller to drive a relay that interfaces to the extruder driver. A circuit diagram can be found in appendix 6.

Global Localization Camera-This camera mounts to the superstructure and provides a "global" view of the entire workspace.

Main Gantry-The gantry stages that move the extruder head. The main gantry is based off a Newmark Systems ETL X-Y gantry.

OpenCV- Open source computer vision and machine learning library we referenced for feature detection with the centroid method.

Position Based Visual Servoing- Control of the pose of a robot's end-effector relative to a target using the desired and measured relative positions of features extracted from an image captured by a camera carried by the robot

RAMPS Board- A circuit board that fits as a shield on an Arduino Mega. The circuit board houses all the electronics needed to control the moving gantry stage, including plug ins for the stepper drivers and outputs for a servo motor.

Reservoir-The portion of the extruder containing the silicone to be extruded, as well as the piston that is acted on by air pressure from the extruder driver.

Superstructure-The aluminium extrusion structure that hold the global localization camera and extruder driver

Path generator gantry-The gantry stages that move the moving target throughout the workspace

Workspace-The 2D space on the baseplate defined by the limits of travel of the moving target. The workspace contains all possible locations where the main gantry can extrude a bead, however the main gantry's limit of travel are larger than the workspace.

Appendix B: Data Packet Structure

This communication protocol is used when sending from the motion controller to the Teensy microcontroller. The major image processing and control algorithms are run on a computer (better performance than a consumer laptop), which assembles a data packet. This packet is sent over serial to the Teensy microcontroller, which parses the command and handles the low level stepper motor commands. The data packet contains X and Y velocity commands (and Z if 3D printing is enabled), as well as a binary extruder driver command. Figure 26 details two example data packets. Note the size of any velocity command is 6 digits (4 digits in front, up to 2 decimal places). In addition to the generalized move/velocity/print commands, several other pre-programmed moves such as homing and diagnostic tests can be run by sending specific codes over serial as well.

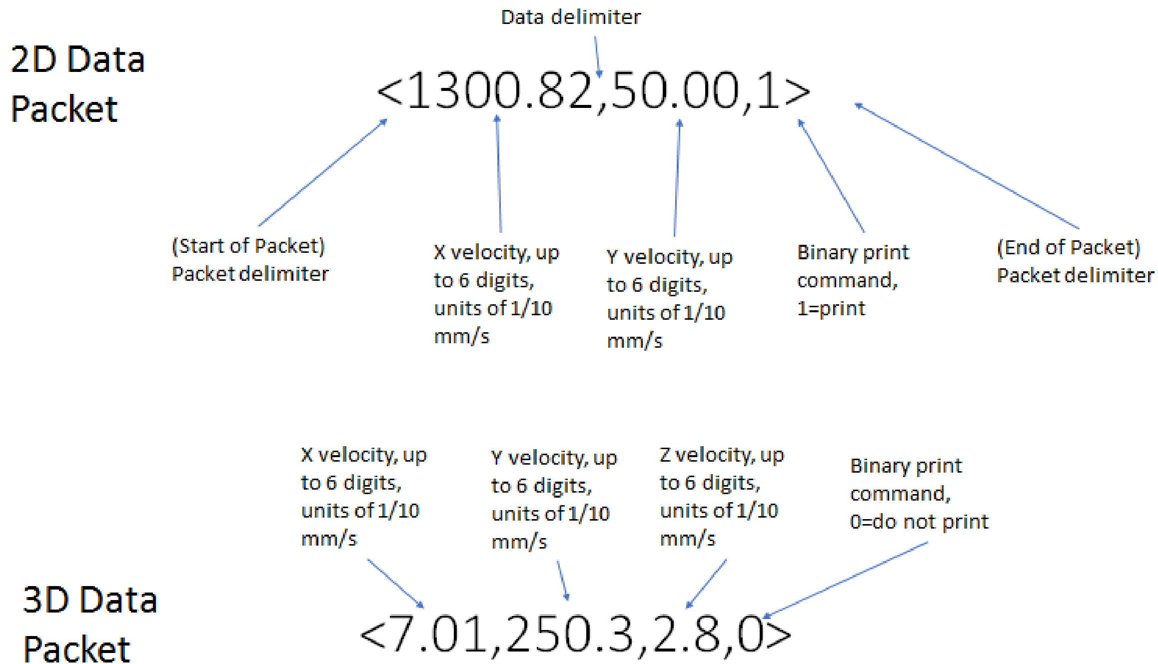


Figure 20: Annotated packet contents diagram

Appendix C: Extruder Latency Characterization

A pulse-width based approach was employed to find the latency of the extruder subsystem. Figure 27 shows a schematic diagram of the subsystem. The input signal, a square pulse of varying length, was generated using a standalone Arduino microcontroller. A short pulse time of 10 ms was initially chosen, and increased until the minimum pulse length was found for silicone to flow from the nozzle. Trials were conducted at 60,70, and 80 psi. As seen in figure $$, latency decreases with increasing pressure, as expected.

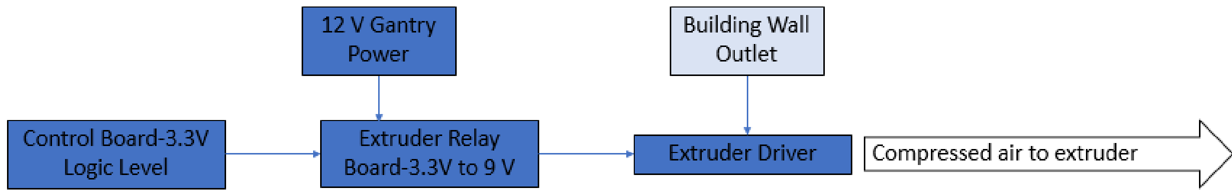


Figure 21: Extruder Latency Test Setup

Pressure	**Pulse time (ms)**	**Flow?**
60 Psi	10	No
	100	Yes
	50	No
	60	No
	70	No
	80	Yes
	75	Yes
	72	No
	74	Yes
70 Psi	50	No
	60	Yes
	55	No
	58	No
	59	No
	60	Yes
80 Psi	50	Yes
	40	Yes

	30	No
	35	No
	38	Yes

Figure 22: Extruder Latency Data

Appendix D: Confirmation of Unobstructed Endoscope FoV with 1.5” Needle

The spacing of the endoscopes from the extruder was designed to create a 1.4 cm x 7.2 cm area of overlapped FOVs to facilitate a transfer in feature detection from one endoscope to another. A 1.5” needle length was then selected to prevent the syringe body and plastic end of the needle from obstructing any of the camera’s line of sight. The image in Figure 28 is drawn approximately to scale.

