OctoPlus: 3D Printing Error Detection System

Authors: Joshua Bas, Hannah Preston, Lucas Moiseyev: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—We aim to close the loop on Fused Deposition Modeling (FDM) 3D printing. The project will be developed as an extension of an already existing 3D printing monitoring program, OctoPrint. Using inputs from two cameras we will run custom computer vision (CV) algorithms on a Raspberry Pi (RPi) to detect four major classes of printing errors. The system will generate 3D models from user-inputed g-code and then compare them against active prints. Should an error occur, our system will alert the user via a modified version of the OctoPrint web interface. Upon user confirmation, our system will send a stop command to the printer to abort the active print.**

***Index Terms*—additive manufacturing, edge detection, epipolar geometry, fused deposition modeling, g-code, point clouds, stereo imagery, triangulation**

1 INTRODUCTION

Despite rapidly becoming a cornerstone of Maker culture, 3D printing has remained an error-prone method of manufacturing. Due to imperfections in model design, system miscalibrations, and other environmental factors, FDM prints often encounter a range of defects and outright failures. While some errors are insignificant and cause negligible harm to the appearance or function of the printed object, others - especially those that occur early on in the printing process - can result in serious misprints. The primary cost of failed prints is time. Most non-trivial print jobs take several hours to complete, and the manufacturer has to completely restart the process if a severe error occurs. These errors thus critically bottleneck the rapid prototyping process flow.

Though some systems exist to monitor 3D prints via live video stream and others provide retroactive error detection on completed parts, we have yet to see a system that accurately detects errors in real time. We seek to address this critical application gap using a set of computer vision algorithms. The system we will design covers ECE areas spanning Software, Signals & Systems (through image processing), and Hardware. Our approach uses stereo imagery to generate a sparse reconstruction of the current print which can then be compared to the expected model. This model is constructed using the g-code sent to the printer, or the set of commands dictating the movement and feed rates of the extruder head. Other systems employ the use of the Canny edge detector. However, this only accounts for one print orientation and is not robust enough to achieve the high accuracy rates we require. Of course, our design uses a edge detector for redundancy checks. For the design to succeed, an actual error must be classified as such 85% of the time. A check must be completed upon layer completion, and an actual error must be classified within 5 checks. Once an error is classified, the system will notify the user and offer the opportunity to stop the print. The scope of our design is limited to hobbyist level FDM 3D printers. To develop, test, and demo our design, we will primarily use the PrintrBot platform.

2 DESIGN REQUIREMENTS

2.1 Relevant Equations

2.1.1 Camera Equations

In order to determine hardware specifications, we looked at various equations. Angle of view (AOV) is defined as the "angle of a 360-degree circle that is visible" [3]. We take the working distance as the distance from the lens to the object under inspection. Thus, our field of view is "how much of a scene is visible" [3] for a specific working distance. Our depth of view is the difference between the farthest object in the scene that is still in focus and the nearest object that is still in focus, i.e. the distance of focus centered on our object under inspection.

In order to calculate the required sensor resolution r_h by r_v, let p be the number of pixels we allow to detect the smallest object in the field of view. Let s be the size in millimeters of the smallest object in the field of view. Lastly, FOV_h and FOV_v are the horizontal and vertical field of view in millimeters, respectively. Then the general resolution equation is

$$r = p(\frac{FOV}{s}) \tag{1}$$

Considerations for lenses were also based on the general principle that lenses with shorter focal lengths tend to have a deeper depth of view than those with larger focal lengths. However, we discovered that distortion may occur if the focal length is less than 12 mm. The focal length is calculated as

$$f = \frac{W}{2} \arctan(\frac{\theta}{2}) \tag{2}$$

where W is the width of the camera sensor in pixels and θ is our AOV in radians.

2.1.2 Error Assignment Equation

We chose the Hausdorff distance as our error metric. The basic Hausdorff distance [4], where two sets are fixed relative to each other, is

$$H(A, B) = \max(h(A, B), h(B, A)) \tag{3}$$

where A, B are sets (we take our images to be sets) and h is the directed Hausdorff distance. The directed Hausdorff distance is further defined as

$$h(A, B) = \max_{a \in A} \min_{b \in B} \mid a - b \mid$$

H then indicates if there is a point in A nearby a point in B, and vice versa. This definition can be extended to allow one set to rotate and translate with respect to the other fixed set.

2.2 Error Types

There are many different types of printer failures, and causes of those errors. Originally, our system would have detected four key errors:

1. Lack of build plate adhesion
2. Extrusion stopping mid-print
3. Layer shifting
4. "Hair-balling"

However, due to COVID-19, we down-scoped and are only focusing on the build plate adhesion and extrusion stopping errors. We have narrowed our scope to detecting these two errors because they seem to be the most common types. Of course, user experience plays a role into the frequency and type of the errors produced; our system will simply compare the resulting print to the reference model.

