Automatic Error Detection in 3D Pritning using Computer Vision

Stig Andreas Langeland

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

January 2020

Abstract

During recent years Additive Manufacturing Technology, or 3D Printing, has become extremely popular. 3D printing is being actively used in fields ranging from rapid prototyping and rapid manufacturing to bioprinting for tissue engineering. However, it is a very time consuming process as a single object, depending on its size and complexity, may take from only a couple of hours to several days to print. In many cases, errors occur in the middle of a printing process due to misalignment of the 3D printed object, slicing errors or blocked filament extrusion, causing a complete failure of the process. During longer printing processes such errors may occur several hours before we are able to detect them, and a lot of time and material are wasted. If we are able to detect these errors automatically as they occur we may be able to interrupt the process and save both time and material. Severe damage may be caused to a 3D printer if layers of material are continuously added to an object that is misaligned or has detached from the build plate. In this thesis we investigate the possibilities of using traditional Computer Vision algorithms and image processing techniques to automatically detect these errors as they occur. We built a prototype using two different camera angles to analyze both the first layer from a top-down view and the subsequent layers by placing the second camera in front of the build plate. In one of the modules developed in our prototype we managed to compare the 3D printed bottom layer with a simulation of the same layer to detect deviations from the CAD model.

Acknowledgements

I would like to thank my supervisors at The Western Norway University of Applied Sciences, Adrian Rutle and Rogardt Heldal. They have provided me with excellent feedback and support throughout the duration of this thesis.

I would also like to express my gratitude to my friend and PhD student, Frikk Fossdal, who has provided me with invaluable knowledge through workshops and discussions. Without his expertise and support this thesis would not have been possible.

Glossary

Additive Manufacturing A manufacturing process b successively adding material layer by layer.

Clog The molten material is blocked and there is no extrusion.

G-Code Programming language used for Computer Numerical Control machines..

Infill Structure printed inside an object. Infill pattern are generated by a slicing algorithm.

Nozzle The component of the printer that deposits the molten material.

Slicer Software used for converting a 3D model into specific instructions for the 3D printer by slicing the object into layers.

Superpixel A Superpixel is a group of pixels that shares common characteristics. Useful for image segmentation in Computer Vision Applications.

Thresholding The process of creating a binary image by setting all pixels above a certain threshold to white.

Contents

Abstract **i**

Acknowledgements **ii**

Glossary **iii**

1 Introduction **1**

1.1 Motivation 1
1.2 Problem Description 4
1.3 Research Questions 5
1.4 Method 6
1.5 Related Work 7

2 Background **10**

2.1 Additive Manufacturing 10
2.2 Computer Numerical Control 11
2.3 G-Code 12
2.4 Triangle Mesh Slicing 16
2.4.1 Slicing Pipeline 16
2.4.2 STL File Format 20
2.4.3 Slicing Software 21
2.5 Technologies Used 24
2.5.1 Computer Vision 24
2.5.2 Three.js - A JavaScript 3D Library 28

3 Problem Analysis **29**
3.1 Why Computer Vision? 30
3.2 Problems in FDM 3D Printing 31
3.3 Technical Problems in FDM 3D Printing 33
3.3.1 Clogged Nozzle 34
3.3.2 Warping and Deformation 37
3.3.3 Print detachment 41

4 Design and Implementation **43**
4.1 The Prototype 43
4.2 System Overview 45
4.3 First Layer Verification Process 49
4.3.1 What are we looking for? 49
4.3.2 A modified 3D printing pipeline 53
4.3.3 Thresholding the First Layer 56
4.3.4 Contour Approximation 66
4.4 Fault Detection Simulation Framework 68
4.5 Nozzle Analyzer 74
4.5.1 Camera Position Relative to Y-Axis 74
4.5.2 Region of Interest 77
4.5.3 Detection of Extruded Material 81
4.5.4 Contour Approximation and Distance Measuring . . . 85
4.6 Training Data Collection for CNNs 89