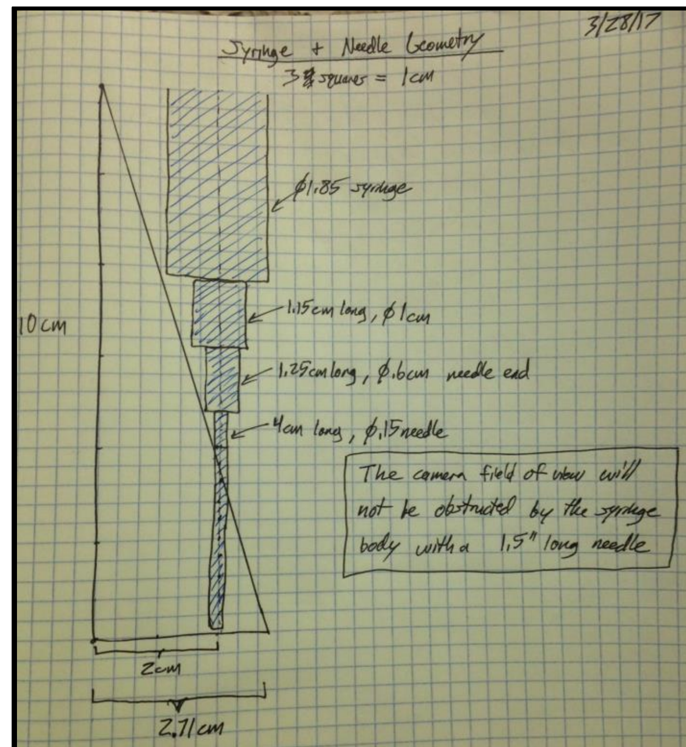


Figure 23: Camera line of sight obstruction estimation

Appendix E: Extruder Relay Board

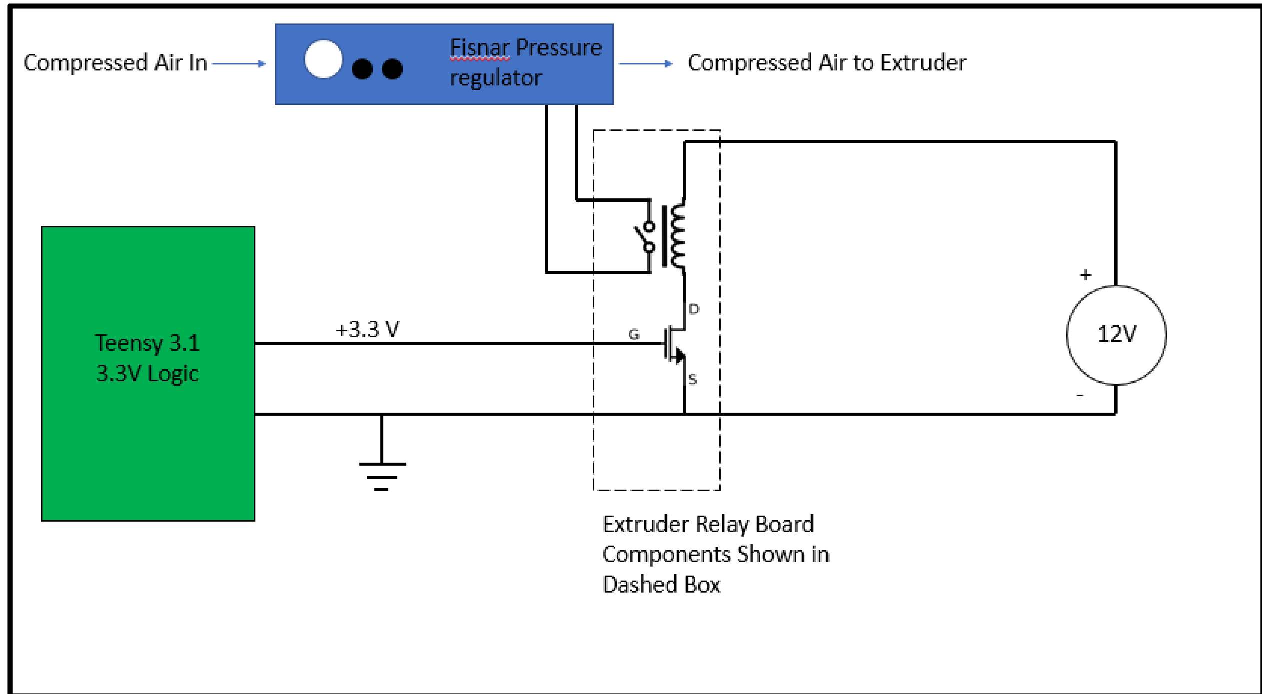


Figure 24: Extruder Relay Board Schematic

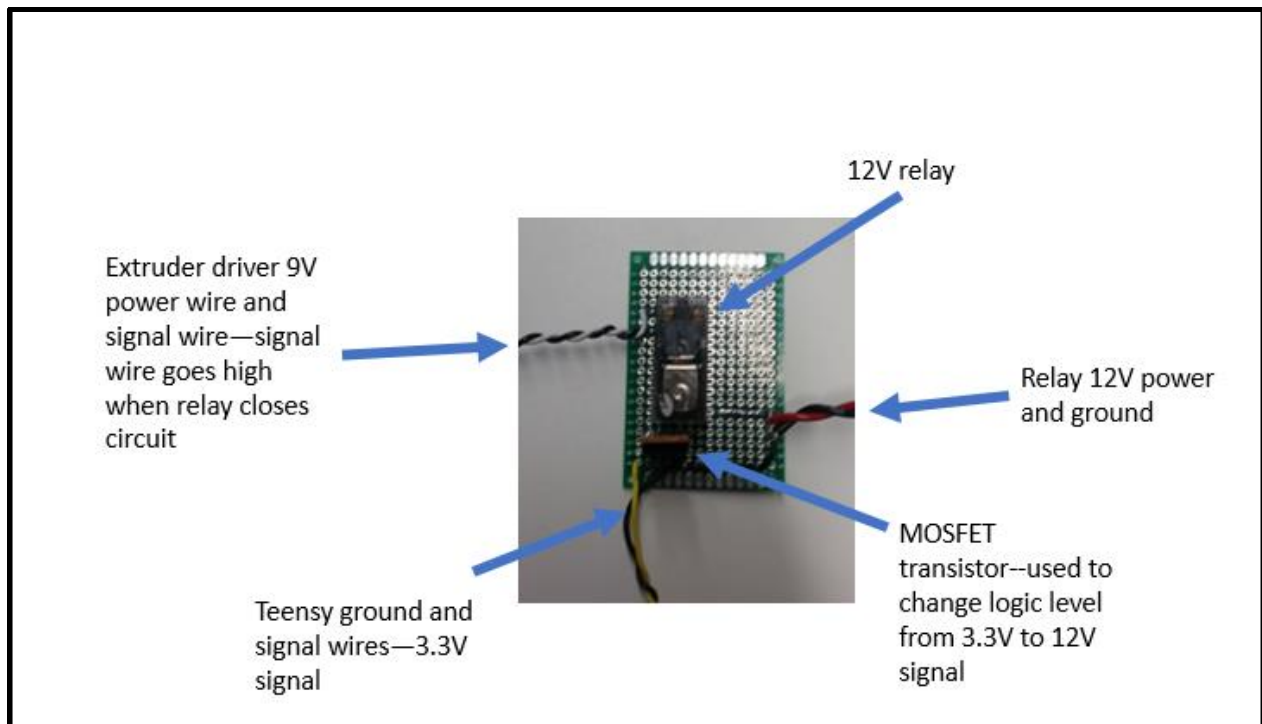


Figure 25: Extruder Relay Circuit Implemented on Development Board

Appendix F: Open CV Frame Rate

In a tracking system, the idea is to sample enough information about a target to calculate its position, velocity, and acceleration. The sampling rate of this system is directly proportional to the frame rate, or frames per second, that our camera-computer system can process. The program will get and process data as follows:

1. Ask the serial port connected to the camera for the next frame and store it as a multidimensional array (2D pixel grid with RGB values for each pixel, so effectively a 3D array)
2. Process the frame for the tracked features using the Hough Circles OpenCV algorithm.
3. Determine current extruder position relative to features
4. Determine *desired* extruder position relative to features
5. Move extruder based on differences between current and desired position.

Accessing data is cheap with respect to computational power, but asking for data (new frame from camera) and processing data (iterate through pixel array) are expensive in that sense. As a gross test of frame rate, a C++ code was developed using some built-in libraries found in OpenCV along with timing functions as part of the C++ standard library. All this program does is ask a camera for 120 frames and save them to virtual (processing) memory while the program is running; no other processing is performed on the frames. Before and after the loop a system clock is asked for the number of system counts. Each system is different, so the standard C++ library defines a variable within time.h called CLOCKS_PER_SEC will report the number of system cycles that have occurred since the clock had been started. With this, the number of system clocks that occurred can be converted into the amount of seconds that passed. This is how an approximation of frame rate is found for both RGB cameras in Table 2. However, it should be stated that this code on its own is demonstrating *best case* frame rate since this program does not include additional processing on the frame as will be the case within the image processing program. The C++ code is included below Table 2.