2.3 Sensor Coverage Region

The two cameras we use must cover a region of 8.9L x 6.7W x 6.7H inches. This requirement was taken from averaging the build areas of several hobbyist 3D printers, prior to choosing the PrintrBot as our sole development and testing platform. The PrintrBot's build area is 7.3L x 4.6W inches. (Considering that strictly following the larger coverage requirement will likely introduce noise to our system, we can solve this issue by defining a bounding box of computation specific to the printer in use.) Committing the coverage region to match the larger averaged volume rather than the PrintrBot's will better allow for cross-printer testing later on.

2.4 Error Check Rate

Our system must check the printed object upon layer completion. The PrintrBot has a max extruder translational speed of 150mm/sec and a standard nozzle tip of 0.4mm; converted to millimeters, the print bed area is 185.42L x 116.84W. We can divide the bed up into units of 0.4mm width strips, each having a length of 185.42mm. One strip would take the PrintrBot 1.236 seconds. The worst case time for the PrintrBot to complete one layer is 361.036 seconds, or about 6 minutes. Of course, the best case time (for one layer) consists of the PrintrBot extruding at a point, without having to move. Given the bed size and the default speed of the extruder, we think that checking the object upon layer completion is reasonable. However, we do recognize that the extruder speed and actual layer size is dependent on the specific print. The hardware and software implementation will have to be optimized in order to meet this requirement for smaller layer areas.

2.5 Error Detection Rate

We also want the system to detect an actual error within 5 error check cycles. Following from the above discussion, our worst case time from error introduction to error detection would be 30 minutes. Furthermore, after 5 cycles, the layer height would be about 2mm. However, the average case does not reflect this upper time limit. We believe that this worst case time limit and the 2mm layer height until detection are reasonable.

2.6 Error Detection Accuracy

We want our system to be highly accurate. After studying the literature, we found that current systems with less complicated algorithms can detect errors with a maximum accuracy of 80%. With our stereo imagery algorithm, and our fallback of edge detection, we believe our system can perform with an accuracy of 85% – that is, an actual error is classified as an error 85% of the time.

2.7 False Positive Rate

Our false positive rate must be within 20% – an error is indicated when a real error has not occurred 20% of the time.

2.8 False Negative Rate

Our false negative rate must be within 10% – a real error is not classified as an error 10% of the time.

2.9 Runtime

We require that our system is able to function for at least 6 hours. This requirement is derived from averaging test-print print times. A long runtime is necessary because the objective of the project is to save the time invested into the print; the shorter the print job, the less useful our system becomes.

2.10 Size and Weight

The size of the final device should not be more than 6L x 3W inches. The final weight of the system should not exceed 4 pounds.

2.11 Validation Plan

Testing the system's accuracy, false positive rate, and false negative rate involves running our system on a print. Initial tests would look like:

1. Begin a print job
2. Pause the PrintrBot and our system
3. Physically induce one of the four targeted errors
4. Resume the PrintrBot and our system
5. Record whether the error was detected within 5 error checks

To test more efficiently, we can use a model designed to testbench 3D printers. This test print contains hard slopes and overhangs that are more likely to cause errors. We also can design our own test vector that contains errors at predefined locations. For our minimal viable project, we should limit our testing to a couple of print models; once we can meet system requirements with those prints, we can work towards achieving more reliable results with a greater number of objects. While unit testing our algorithms, we will be using the 2014 stereo dataset from Middlebury [6].

In order to test the device weight restriction, we can use a simple scale. Likewise, to test our runtime requirement, we can initiate a print that is expected to last for more than six hours. System status reports are timestamped and logged into a file. To validate that our cameras achieve the correct coverage region, we can just visually confirm that the camera setup results in the correct coverage. For more automated testing, we can optionally take the area of the trapezoid of coverage. To get the vertices of this trapezoid, we can follow the computation described below [1]:

1. Let $\tau = \frac{\epsilon}{\cos(\theta_2+\psi)}$. If $\tau > T$ or $\theta_2 + \psi < 90°$, stop the calculation.
2. If $P_0 = (0,0)$ and $\psi = 0$, the FOV is made of vertices $\{p'_1, p'_2, p'_3, p'_4\}$, each point p'_i being composed of $\{x'_i, y'_i\}$:

$$\begin{bmatrix} h \times \tan(\psi) \\ \frac{h}{\cos(\psi)} \times \{+\tan(\frac{\theta_1}{2})\} \\ h \times \tan(\psi) \\ \frac{h}{\cos(\psi)} \times \{-\tan(\frac{\theta_1}{2})\} \\ h \times \tan(\psi+\theta_2) \\ \frac{h}{\cos(\psi+\theta_2)} \times \{+\tan(\frac{\theta_1}{2})\} \\ h \times \tan(\psi+\theta_2) \\ \frac{h}{\cos(\psi+\theta_2)} \times \{-\tan(\frac{\theta_1}{2})\} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

3. Calculate vertex p''_i by rotating p'_i by ϕ:

$$\begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \begin{bmatrix} x''_i \\ y''_i \end{bmatrix}$$

4. Add actual camera installation information to each p''_i:

$$\begin{bmatrix} x''_i \\ y''_i \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

where x_0, y_0 is the camera installation location; ϵ is the camera installation height; ϕ is the horizontal angle; ψ is the vertical angle; (θ_1, θ_2) are the respective horizontal and vertical viewing angles of the captured scene; and T is the maximum recognition distance. For a particular camera, the installation coordinate is $P_0(x_0, y_0)$ at height ϵ and the recognition distance is τ. We initially implemented this algorithm in order to determine the camera locations, but much later built the camera mount system, so further tests with this implementation were not conducted.

3 ARCHITECTURE OVERVIEW

3.1 Raspberry Pi

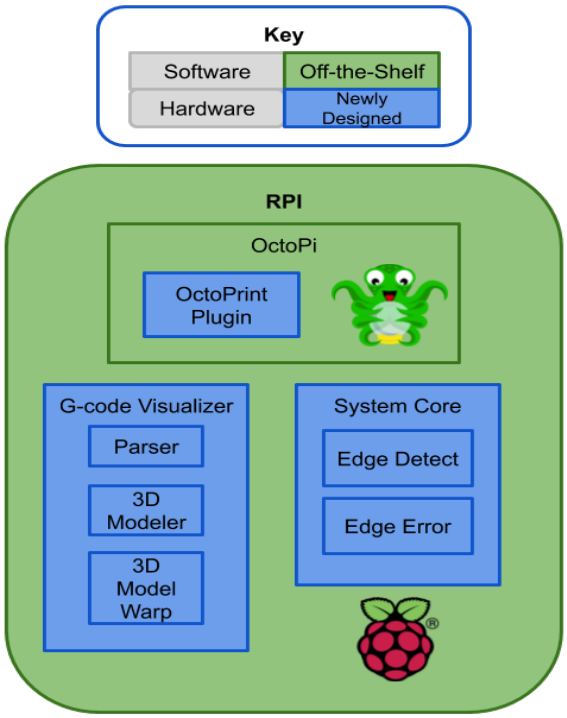

Figure 1: Block Diagram

3.2 OctoPlus Plugin

OctoPrint is a ubiquitous method of sending g-code commands to a 3D printer and monitoring the print status via a live video stream. The software also provides temperature feedback and the ability to print models from an

SD. We chose this platform not only because of its user-friendliness but also because of its well-documented plugin ecosystem. For this project, we created a simple plugin, called OctoPlus, that triggers the camera to take a picture. Once this image is saved, the plugin then executes the implementation of our image processing algorithms. The plugin also provides the user with a form in which the filament's HSV value can be inputted.

3.3 3D Print Modeling

To create our g-code models, we took 3 separate steps. The user will upload their g-code to our system, in which it will parse, and graph in 3D the print at each different z-layer. The function will also have the ability to take in the angle of the camera from the build plate so that the images from the camera will match with the models.

3.4 Edge Detection

This is the basic approach we want to take for both our edge detection. Because we hope to have the point cloud analysis be the primary accuracy check, this will function as a redundancy check to check for any very sudden and obvious changes in the print.

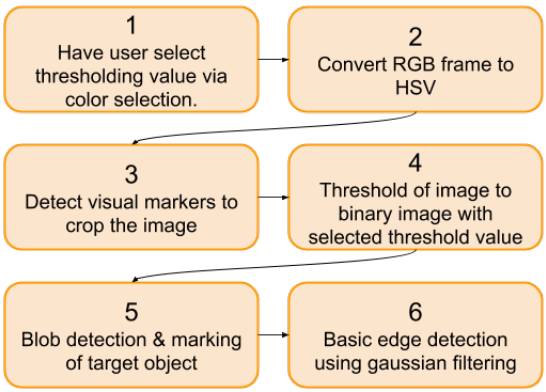

Figure 2: Edge Detection Block Diagram

4 DESIGN TRADE STUDIES

4.1 Solution Approaches

We researched and discussed various approaches to addressing our problem. Our first idea was more hardware-based. We would have designed and implemented a custom single board computer that included all the necessary ports and power management modules required. We would have had a single front-facing camera that solely performed edge detection. This approach was flexible in that we could have made our device blind to the type of 3D printer in use. After discussing our concept with Professor Rowe, we realized that this approach focused too heavily on the exercise of designing the custom board, and did not present a thought-out solution for the computer vision aspect of the project.

Our next approach consisted of the RaspberryPi platform, a front-facing camera, and a corner camera (this was when we were still considering the Ultimaker3 as our printer of choice; it has a built-in camera). Here we would have created a 3D model of the g-code and warped it into a 2D image to compare with the camera outputs. Our primary vision algorithm would have been the Canny edge detector. Because we realized that miscalibrations would introduce system errors that could lead the system to believe that a print error occurred, we included a set of perceptrons that would manage the system parameters according to system output and user confirmation. However, the literature suggested the sole reliance on the edge detector in our context produced inconsistent results, ranging from 60% to 80% accurate. In any case, we desire an 85% accuracy. The perceptron idea also seemed like an effort to fix issues intrinsic to our system anyways; it would be better to figure out those root problems. This report describes and discusses the design we aim to follow for the rest of the semester.