Table 4: Approximate Frame Rates

Specification	Logitech HD Pro Webcam C920	DBPOWER 5M USB Waterproof HD 6 LED Endoscope
Resolution	Up to 2048x1080	640x480
Approximate Frames per Second	30	23

```
#include "opencv2/opencv.hpp"
#include <time.h>
#include <conio.h>

using namespace cv;
using namespace std;
```

```
int main(int argc, char** argv)
{

	// Start default camera
	VideoCapture video(0);

	video.set(CAP_PROP_FRAME_WIDTH, 640);
	video.set(CAP_PROP_FRAME_HEIGHT, 480);

	double fps;

	// Number of frames to capture
	int num_frames = 120;

	// Time
	clock_t time;
	double seconds;

	// Variable for storing video frames
	Mat frame;

	cout << "Capturing " << num_frames << " frames" << endl;

	// Start time
	time = clock();

	// Grab a few frames
	for (int i = 0; i < num_frames; i++)
	{
	video >> frame;
	cout << "frame\n";
	}

	// End Time
	time = clock() - time;

	// Time elapsed
	seconds = (double)time / CLOCKS_PER_SEC;
	cout << "Time taken : " << seconds << " seconds" << endl;

	// Calculate frames per second
	fps = (double)num_frames / seconds;
	cout << "Estimated frames per second : " << fps << endl;

	// Release video
	video.release();
	_getch();
	return 0;
}
```

Appendix G: Computer Vision Detection Methods Explored

Color Thresholding

To detect the desired outline and contours of a target in a digital image, it is important to threshold or enhance contrast of the image color for isolating and locating the desired object. For this application, black and white thresholding is used along with different color thresholding methods explored. Thresholding is the conversion of a color (or RGB) image to a grayscale image based on a set pixel values in a range of 0 to 255. All values above a threshold value are converted to 1 (white) while all pixels below this threshold are converted to 0 (black). To minimize the effect of lighting and improve contrast on the color appearance between the target and background, a range of 0 to 10 is chosen for detecting black object on a white background. A thresholded image with blue and black shapes is shown in Figure **15**. This method was not used in the finalized design due to its dependency on lighting conditions of the surrounding environment.

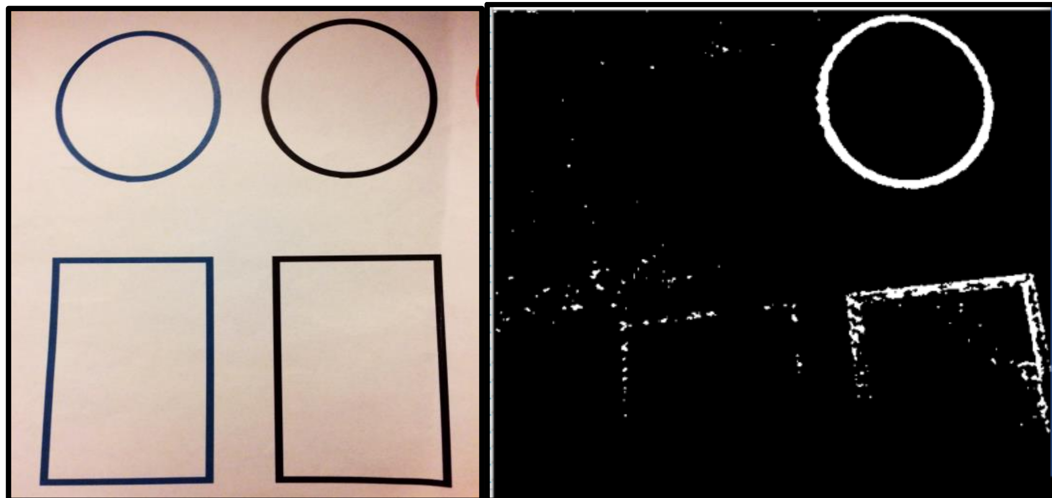


Figure 26: Image before and after thresholding

Contour Tracking

One of the first methods explored was the use of contour tracking on a grayscale image. A snippet of the code is shown below. This code finds the contours of an image and extract the innermost one—the child contour, and draws a bounding box around the contour with the center circled on screen.

```
void findcenter(Mat &frame_current, Mat&video, float &xpos, float &ypos)
{
	findContours(frame_current, contours, hierarchy, CV_RETR_CCOMP,
CHAIN_APPROX_SIMPLE);
	if ((contours.size()>0))
	{
		for (int i = 0; i < contours.size(); i++)
		{
			//hierarchy[idx][{0,1,2,3}]={next contour (same level), previous
contour (same level), child contour, parent contour}
```

```
                    if (hierarchy[i][2]<0)
                    {

                            //drawContours(video,contours,i,(255,0,0),3);
                            BoundingRectangle = boundingRect(contours[i]);
                            xpos = ((BoundingRectangle.x + BoundingRectangle.width / 2.0)
);

                            ypos = ((BoundingRectangle.y + BoundingRectangle.height /
2.0));
                            //Circle the center and draw lines on the target
                            circle(video, Point(xpos, ypos), 20, Scalar(0, 255, 0), 2);
                            line(video, Point(xpos, ypos), Point(xpos - 10, ypos),
Scalar(255, 0, 0), 2);
                            line(video, Point(xpos, ypos), Point(xpos + 10, ypos),
Scalar(255, 0, 0), 2);

                            x = (xpos - 320)*7.2 / 640;
                            y = (ypos - 240)*5.2 / 480;
                            cout << "<" << x << " , " << y << ">" << endl;
                            //writeSerial(port, x, y, 0);
                    }
            }
    }
    else { cout << "no countour\n"; }
}
```

Frame Differencing

Another method explored is frame differencing. It essentially takes two frames of a video feed and compare them. The moving object detected will be the object that has changed position between the two frames. This method is perfect for fast moving object that appears small on the screen. See code snippet below:

```
void findcenter(Mat &frame_current, Mat &video)
{

    findContours(frame_current, contours, hierarchy, CV_RETR_CCOMP,
CHAIN_APPROX_SIMPLE);
    if ((contours.size()>0) ) {

            vector< vector<Point> > largestContourVec;
            largestContourVec.push_back(contours.at(contours.size() - 1));
            for (int i = 0; i < contours.size(); i = hierarchy[i][0])
            {
                    //hierarchy[idx][{0,1,2,3}]={next contour (same level), previous
contour (same level), child contour, parent contour}
                    //if (hierarchy[i][3] <0 ) {
                            objectBoundingRectangle =
boundingRect(largestContourVec.at(0));
                            //objectBoundingRectangle = boundingRect(contours[i]);
                            int xpos = objectBoundingRectangle.x +
objectBoundingRectangle.width / 2;
                            int ypos = objectBoundingRectangle.y +
objectBoundingRectangle.height / 2;

                            //Circle the center and draw lines on the target
                            circle(video, Point(xpos, ypos), 20, Scalar(0, 255, 0), 2);
```

```
                            line(video, Point(xpos, ypos), Point(xpos - 25, ypos),
Scalar(0, 255, 0), 2);
                            line(video, Point(xpos, ypos), Point(xpos + 25, ypos),
Scalar(0, 255, 0), 2);

                        cout << xpos << " , " << ypos << endl;

                }

        }
        else { cout << "no countour\n"; }


}
```

Shape Detection and Centroid Detection

After thresholding and filtering out noise using existing functions in the OpenCV library, features can be tracked using methods such as finding the centroid of an object by approximating the target shape using the "approxPolyDP" function. With this algorithm, we are able to differentiate features by the number of sides they have. In this program, features can be identified as triangle, rectangle, pentagon and so on.

After identifying the feature, the centroid of the object is found. Suppose that there is a plate with a region bounded by the two curves f(x) and g(x) on the interval [a,b]. So, we want to find the centroid of plate.

The mass of this plate:

$$\begin{aligned} M &= \rho(\text{Area of plate}) \\ &= \rho\int_a^b f(x) - g(x)\,dx \end{aligned}$$

There are two moments, denoted by M_x and M_y. The moments measure the tendency of the region to rotate about the x and y-axis respectively. The moments are given by:

$$M_x = \rho\int_a^b \frac{1}{2}\left(\left[f(x)\right]^2 - \left[g(x)\right]^2\right)dx$$

$$M_y = \rho\int_a^b x\left(f(x) - g(x)\right)dx$$

The coordinates of the centroid, $(\bar{x}, \bar{y})$, are then:

$$\bar{x} = \frac{M_y}{M} = \frac{\int_a^b x\left(f(x) - g(x)\right)dx}{\int_a^b f(x) - g(x)\,dx} = \frac{1}{A}\int_a^b x\left(f(x) - g(x)\right)dx$$

$$\bar{y} = \frac{M_x}{M} = \frac{\int_a^b \frac{1}{2}\left(\left[f(x)\right]^2 - \left[g(x)\right]^2\right)dx}{\int_a^b f(x) - g(x)\,dx} = \frac{1}{A}\int_a^b \frac{1}{2}\left(\left[f(x)\right]^2 - \left[g(x)\right]^2\right)dx$$

Where,

$$A = \int_{a}^{b} f(x) - g(x) dx$$

The origin of the coordinates is at the top-left corner of the image. Positive x-coordinate points to the right, and positive y-coordinate points downwards. The algorithm finds the centers of designated geometry in the image. This allows detection of translation and rotation, which is further explained in section 2.6.3. The algorithm outputs the image size and the pixel coordinates of the centers. After calibration, a scaling factor will be derived. Therefore, actual positions and orientation will be determined.