4.2 3D Printer

We initially considered several 3D printer setups. Our main limitations in this area were the budget constraints and access to readily-available printers. At first, we wanted our design to be used on multiple brands of printers. However, we ultimately decided to develop, test, and demo with one type of printer for our minimum viable project.

We first considered the Dremel printers, found in the TechSpark makerspace. Because we would not be allowed to move a Dremel to the demo area, we quickly decided to not proceed with this brand. However, TechSpark did give us permission to experiment with and move the Ultimaker3+ printer. While we began the design phase with this printer in mind, our research led us to the conclusion that the Ultimaker3+ model we had did not allow easy access to its main serial bus; thus getting OctoPrint to connect to the Ultimaker3 would be a huge undertaking beyond the scope of our project.

We are ultimately proceeding with the PrintrBot platform. This type of printer is highly modifiable and offers much easier access to the hardware. Because of the open-end nature of the printer, we are much better able to design fixtures to hold our sensors in place. Lastly, OctoPrint works seamlessly with this printer, so we could easily build our system on the OctoPrint framework.

4.3 Camera & Lens

Choosing a camera required assigning values to our desired FOV and the number of pixels allocated to detect the smallest object. In order to meet our coverage requirement, we decided that an FOV_h of 215mm and an FOV_v of 300mm were sufficient. We think that starting off with 2 pixels to detect a 0.5mm object would be a good starting

point. Following equation (1), our minimum resolution is 860 x 1200. Since we are doing stereo imagery, we need to choose at least two cameras.

The first camera we came upon was the OpenMV Cam H7. This is a camera module that was created for CV usage. It has a lot of different features related to image and video processing, including marker tracking, line/circle/rectangle shape detection, frame differencing, and template matching. We ultimately steered away from this camera because it could essentially do a lot of our project for us, thus leaving the team with very little to contribute besides just programming the module.

We also discovered the Sony Spresense 5MP Camera, prompting us to do a study on its companion board described below. It supports 1920 x 1030 resolution videos at 30 frame/s, which is well above our minimum required resolution. However, the companion board did not meet design requirements, so we could not proceed with this camera.

Next we looked at the Pixy2 CMUcam5 Sensor. The Pixy2 is a camera that has been used for many different maker-type projects, so we obviously wanted to look into it in more detail. In terms our our project and features that it had wouldn't really contribute anything useful besides line tracking. As we continued to look into different cameras, we came to the conclusion that we didn't care as much about the additional features of the module and rather wanted to select a camera module that had good compression capabilities, would be easy to communicate with, and had easily replaceable lens.

The Raspberry Pi Camera Module v2 has a sensor resolution of 2592 x 1944 pixels. This also meets the minimum resolution requirements. Since the RPi only has one CSI port, we can interface with the RPi camera module via CSI, and our serial camera via TTL without interferance. One concern is accounting for the latency differences in serial and parallel communication. However, we decided we can work with the RPi Camera Module v2 as our first sensor.

For our second camera, other choices we looked at were the the UCAM-III Serial Camera Module and the TTL Serial JPEG Camera with NTSC Video. Both of these cameras have a pixel size of 5.6 x 5.6 μm. They also both feature good compression in terms of being able to take relatively high quality photos but still keeping the file size relatively small. We ultimately chose the TTL Serial JPEG Camera because it uses TTL communication and will also be able to work with the module using Python.

However, because we still want the lens to have the widest possible viewing angle, we are using the UCAM-III-116 lens, which has a 116° Horizontal Field of View (HFOV), in comparison to the TTL Serial Module's 56° HFOV and the Lens Board's 56° view. The lenses are interchangeable using a M12 lens mount, so we can easily make this replacement.

4.4 Microprocessor

We considered the Sony Spresense main board primarily because it is required under the Spresense hardware ecosystem. It is a low-power hexacore micro-controller that operates around 156MHz. This board offers 2D acceleration and one parallel camera interface. Since our design implementation changed to involve two cameras, and because its clock rate is lower than other modules, we decided to not proceed with this board.

We also looked at a RaspberryPi variant, the Compute Module 3+. This board actually contains the same processor as the RPi 3B+, but gives the user access to many more IO ports, including the extra CSI camera interface. A disadvantage of the Compute module is its form factor: it plugs into a DDR2 SODIMM connector. To solve this, we found the open-source StereoPi. This carrier board is designed specifically for the Compute module and stereo video capabilities. To this end, we have convenient access to those CSI ports. We also have network access via Ethernet. Lastly, since the Compute module is essentially the RPi 3B+, we also can run Raspbian distributions, such as OctoPi.