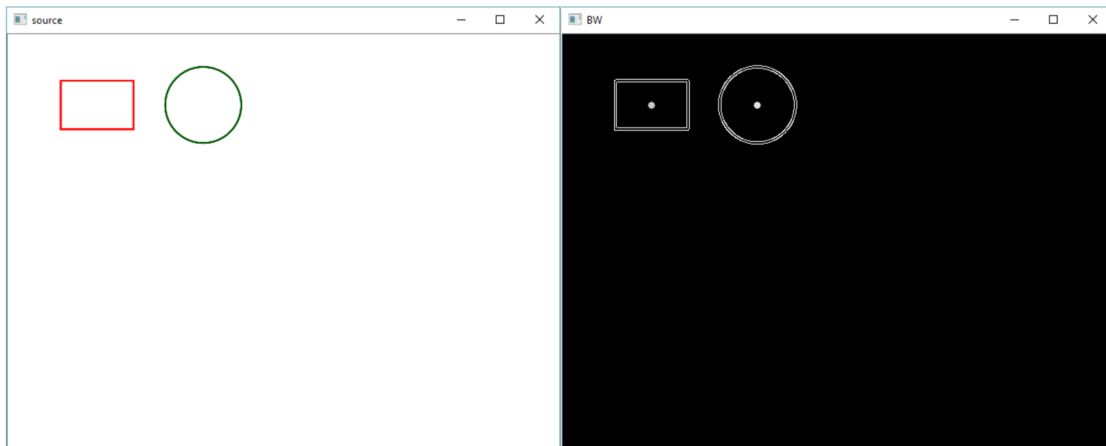


Figure 27: Initial image and processed image with centroid detection

```
Image Width : 640
Image Height: 480
Center of Rectangle is (103.01 , 81.50)
Center of Circle is (226.27 , 81.66)
```

Figure 28: X-Y centroid positions as found from centroid algorithm

Source Code for Centroid Detection:

```
#include <math.h>
#include <opencv2/core/utility.hpp>
#include "opencv2/video/tracking.hpp"
#include <opencv2/core.hpp>
#include "opencv2/imgproc.hpp"
#include "opencv2/videoio.hpp"
#include "opencv2/highgui.hpp"
#include <opencv2/opencv.hpp>
```

```
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <ctype.h>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>

using namespace cv;
using namespace std;

int main()
{
	RNG rng(12345);
	Mat src = imread("C:/Users/jun/Desktop/1.png");
	Mat gray;
	cvtColor(src, gray, CV_BGR2GRAY);
	cout << "Width : " << src.size().width << endl;
	cout << "Height: " << src.size().height << endl;
	Mat bw;
	Canny(gray, bw, 800, 850, 5, true);
	vector<vector<Point>> contours;
	findContours(bw.clone(), contours, CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE);
	vector<Moments> mu(contours.size());
	vector<Point2f> mc(contours.size());
	vector<Point> approx;
	Mat dst = src.clone();
	int a = 0;
	int b = 0;
	int c = 0;
	int d = 0;
	for (int i = 0; i < contours.size(); i++)
	{
	approxPolyDP(Mat(contours[i]), approx, arcLength(Mat(contours[i]), true) * 0.01, true);
	if (approx.size() == 3)
	{
		mu[i] = moments(contours[i], false);
		mc[i] = Point2f(mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
		Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
		circle(bw, mc[i], 4, color, -1, 8, 0);
		printf("Center of Triangle is (%.2f , %.2f)\n", mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
		a = i;
		break;
	}

	}

	for (int i = 0; i < contours.size(); i++)
	{
	approxPolyDP(Mat(contours[i]), approx, arcLength(Mat(contours[i]), true) * 0.01, true);
	if (approx.size() == 4)
	{
		mu[i] = moments(contours[i], false);
		mc[i] = Point2f(mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
		Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
		circle(bw, mc[i], 4, color, -1, 8, 0);
		printf("Center of Rectangle is (%.2f , %.2f)\n", mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
```

```
                b = i;
                break;
        }

        }

        for (int i = 0; i < contours.size(); i++)
        {
        approxPolyDP(Mat(contours[i]), approx, arcLength(Mat(contours[i]), true) * 0.01, true);
        if (approx.size() == 5)
        {
                mu[i] = moments(contours[i], false);
                mc[i] = Point2f(mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
                Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
                circle(bw, mc[i], 4, color, -1, 8, 0);
                printf("Center of Pentagon is (%.2f , %.2f)\n", mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
                c = i;
                break;
        }

        }

        for (int i = 0; i < contours.size(); i++)
        {
        approxPolyDP(Mat(contours[i]), approx, arcLength(Mat(contours[i]), true) * 0.01, true);
        if (approx.size() > 10)
        {
                mu[i] = moments(contours[i], false);
                mc[i] = Point2f(mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
                Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
                circle(bw, mc[i], 4, color, -1, 8, 0);
                printf("Center of Circle is (%.2f , %.2f)\n", mu[i].m10 / mu[i].m00, mu[i].m01 / mu[i].m00);
                d = i;
                break;
        }
        }
        //printf("Labels: %d , %d , %d %d \n", a, b, c, d);
        imshow("source", src);
        imshow("BW", bw);
        waitKey(0);
        return 0;
}
```

Hough Transform

Hough transforms such as Hough circles are used in detecting the shape of the object and outputting the center location of that shape as pixel coordinate. The existing functions in OpenCV library utilize the basic principle of Hough transforms, which are introduced below.

When there is a line in the format of $y = m_0 x + b_0$ in the image space, the corresponding mapping of the line in polar coordinates will look like: $y = -(cos(\theta)/sin(\theta))x \quad + r/sin(\theta)$. For every point (x_0, y_0) in the image space, there is a set of (r, θ) values that make up lines that intersect at this point. As shown in Figure 19, for every point (r, θ) in this set, there is a corresponding line that goes through point (x_0, y_0) in the image space. For each (x_0, y_0) value, all

(r, θ) values that pass it can be plotted as a sinusoid. Doing this for all points, the number of sinusoids that intersect in hough space will indicate the number of points on the same line in image space. Therefore, a threshold value can be set to determine the minimum number of points detected that can be considered a linear line.

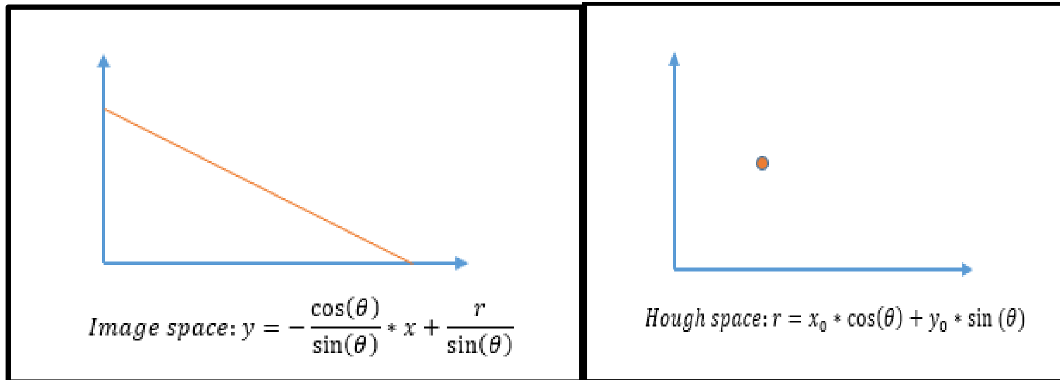


Figure 29: Hough Transform Line Detection

The Hough Circle transform works in a similar fashion. In this case, the parameters used to define a circle are x coordinate of center, y coordinate of center, and radius.The intersection of circles with centers of (x,y) in the Hough space will indicate the center of the actual circle (a,b) in image space.