The RaspberryPi 3B+ is a familiar platform for the quad-core BCM2837 processor, which is intended for mobile applications. This RPi is specified to operate at 1.4GHz, much faster than the Spresense; this is beneficial since we are running non-trivial computer vision algorithms. This platform also has networking capabilities, allowing us to access the 3D printer remotely. However, while the BCM2837 officially supports two camera inputs, the RPi designers only included one CSI port for a camera. We decided to choose the RPi 3B+ as our platform because it does not require carrier boards, we can run a second camera through UART, and it has networking abilities built-in.

5 SYSTEM DESCRIPTION

5.1 Camera Placement

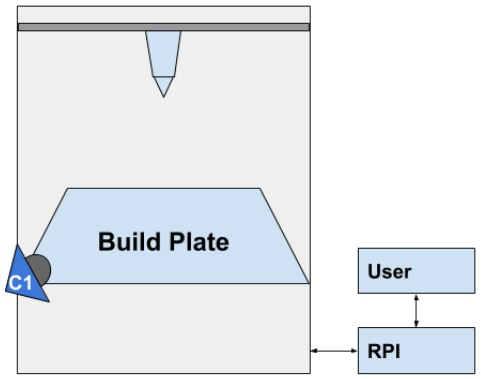

Figure 3: Camera Placement

Our minimum viable project will consist of a two camera setup. C1 and 12, seen in Figure 3, will output the images used for our point cloud analysis. These camera

are pointing at the build plate at an angle such that their depth of field is maximized.

5.2 System Core

The system core interacts with, and is essentially a plugin for, OctoPrint. This module will generate the signal for the error check, and will include function calls to the point cloud analysis subsystem described below. This error check signal is asserted when the command to increment the extruder z-direction is sent to the PrintrBot. Upon device start, the system core will wait for the user to input a g-code file and a list of parameters. These parameters will primarily include the RGB value of the filament the print job will use. This color parameter will be used during the edge detector redundancy check. For debugging purposes, other parameters might include threshold values and options to choose between various similarity metrics. The system core will run the detection algorithms while the print job status is active. Using the OctoPrint framework, we will be able to capture the specific g-code command currently being executed for the particular layer.

When an error is detected, the system core will notify the user via the OctoPrint user interface. The user will be shown a disparity map between the g-code point cloud and the point cloud for the printed object. Then, the user will be able to choose whether the print will continue. If print termination is selected, the system core will send the appropriate set of commands to the PrintrBot. For our minimal viable project, the PrintrBot will be paused while the system core performs these checks. Lastly, the system core will handle the creation and logging of system status in a human-readable format pursuant to our validation plan.

5.3 3D Print Modeling

In order to perform our error detection, we first have to create a model of what the 3D print will look like. STL to g-code converters will often add additional support material to the print that can be peeled away at the end, which means that we could not just use a model from the STL file for the print. This is why we needed to create our own 3D model.

For this project, we can break down the process of creating the g-code model into a number of different components.

5.3.1 Parsing and Slicing

For this project, we only cared about the print command and the XYZ coordinates that followed. In order to do this, we created a specific parsing function that took in a `.gcode` file and flagged each print command `G0` and `G1`. Following each of these is a series of XYZ-F coordinates formatted as `X???  Y???  Z???  F???`. The XYZ are the coordinates of where the print head should be and F is the set speed for the print head. We only needed the XYZ to graph, thus disregarding the F??? command. We separated the XYZ values into three separate lists, where the indexing of each list corresponds to the command order. Our parsing function then returns these three lists.

Following the parser function, we now need to know the z coordinate for each layer. This is so that when it comes time for performing our error checks, we already know what the height of the print should be at any given time. To do this, we simply ran through the z-coordinate list and flagged each time the value increased.

5.3.2 Graphing and Saving

Because our goal was to perform error checks at each new layer of the print, the most efficient way to do this was to create a model of the print at each individual layer. Using python's matplotlib's 3D extension, we were able to give its plot function the XYZ coordinates for our model and it would create a graph. Using the flags from the z-coordinates list, we were able to accurately graph the models, with each new model having an additional layer. Once each new model has been graphed, using python's OS extension, we were able to index the graphs and then save them to be used in the error detection checks.

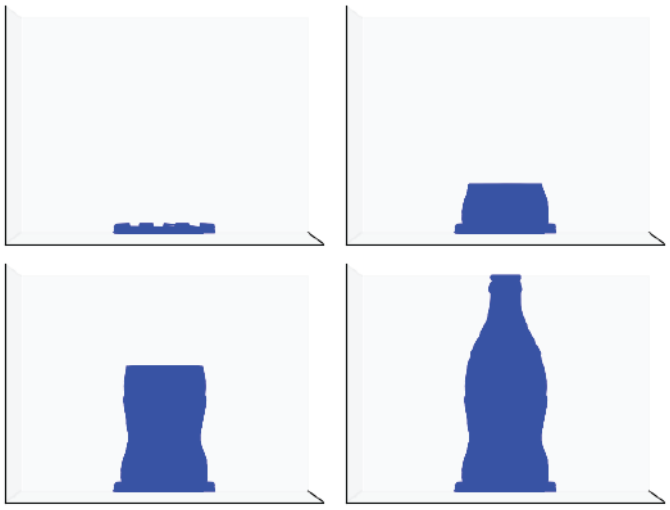

Figure 4: The build up of a 3d printed coke bottle model

As well, we had to take into account the placement of the camera in order to get the models to best match up with the photos from the camera. To do this, we need to take into account two separate angles. Looking at figure 5, θ_1 is the angle from the y-axis to the camera, and θ_2 is the angle from the x-axis to the camera. Using these two angles, as well as the known distance of the camera to the center of the build plate, we can get an as accurate as possible model of what the print should look like on the build plate. Since we weren't able to get any testing done, we went under the assumption that the camera would be where the blue dot is, with $\theta_1 = 0$ and $\theta_2 = 0$.

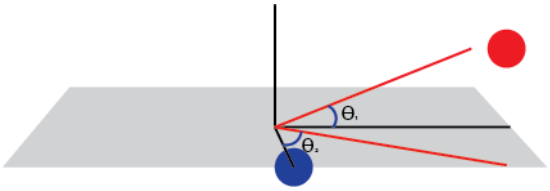

Figure 5: Angle measurements for modeling function

Some things to note when creating a graph in 3D, Python automatically scales all of the axes to 'maximize' the surface area of the thing being graphed, but that ended up warping our models. To fix this, a quick function was created to find the max value among all possible coordinates, and then setting the axes to be a cube of the max value. This prevented any distortion of the print when graphing. In addition, Python automatically includes graph lines and axes markers that needed to be masked out.

5.4 Edge Detection

We are including a edge detection algorithm in conjunction with our stereo imagery algorithm. The edge detection serves as a redundancy check on our system, but it also efficiently accounts for sudden movements on the print.

1. The user will provide some insight to our program by giving it an estimate of the color of the material being extruded. This is to make computations simpler by already knowing what rough color to look for.

2. The camera output will be an image in RGB format, but HSV (Hue, Saturation, Value) is the more common format for CV usage. Because the R, G, and B components of an object's color in a digital image are all correlated with the amount of light hitting the object, and therefore with each other, image descriptions in terms of those components make object discrimination difficult. But because our goal is object detection, roughly separating hue, lightness, and chroma or saturation is effective, because there is no particular reason to strictly mimic human color response.

3. This is in collaboration with the camera placement. We want to find consistent objects where their placements don't often change, and use those as points of reference to be able to get a more specific view of the print. To do this, we will be implementing some sort of scale-invariant feature transform algorithm to best "locate" and crop in on the actual focus of the camera – the print.

4. Next is the conversion of the image into a binary image – aka a black and white image. Using the threshold value from the user, we will attempt to "keep" all the pixel values within a certain range, and then "remove" all the pixels that don't fall within the threshold. This will ideally give us a black and white image, where the white values should be roughly the outline of the print.

5. Once we have the binary image, we can perform basic edge detection on this to smooth out any edges and to make one cohesive item in the image.

6. Once we have the binary image, we can perform basic edge detection on this to smooth out any edges and to make one cohesive item in the image.

The unit tests below exhibit edge images of various objects and their Hausdorff distances relative to the original edge image, that of a gear:

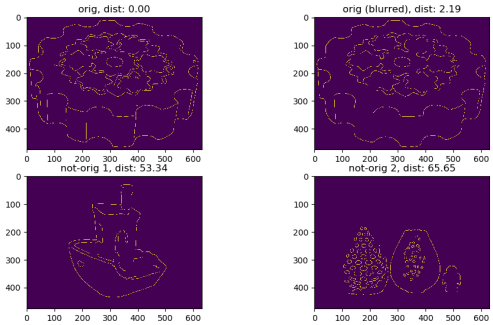

Figure 6: Edge Unit Tests

6 PROJECT MANAGEMENT

6.1 Schedule

Coming out of Spring Break, we were behind schedule and also obviously in need of a re-scope. Our team took 1-2 weeks to re-adjust to virtual life, and then we started working again. For the following 3 weeks, we focused on creating our software (the plugin, g-code models, the edge/error detection). We ended up trying to complete our MVP and almost succeeded, but due to several unforeseen hardware errors, we weren't able to get any integration testing done. So instead we focused on testing the individual software components in the remaining weeks up until the end of the semester.

6.2 Team Member Responsibilities

Following the transition to virtual life, our project objective presents a software problem. Thus, we designated two members on our team to work on the software portions. The third team member was responsible for the hardware integration and testing.

Joshua read through the OctoPrint Plugin API and built the OctoPlus Plugin. He also built the edge detection code block using the OpenCV and scikit-image libraries. In addition to these primary tasks, he implemented the

somewhat-buggy camera placement optimizer that ended up not being used.