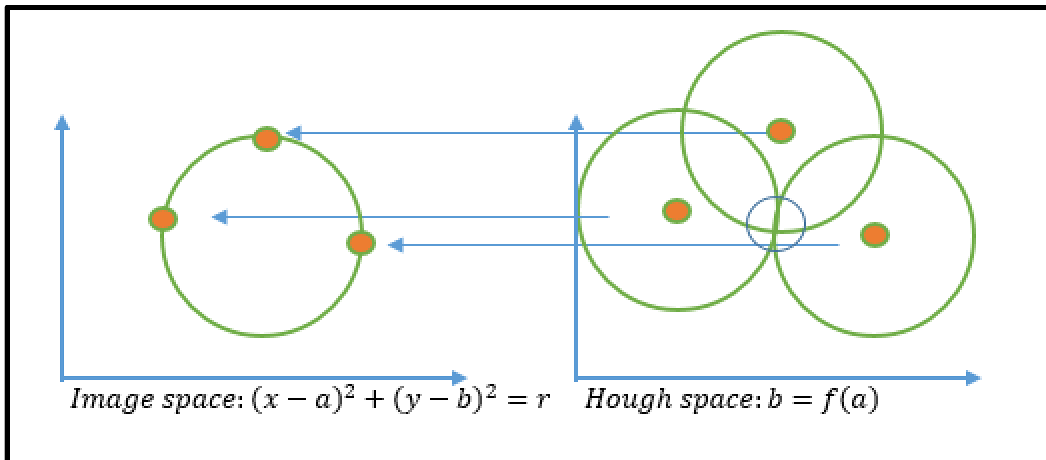


Figure 30: Hough Transform Circle Detection

Two small features on the target will be chosen to obtain the location and orientation of the target. As shown in Figure 20, the angle of rotation of the target from one frame to the next can be calculated using the law of cosines, as given in equation 1. It is assumed that the rotation is within 180 degrees from frame to frame.

$$cos(\theta) = < x_2 - x_1, y_2 - y_1 >. < x_2' - x_1', y_2' - y_1' > / \left(\sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)} * \sqrt{((x_2' - x_1')^2 + (y_2' - y_1')^2)}\right)$$

Law of cosines, where (x_1, y_1) and (x_2, y_2) are the initial coordinates of the two small features on the target, and$(x_1', y_1')$and $(x_2', y_2')$are the coordinates of the two features after the target rotation respectively.