Hannah worked on the g-code parsing, interpretation and visualization for the reference model for the error detection.

Lucas focused on the hardware aspects of the project. This included the configuration of the camera and the design of fixtures and keeping the 3D printer working. Throughout the second half of the project, he primarily worked on diagnosing and fixing problems with the printer, including replacing a SOP-23-3 package mosfet on the printer's motherboard and replacing the inductive probe sensor.

Each team member was in charge of unit testing their subsystems, with the goal of having Lucas perform integration testing.

6.3 Budget

As can be seen in Appendix B, we are using our budget primarily for the processing platform and camera choices. In terms of software tool, we are developing with Python since OctoPrint interfaces with the system core plugin in Python. Various support modules – OpenCV, NumPy, MatplotLib, Sci-Kit Image – are used for our computer vision algorithms. MeshLab is an optional system we can use to check our RPC generator. We are using RoboClub's PrintrBot, but since it was assembled prior to our acquiring it, we do not know how much it cost to build. Our code will be controlled via Git/GitHub.

6.4 Risk Management

The biggest risk factor is our hardware limiting the software implementation we have chosen. We are manipulating very large data sets and performing multiple searches during each error check. If the CPU on the one RPi cannot handle the load of interfacing with OctoPrint, transferring data, running the intensive point cloud analysis we have proposed, one solution would be to offload some of the work to the built-in GPU. We would definitely find an existing library and compiler; the alternative would be to learn GPU programming and reading through the RPi GPU datasheet, which is an undertaking that is beyond the scope of this project. Another solution would be to have multiple RPi's working in parallel to process the bulk of the computer vision workload, and interface with OctoPrint on a dedicated RPi. In this configuration, the bottleneck would most likely be the data transfers. Lastly, if the latency is too great, we would have to resort to looking for an entirely new microprocessor.

Another risk factor is the system having a higher false positive rate than expected. Although a high false positive rate is preferable to a high false negative rate, we want to also have a low false positive rate. If we come across this issue, we can tune the various similarity metrics we used in order to mitigate this risk.

A minor risk is that the fixture we construct to hold our sensors and devices is too heavy to meet specifications. A simple solution is to change the fixture's material properties. For example, if our armature is built from 3D printed material, we can just make the infill less dense.

Post Spring Break and during the COVID-19 crisis, we realized that the risks above did not really apply. We could only do nominal tests with our camera setup since our cameras were split amongst the team. So since the RPi ended up supporting at most one camera, the latency issue was not a problem. Further, our camera mount was 3D printed and seemed strong enough to deal with the weight of the camera while meeting specifications. Ultimately, our main risk was not having the time to finish the intricacies of each subsystem, so we tried to manage this by being in communication and staying updated via the mandatory meetings with Prof. Sullivan and our TA.

7 RELATED WORK

A team at the University of Stuttgart[2] aimed to detect detachment, missing material flow, and deformity errors. Surface errors and deviations from the models were also discussed but not addressed due to the complexity of the problem. The paper details the use of edge detection to track the printed object and infer sudden movements. The Canny edge detection is performed during a pre-processing stage to find the upper surface of the print bed. This design achieved a detection rate of 60 to 80 percent and also had a false positive rate of 60 to 80 percent.

8 SUMMARY

In short, we have designed an error detection system for the PrintrBot, a hobbyist 3D printer. The approach we chose encompasses primarily the Software and Signals & Systems areas of ECE. Our system will run computer vision algorithms to generate a digitized 2D model of the current print for comparison to our reference model.

8.1 Future Work

In our design report, we described in length the process to construct and compare the point cloud by converting it into a mesh. Implementing that method and optimizing it for the RaspberryPi would be useful and necessary for this system to work and elaborated upon.

The methods described above of warping a 3D point cloud to a 2D image configuration was easier said than done. One of the hangups was computing the world-to-camera matrix that would allow such 3D-2D alignment. This complication was compounded by not having a camera mount. In lieu of that matrix, it might be possible to find the correct alignment using other methods. For example, researchers from Kyushu University [5] were able to align the 3D mesh with its 2D image by iteratively applying a force. This method was minimally explored, and might

be useful if the world-to-camera matrix is difficult to compute. It might also be useful to determine the performance value of this method in a near-real-time system.

Another avenue of future work might involve using the Hausdorff distance more fully. We just chose to use this metric as our error; however, research [4] has shown that one can use the Hausdorff distance to match an edge image to its corresponding template. This might be useful in a limited scope (e.g, if the orientation and scale were invariant), and so the need to find a world-to-camera matrix might be reduced. It is also useful if one needs to compare portions of an edge image to its template.

A further application of this project is to make the system blind to the printer model. This would expand the number of possible users.

References

[1] Jun-Woo Ahn et al. "Two Phase Algorithm for Optimal Camera Placement". In: *Scientific Programming* 2016 (July 2016).