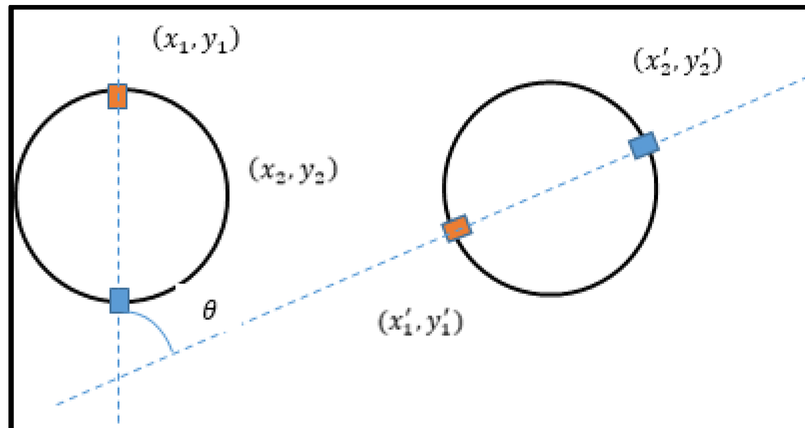


Figure 31: Calculation of rotational angle of the target

Appendix H: Finalized Source Code for Circle Detection

```
#define _CRT_SECURE_NO_DEPRECATE

#include "Serial.hpp"
#include <windows.h>
#include <cmath>
#include <cstdlib>
#include <string>
#include <cstdlib>
#include <cmath>
#include <istream>
#include <iostream>
#include <sstream>
#include <fstream>
#include <time.h>
using namespace std;

//Adding Code for CV
#include <opencv2/core/utility.hpp>
#include "opencv2/video/tracking.hpp"
#include <opencv2/core.hpp>
#include "opencv2/imgproc.hpp"
#include "opencv2/videoio.hpp"
#include "opencv2/highgui.hpp"
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

#include <wtypes.h>
#include <stdexcept>

using namespace cv;
```

```
// End CV Addition

int main()
{
	boost::asio::io_service io;
	boost::asio::serial_port port(io);

	port.open("COM15");//Com5?

	if (!port.is_open()) {
		return -1;
	}
	port.set_option(boost::asio::serial_port_base::baud_rate(115200));	//115200
	boost::asio::io_service io2;
	boost::asio::serial_port port2(io2);


	//Arduino Mega (moving gantry) port: Wont need if using CV
	port2.open("COM16");//Com5?

	if (!port2.is_open()) {
		return -1;
	}
	port2.set_option(boost::asio::serial_port_base::baud_rate(115200));	//115200
	// Could delete above

	double Speed = 10; //Print Speed. units 1/10 mm/s
	double VT[3][5]; // Last 5 target velcoities: vx,vy,omega. Index 1 being the most recent
	double RT[2][6]; // Last 6 target positions: x,y,theta. Index 1 being the most recent
	double dt; // Vision or serial simulated vision loop sampling time in seconds
	int i, j, k; // Index variables
	bool start, done = 0, first = 1;
	clock_t t;  // Loop iteration time
	clock_t tc; // Cummulative time for print direction
	clock_t tt; // Total time
	double tc_s, tt_s; // Time in seconds
	int n = 0; // Print Step Index
	int32_t xs = 0;
	int32_t ys = 0;
	double xs_d = 0;
	double ys_d = 0;
	double xs_prev = 0;
	double ys_prev = 0;
	double xs_new = 0;
	double ys_new = 0;
	int VTx, VTy, VPx, VPy, VEx, VEy, Vsx, Vsy; //Filtered Velocities
	ofstream myfile;

	//FOR CV
	int p1 = 30, p2 = 1200, r1 = 90, r2 = 99, r1max = 300, r2max = 300, tlow = 1000, tlow_max = 3000, ksize = 5, ratio = 3;
	double x_2 = (double)getTickCount();
	float x, y, x_last = 0, y_last = 0, vx, vy;
	vector<Vec3f> circles;
	Mat frame, frame1, frame2, frame1r, frame2r, frame_gray, edge;
	VideoCapture cap(0);
	VideoCapture cap1(1);
	char key;
	if (!cap.isOpened())
	{
		cout << "ERROR! Unable to open camera\n";
		return -1;
	}
	if (!cap1.isOpened())
	{
```

```
		cout << "ERROR! Unable to open camera\n";
		return -1;
	}
	//END CV ADDITION

	//Printing Results
	myfile.open("Results.txt");
	myfile << "xs" << "\t" << "ys" << "\t" << "VTx" << "\t" << "VTy" << "\t" << "VPx" << "\t" << "VPy" << "\t" << "VEx"
<< "\t" << "VEy" << endl;

	//********** square **********
	//double PG[2][5] = { { 0,20,20,0,0 }, { 0,0,20,20,0 } }; // Print Geometry {x},{y} coordinates. units: mm
	//int points = 5;
	//double PD[3][4]; //Print Direction {x,y}, magnitude, print time
	//double Ptime[4];
	//******************************

	//********** UMN Logo ********
	//double PG[2][34] = {{
76,76,72,57,62,62,41,41,43,38,33,35,35,14,14,19,4,0,0,22,22,19,25,27,26,26,50,50,49,51,57,54,54,76 }, {
0,12,12,33,33,44,44,33,33,26,33,33,44,44,33,33,12,12,0,0,12,12,22,18,18,8,8,18,18,22,12,12,0,0 }};
	//int points = 34;
	//double PD[3][33]; //Print Direction {x,y}, magnitude, print time
	//double Ptime[33];
	//****************************

	//Calculates print trajectories and times
	//for (i = 0; i < points - 1; i++)
	//{
	//	Speed = 10;
	//	PD[0][i] = (PG[0][i + 1] - PG[0][i]); // x
	//	PD[1][i] = (PG[1][i + 1] - PG[1][i]); // y
	//	PD[2][i] = sqrt(PD[0][i] * PD[0][i] + PD[1][i] * PD[1][i]); // print segment magnitude
	//	Ptime[i] = PD[2][i] / Speed; //print segment time
	//}

	cin >> start;

	if (start == 1)
	{
		tc = clock() - clock(); // set to 0
		tt = clock() - clock(); // set to 0
		while (true) //Print Loop //CHANGED TO TRUE SO IT STAYS IN LOOP TO TRACK
		{
			t = clock();

			dt = .01;

			//Wont be reading from serial port
			//readSerial(port2, xs, ys); //(UNCOMMENT)

			//Store old positions
			for (i = 5; i > 0; i--)
			{
				for (j = 0; j < 2; j++)
				{
					RT[j][i] = RT[j][i - 1];
				}
			}
			// INSERT CV CODE BELOW
			if (char(waitKey(1)) != 'q' && cap.isOpened()) {

				cap >> frame1;
				cap1 >> frame2;
```

```
//time betweeen two frames
double frequency = getTickFrequency();
double x_delta = 0;
double period;
double x_1 = (double)getTickCount();
x_delta = x_1 - x_2;
x_2 = (double)getTickCount();
period = x_delta / getTickFrequency();
printf("Period is %.2f second\n", period);
if (frame1.empty())
{
	cout << "ERROR! blank frame1 grabbed\n";
	break;
}
if (frame2.empty())
{
	cout << "ERROR! blank frame2 grabbed\n";
	break;
}
//Crop image (need calibration)
int xx1 = 0;
int xx2 = 640;
Rect ROI1(0, 0, 539, 320);
Mat frame1r = frame1(ROI1);
Rect ROI2(255, 0, 385, 320);
Mat frame2r = frame2(ROI2);
//combine 2 images
Size sz1 = frame1r.size();
Size sz2 = frame2r.size();
Mat frame(sz1.height, sz1.width + sz2.width, CV_8UC3);
Mat left(frame, Rect(0, 0, sz1.width, sz1.height));
frame1r.copyTo(left);
Mat right(frame, Rect(sz1.width, 0, sz2.width, sz2.height));
frame2r.copyTo(right);
cvtColor(frame, frame_gray, CV_BGR2GRAY);
createTrackbar("tlow", "Canny", &tlow, tlow_max);
Canny(frame_gray, edge, tlow, tlow*ratio, ksize, true);
imshow("Canny", edge);
if (edge.empty())
{
	cout << "ERROR! blank edge grabbed\n";
	break;


}

createTrackbar("r1 hough", "Video", &r1, r1max);
createTrackbar("r2 hough", "Video", &r2, r2max);

HoughCircles(edge, circles, CV_HOUGH_GRADIENT, 1, frame.rows / 8, p2, p1, r1, r2);
if (circles.size() > 0)
{
	for (size_t current_circle = 0; current_circle < circles.size(); current_circle++)
	{
		Point                    center(round(circles[current_circle][0]),
round(circles[current_circle][1]));
		int radius = round(circles[current_circle][2]);
		circle(frame, center, radius, Scalar(0, 255, 0), 5);
		circle(frame, center, 1, Scalar(0, 0, 255), 5);
		//use acutal radius and meausred radius to calibrate ratio factor
		x = (float)(center.x - 544) / 194 * 101 / 2;
		y = (float)(center.y - 192) / 194 * 101 / 2;
		vx = (x - x_last) / period;
		vy = (y - y_last) / period;
```

```
                x_last = x;
                y_last = y;
                printf("Center of Circle: %0.2f, %0.2f\n", x, y);
                printf("Velocity of Circle: %0.2f, %0.2f\n", vx, vy);
                //writeSerial(port, x, y, 0);
            }

        }

        //Sleep(25); //NEED TO SLEEP?
        namedWindow("Video");
        imshow("Video", frame);
    }

/

    //Trying to use positions in [mm] directly from CV, im sure we need to scale or something
    //RT[0][0] = xs * 6 / 13.5 * 6 / 7.2/2; // ********** Need to set x position in 1/10 mm
    //RT[1][0] = ys * 6 / 13.5 * 6 / 7.2/2; // ********** Need to set y position in 1/10 mm
    // Uncomment above

    RT[0][0] = x*1.5; // ********** Need to set x position in 1/10 mm
    RT[1][0] = y*1.5; // ********** Need to set y position in 1/10 mm
    //End Addition

    //We dont want to subtract last position command, we want to subtract '0' saying 0 was the last position
    for (i = 0; i < 5; i++)//calculates target velocity and saves 5 past velocities for possible filtering
    {
        for (j = 0; j < 2; j++)
        {
            //VT[j][i] = (RT[j][i] - RT[j][i + 1]) / dt;
            VT[j][i] = (RT[j][i] - 0) / dt;
        }
    }

    // Target Velocity mm/s

    VTx = round(VT[0][0]);
    VTy = round(VT[1][0]);

    // Print Velocity
    // SETTING PRINT VELOCITY TO ZERO, JUST TRACKING
    Speed = 10;
    //VPx = round(Speed*PD[0][n] / PD[2][n]) * 160; //(160 steps/mm)
    //VPy = round(Speed*PD[1][n] / PD[2][n]) * 160;
    VPx = 0;
    VPy = 0;
    //END

    //Extruder Velocity
    VEx = VPx + VTx;
    VEy = VPy + VTy;

    //Flipped Signs of Velocities before sending
    writeSerial(port, -VEx, -VEy, 1);
    Vsx = VTx;
    Vsy = VTy;
    //writeSerial(port, 0, 0, 0);

    Sleep(10); // Pause for 10 milliseconds
    t = clock() - t;
    tc = tc + t;
    tt = tt + t;
    tt_s = ((double)tt) / CLOCKS_PER_SEC;
```

```
                        tc_s = ((double)tc) / CLOCKS_PER_SEC;

                        cout << xs << "\t" << ys << "\t" << VT[0][0] << "\t" << VT[1][0] << "\t" << tt_s << "\t" << tc_s;
                        cout << "\t" << VEx << "\t" << VEy << endl;
                        myfile << xs << "\t" << ys << "\t" << VTx << "\t" << VTy << "\t" << VPx << "\t" << VPy << "\t"
<< Vsx << "\t" << Vsy << endl;

                        //if (tc_s > Ptime[n])
                        //{
                        //      n = n + 1;
                        //      tc_s = 0;
                        //      tc = tc - tc;
                        //}
                        //if (n > points - 2)
                        //      done = 1;
                }
        }

        writeSerial(port, 0, 0, 0);
        myfile.close();
        port.close();
        port2.close();
        return 0;
}
```

Appendix I: Comparison of Extruder Head Manipulator Systems

To move the extruder and follow a moving object to be 3D printed on, a robotic manipulator is needed. Several different types of robots were analyzed. These manipulators include a delta style robot, a multiple degree of freedom robotic arm, and a XY gantry system. Since there are multiple variations and brands of each style of robotic manipulator, it is difficult to quantitatively describe and compare each style. However, a more qualitative comparison is shown in table 3 below.

Table 5: Extruder Manipulator Comparison

	Advantages:	Disadvantages:
Delta Robot:	· High Speed · Rigid	· Light Payload · Small Working Volume
Robot Arm:	· Large Working Volume · Small Footprint	· Expensive · Slower Motion

Gantry System:	· High Precision · Heavy Payload · Large Working Volume · Constant Manipulability	· Large Footprint

Appendix J: Sensing Hardware Selection

To garner position and velocity data of the target, some sensor options considered are listed:

- RGB Cameras
- IR Sensors and Emitters
- Ultrasonic Sensors
- LIDAR Sensor
- Time of Flight Sensor

Each of these sensors have different methods for gathering localized data, and each incorporates their own level of uncertainty for determining our desired parameters of location and velocity. With respect to the sensors, each provides their own advantages and disadvantages. Table 2.5.1 lists an abbreviated set of advantages and disadvantages for sensors that can measure the position and velocity of the target.

Table 6: Sensor Advantages and Disadvantages

Sensor	Advantage	Disadvantage
RGB Camera	Large market, high data density (position, color), 1-2 Cameras	Occlusion, Frames per second*, Computational Power
IR Sensors and Emitters	Digital signal (1 or 0), simple to implement	Need many sensors for accuracy, target dependent
Ultrasonic Sensor	Dust-tight, simple to implement	High uncertainty, low resolution, need many sensors for accuracy, target dependent
LIDAR Sensor	High Detail and Accuracy	Cost versus accuracy and speed
Time of Flight Sensor	High Frame Rate (160 FPS), 1-2 Cameras	Low resolution, Occlusion

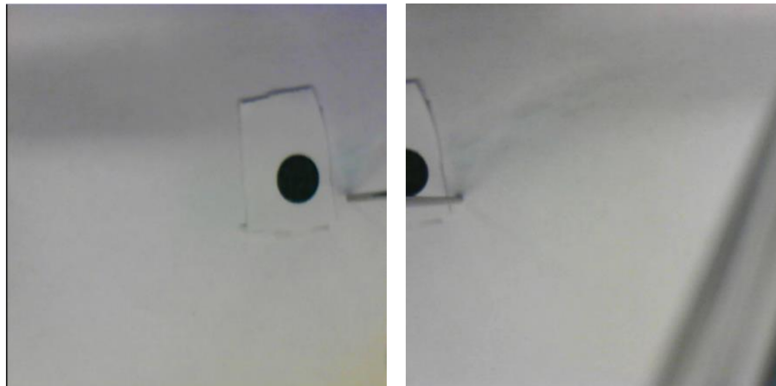
The LIDAR Sensor was the first to be removed from the contention due to its high price. On average, it is an order of magnitude more expensive than any of the other sensors listed in the table.

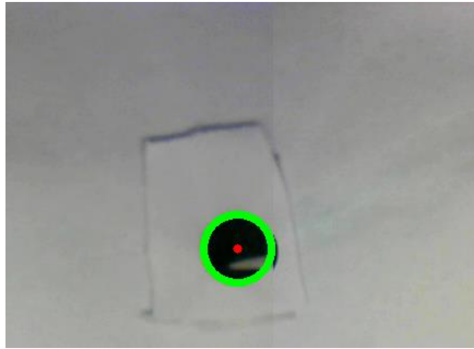
Ultrasonic and IR Sensors were rejected simultaneously. Both sensor types have a dependency on the number of sensors which introduces a hardware requirement on the system. Each of the sensors requires a connection which adds an external requirement beyond the microcontroller for the gantry system. Lastly, the sensor placement within the system to grant sufficient accuracy would be difficult. While the design target presented is constant, the idea is that this will be expanded to arbitrary geometry. With that in mind, no single configuration of IR sensors would work for all geometries. Ultrasonic would fare better against arbitrary geometry, but if something is enclosed within the target it would quickly become problematic.

At this point, two camera options remain: RGB Camera and Time of Flight Camera. The time of flight sensor was rejected due to its pixel density. With a low resolution, the amount of data per pixel would infringe upon the design's ability to attain sub millimeter print resolution. By limiting the speed of the target, a reduction in frames per second from the RGB Camera can be allowed for the large increase in resolution. In addition, better resolution will allow the computer vision algorithms from OpenCV to compute features or centroids with less error.

Appendix K: Field of View Fusion of Two Cameras

There are two cameras mounted on the gantry system. After detection of feature succeed, two cameras are combined to increase tracking quality. In the figure below, there are two circles in both cameras meaning two cameras can both detect the circle. First, two frames are combined side by side as shown below. Original image size is 640 pixels * 480 pixels. After combination, the image size becomes 1280 pixels * 480 pixels. From the center output of detection algorithm, two positions of the circles are obtained. "y1" is relatively equal to "y2" meaning the vertical offset is negligible. The overlap area which is from "x1" to "x2" needs to be deleted. The figures below show this reconstruction.





Programming Code in C++:

```
//Crop image
		int xx1 = 539;
		int xx2 = 895;
		Rect ROI1(0, 0, xx1, 320);
		Mat frame1r = frame1(ROI1);
		Rect ROI2(xx2 - 640, 0, 1280 - xx2, 320);
		Mat frame2r = frame2(ROI2);
//combine 2 images
		Size sz1 = frame1r.size();
		Size sz2 = frame2r.size();
		Mat frame(sz1.height, sz1.width + sz2.width, CV_8UC3);
		Mat left(frame, Rect(0, 0, sz1.width, sz1.height));
		frame1r.copyTo(left);
		Mat right(frame, Rect(sz1.width, 0, sz2.width, sz2.height));
		frame2r.copyTo(right);
```