[2] Felix Baumann and Dieter Roller. "Vision based error detection for 3D printing processes". In: 2016.

[3] Tim Dobbert. *Matchmoving: The Invisible Art of Camera Tracking*. Sybex, Nov. 2012.

[4] Daniel Huttenlocher, Gregory Klanderman, and William Rucklidge. "Comparing Images Using the Hausdorff Distance". In: *IEEE Transactions on Pattern Analysis and Machine Intelligece* 15.9 (Sept. 1993), pp. 850 –863.

[5] Yumi Iwashita et al. "Fast Alignment of 3D Geometrical Models and 2D Color Images using 2D Distance Maps". In: *Fifth International Conference on 3-D Digital Imaging and Modeling (3DIM'05)*. 2005, pp. 164 –171.

[6] D. Scharstein et al. "High-resolution stereo datasets with subpixel-accurate ground truth". In: German Conference on Pattern Recognition (GCPR 2014). Münster, Germany, Sept. 2014.

Appendix A

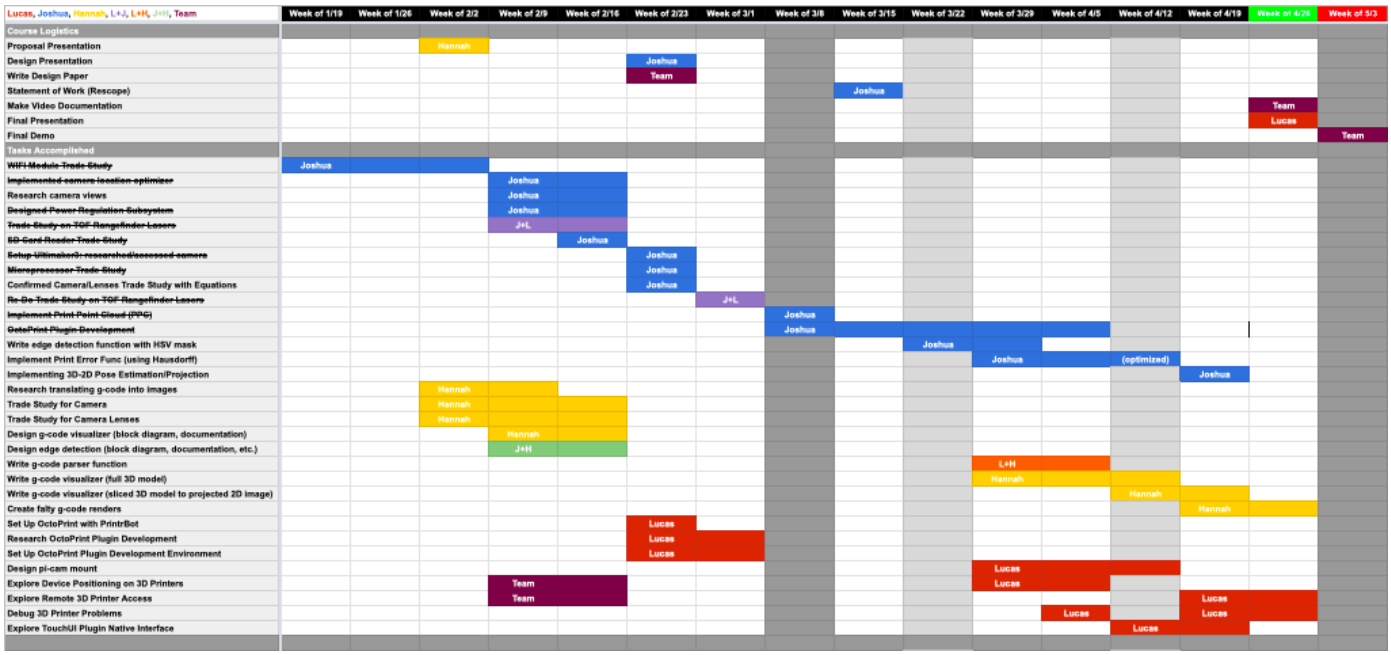

Figure 7: Gantt Chart

Appendix B

Bill of Materials				
Part Name	Function	Source	#	Unit Price
RaspberryPi 3B+	Microprocessor	Adafruit	2	$35.00
RaspberryPi Camera Board v2	Camera 1	Amazon	1	$18.99
TTL Serial JPEG Camera with NTSC Video	Camera 2	Adafruit	1	$39.95
UCAM-III-116LENS	Wide camera lens (116° HFOV)	Digi-Key	2	$10.19
PrintrBot	3D Printer	RoboClub	1	—
Python 3.7	Primary Programming Language	—	—	—
OpenCV	Computer Vision Module	—	—	—
NumPy	Scientific Computing Module	—	—	—
Sci-Kit Image	Image Processing Module	—	—	—
MatplotLib	Plotting Module	—	—	—
Git/Github	Versioning Control Software	—	—	—
MeshLab (optional)	STL Mesh Generator	—	—	—
			Total	**$149.32**