Appendix L: Main Gantry Print Algorithm

```
#include "Serial.hpp"
#include <windows.h>
#include <cmath>
#include <cstdlib>
#include <string>
#include <cstdlib>
#include <cmath>
#include <istream>
#include <iostream>
#include <sstream>
#include <fstream>
#include <time.h>
using namespace std;

int main()
{
	boost::asio::io_service io;
	boost::asio::serial_port port(io);

	port.open("COM15");//Com5?
```

```
	if (!port.is_open()) {
		return -1;
	}
	port.set_option(boost::asio::serial_port_base::baud_rate(115200));   //115200
	boost::asio::io_service io2;
	boost::asio::serial_port port2(io2);

	port2.open("COM16");//Com5?

	if (!port2.is_open()) {
		return -1;
	}
	port2.set_option(boost::asio::serial_port_base::baud_rate(115200));  //115200

	double Speed = 10; //Print Speed. units 1/10 mm/s
	double VT[3][5]; // Last 5 target velcoities: vx,vy,omega. Index 1 being the most
recent
	double RT[2][6]; // Last 6 target positions: x,y,theta. Index 1 being the most
recent
	double dt; // Vision or serial simulated vision loop sampling time in seconds
	int i, j, k; // Index variables
	bool start, done = 0, first = 1;
	clock_t t;  // Loop iteration time
	clock_t tc; // Cummulative time for print direction
	clock_t tt; // Total time
	double tc_s, tt_s; // Time in seconds
	int n = 0; // Print Step Index
	int32_t xs = 0;
	int32_t ys = 0;
	double xs_d = 0;
	double ys_d = 0;
	double xs_prev = 0;
	double ys_prev = 0;
	double xs_new = 0;
	double ys_new = 0;
	int VTx, VTy, VPx, VPy, VEx, VEy;
	ofstream myfile;
	myfile.open("Results.txt");
	myfile << "xs" << "\t" << "ys" << "\t" << "VTx" << "\t" << "VTy" << "\t" << "VPx"
<< "\t" << "VPy" << "\t" << "VEx" << "\t" << "VEy" << endl;

	//********** square **********
	double PG[2][5] = { { 0,20,20,0,0 },{ 0,0,20,20,0 } }; // Print Geometry {x},{y}
coordinates. units: mm
	int points = 5;
	double PD[3][4]; //Print Direction {x,y}, magnitude, print time
	double Ptime[4];
	//******************************

	//********** UMN Logo ********
	//double PG[2][34] = {{
76,76,72,57,62,62,41,41,43,38,33,35,35,14,14,19,4,0,0,22,22,19,25,27,26,26,50,50,49,51,57,
54,54,76 }, {
0,12,12,33,33,44,44,33,33,26,33,33,44,44,33,33,12,12,0,0,12,12,22,18,18,8,8,18,18,22,12,12
,0,0 }};
	//int points = 34;
	//double PD[3][33]; //Print Direction {x,y}, magnitude, print time
	//double Ptime[33];
	//****************************
```

```
	//********** circle **********
	//double R = 25, w = 1, s = 2, x[100], y[100]; //Circle Radius in mm, print bead width, print raster spacing, coordinates
	//int even = 0;
	//double PG[2][300] = { { 0.0, 2.2, 4.4, 6.5, 8.7, 10.8, 12.9, 15.0, 17.1, 19.1, 21.1, 23.1, 25.0, 26.9, 28.7, 30.4, 32.1, 33.8, 35.4, 36.9, 38.3, 39.7, 41.0, 42.2, 43.3, 44.4, 45.3, 46.2, 47.0, 47.7, 48.3, 48.8, 49.2, 49.6, 49.8, 50.0, 50.0, 50.0, 49.8, 49.6, 49.2, 48.8, 48.3, 47.7, 47.0, 46.2, 45.3, 44.4, 43.3, 42.2, 41.0, 39.7, 38.3, 36.9, 35.4, 33.8, 32.1, 30.4, 28.7, 26.9, 25.0, 23.1, 21.1, 19.1, 17.1, 15.0, 12.9, 10.8, 8.7, 6.5, 4.4, 2.2, 0.0, -2.2, -4.4, -6.5, -8.7, -10.8, -12.9, -15.0, -17.1, -19.1, -21.1, -23.1, -25.0, -26.9, -28.7, -30.4, -32.1, -33.8, -35.4, -36.9, -38.3, -39.7, -41.0, -42.2, -43.3, -44.4, -45.3, -46.2, -47.0, -47.7, -48.3, -48.8, -49.2, -49.6, -49.8, -50.0, -50.0, -50.0, -49.8, -49.6, -49.2, -48.8, -48.3, -47.7, -47.0, -46.2, -45.3, -44.4, -43.3, -42.2, -41.0, -39.7, -38.3, -36.9, -35.4, -33.8, -32.1, -30.4, -28.7, -26.9, -25.0, -23.1, -21.1, -19.1, -17.1, -15.0, -12.9, -10.8, -8.7, -6.5, -4.4, -2.2, 0.0 },{ 50.0, 50.0, 49.8, 49.6, 49.2, 48.8, 48.3, 47.7, 47.0, 46.2, 45.3, 44.4, 43.3, 42.2, 41.0, 39.7, 38.3, 36.9, 35.4, 33.8, 32.1, 30.4, 28.7, 26.9, 25.0, 23.1, 21.1, 19.1, 17.1, 15.0, 12.9, 10.8, 8.7, 6.5, 4.4, 2.2, 0.0, -2.2, -4.4, -6.5, -8.7, -10.8, -12.9, -15.0, -17.1, -19.1, -21.1, -23.1, -25.0, -26.9, -28.7, -30.4, -32.1, -33.8, -35.4, -36.9, -38.3, -39.7, -41.0, -42.2, -43.3, -44.4, -45.3, -46.2, -47.0, -47.7, -48.3, -48.8, -49.2, -49.6, -49.8, -50.0, -50.0, -50.0, -49.8, -49.6, -49.2, -48.8, -48.3, -47.7, -47.0, -46.2, -45.3, -44.4, -43.3, -42.2, -41.0, -39.7, -38.3, -36.9, -35.4, -33.8, -32.1, -30.4, -28.7, -26.9, -25.0, -23.1, -21.1, -19.1, -17.1, -15.0, -12.9, -10.8, -8.7, -6.5, -4.4, -2.2, 0.0, 2.2, 4.4, 6.5, 8.7, 10.8, 12.9, 15.0, 17.1, 19.1, 21.1, 23.1, 25.0, 26.9, 28.7, 30.4, 32.1, 33.8, 35.4, 36.9, 38.3, 39.7, 41.0, 42.2, 43.3, 44.4, 45.3, 46.2, 47.0, 47.7, 48.3, 48.8, 49.2, 49.6, 49.8, 50.0, 50.0 } };
	//for (i = 0; i < 145; i++)
	//{
	//	PG[0][i] = PG[0][i];
	//	PG[1][i] = PG[1][i];
	//}
	//int points = 145;
	//double PD[3][144]; //Print Direction {x,y}, magnitude, print time
	//double Ptime[144];


	//******** Raster Circle w/ no outline
	//double R = 25, w = 1, s = 2, x[100], y[100]; //Circle Radius in mm, print bead width, print raster spacing, coordinates
	//int even = 0;
	//double PG[2][300] = {{0},{50}};
	//for (i = 0; i < 145; i++)
	//{
	//	PG[0][i] = PG[0][i] * .5;
	//	PG[1][i] = PG[1][i] * .5;
	//}
	//int points = 1;
	//x[0] = 0;
	//y[0] = R - s;
	//i = 0;
	//while (y[i] > -R + s)
	//{
	//	if (even % 2 == 0)
	//	{
	//		x[i + 1] = sqrt(R*R - (R - (s)*(i / 2 + 1))*(R - (s)*(i / 2 + 1)));
	//	}
	//	else
	//	{
	//		x[i + 1] = -sqrt(R*R - (R - (s)*(i / 2 + 1))*(R - (s)*(i / 2 + 1)));
```

```
//      }
//      y[i + 1] = R - (s)*(i / 2 + 1);
//      x[i + 2] = -x[i + 1];
//      y[i + 2] = y[i + 1];
//      i = i + 2;
//      even = even + 1;
//}
//points = points + i;
//double PD[3][300]; //Print Direction {x,y}, magnitude, print time
//double Ptime[300];
//for (k = 0; k < i + 1; k++)
//{
//      PG[0][1 + k] = x[k];
//      PG[1][1 + k] = y[k];
//}

//****************************
for (j = 0; j < 200; j++)
        myfile << PG[0][j] << "\t" << PG[1][j] << endl;
myfile.close();


//Calculates print trajectories and times
for (i = 0; i < points - 1; i++)
{
        Speed = 10;
        PD[0][i] = (PG[0][i + 1] - PG[0][i]); // x
        PD[1][i] = (PG[1][i + 1] - PG[1][i]); // y
        PD[2][i] = sqrt(PD[0][i] * PD[0][i] + PD[1][i] * PD[1][i]); // print segment
magnitude
        Ptime[i] = PD[2][i] / Speed; //print segment time
}


cin >> start;

if (start == 1)
{
        tc = clock() - clock(); // set to 0
        tt = clock() - clock(); // set to 0
        while (done == 0) //Print Loop
        {
                t = clock();

                dt = .01;
                readSerial(port2, xs, ys); //(UNCOMMENT)

                                                            //Store old positions
                for (i = 5; i > 0; i--)
                {
                        for (j = 0; j < 2; j++)
                        {
                                RT[j][i] = RT[j][i - 1];
                        }
                }

                RT[0][0] = xs * 6 / 13.5 * 6 / 7.2 / 2; // ********** Need to set x
position in 1/10 mm (removing /2 -> 2cm square)
```

```
			RT[1][0] = ys * 6 / 13.5 * 6 / 7.2 / 2; // ********** Need to set y
position in 1/10 mm

			for (i = 0; i < 5; i++)//calculates target velocity and saves 5 past
velocities for possible filtering
			{
				for (j = 0; j < 2; j++)
				{
					VT[j][i] = (RT[j][i] - RT[j][i + 1]) / dt;
				}
			}

			// Target Velocity mm/s

			VTx = round(VT[0][0]);
			VTy = round(VT[1][0]);

			// Print Velocity
			Speed = 10;
			VPx = round(Speed*PD[0][n] / PD[2][n]) * 160; //(160 steps/mm)
			VPy = round(Speed*PD[1][n] / PD[2][n]) * 160;

			//Extruder Velocity
			VEx = VPx + VTx;
			VEy = VPy + VTy;

			writeSerial(port, VEx, VEy, 1);

			//writeSerial(port, 0, 0, 0);

			Sleep(10); // Pause for 10 milliseconds
			t = clock() - t;
			tc = tc + t;
			tt = tt + t;
			tt_s = ((double)tt) / CLOCKS_PER_SEC;
			tc_s = ((double)tc) / CLOCKS_PER_SEC;

			cout << xs << "\t" << ys << "\t" << VT[0][0] << "\t" << VT[1][0] <<
"\t" << tt_s << "\t" << tc_s;
			cout << "\t" << VEx << "\t" << VEy << endl;
			//myfile << xs << "\t" << ys << "\t" << VTx << "\t" << VTy << "\t" <<
VPx << "\t" << VPy << "\t" << VEx << "\t" << VEy << endl;

			if (tc_s > Ptime[n])
			{
				n = n + 1;
				tc_s = 0;
				tc = tc - tc;
			}
			if (n > points - 2)
				done = 1;
		}
	}

	writeSerial(port, 0, 0, 0)
	port.close();
	port2.close();
	return 0;
}
```