5 Results **92**
5.1 First Layer Verification 92
5.2 The Nozzle Analyzer 94

6 Conclusion **96**
6.1 Automatic Fault Detection Prototype 96
6.2 G-Code Simulation Framework 98

7 Future Work **100**

List of Figures

1.1 Market size of AM from 2012 to 2021. 2
1.2 Print loosened from the print bed due to bad adhesion 8
1.3 Under-extrusion, Over-extrusion and Good quality surfaces . . 9

2.1 The manufacturing pipeline using CNC machines 11
2.2 The Slicing Pipeline 16
2.3 Phases of the slicing pipeline 17
2.4 The same model printed with 4 different parameters 18
2.5 A 3D model of a gear before and after slicing. 21
2.6 The first input for the addWeighted function. 26
2.7 Second input for addWeighted function. 26
2.8 Result after doing a weighted addition operation on two images using OpenCV. 26

3.1 Tiny microholes 31
3.2 An extruder made up of several different parts 34
3.3 Shrinking in the material causes warping 37
3.4 Warping on a smaller model causes the entire model to bend . 38
3.5 Forces applied on the print object as it cools down 39
3.6 Object displacement during printing 41
3.7 Printing on overhang without support results in spaghetti print 42

4.1 A graphical overview of the main components in our Client-Server Architecture 47
4.2 A 3D printed octopus with flexible legs 50
4.3 The first layer showing all the joints and the finished result . . 52

4.4 A flow chart for the first layer verification prototype 53
4.5 Thresholding algorithms applied using two different thresholding values. 58
4.6 Histogram of the binary images in figure 4.5. The red graph reflects the image with lowest threshold value. 59
4.7 HSV Color Space 63
4.8 The resulting mask after applying algorithm 2 on the image . 64
4.9 Image segmentation algorithms are applied on a snapshot of the first layer print. 65
4.10 Approximation of contours are shown in red 67
4.11 Object in the ThreeJS scenegraph 69
4.12 The simulated G-Code from two different camera angles . . . 72
4.13 Deviation detected between two images 73
4.14 3D Printer setup with a Rasperry Pi controlled camera for monitoring 75
4.15 Simulating a camera projection in Unity3D 76
4.16 Build plate with reference points seen from the front camera setup 78
4.17 Frame from the front camera are cropped to the reference points creating a ROI 80
4.18 UI to find a suitable HSV range for image segmentation 82
4.19 Smoothing of binary image 84
4.20 Calculating a bounding rectangle around the contour 85
4.21 Distance between the nozzle and the 3D printed object 87
4.22 Clients posting and requesting training data through HTTP endpoints 91

List of Tables

2.1 Most common G/M-Codes used in 3D printing 13

3.1 Overview of common 3D printing failures 33

4.1 Pixel distribution of the binary images 60

Listings

2.1 Example G-code program to print a simple square frame. . . . 14
2.2 ASCII Representation of STL file format 20
2.3 Commented G-Code file using the Verbose GCode setting in PrusaSlicer. 23
2.4 Simple addition of two images in OpenCV 25
4.1 A socket in python acting as a client 46
4.2 A socket in python acting as a server 46
4.3 Request object sent from the front-end application 70

List of Algorithms

1 Detect deviations in first layer 55
2 Algorithm for creating the mask used for segmentation 62
3 Calculate ROI points for print area 79
4 Verify nozzle distance 87

Chapter 1

Introduction

1.1 Motivation

3D Printing has been around since the 1980's when Charles Hull started the history of 3D printing by inventing stereolithography in 1984. This invention allowed designers to create 3D models from digital data, which could be used to create concrete objects from 3D models[14]. Although 3D printers have been around for more than 30 years, the topic didn't gain much popularity until recently. The Additive Manufacturing technology was more focused on factory manufacturing, but during the past 10 years affordable desktop 3D printers have become available to millions of people and thus the tremendous growth in popularity lately [2]. As we can see in figure 1.1 there has been a great development from 2012 in the field of additive manufacturing.

There exists multiple different technologies in additive manufacturing and 3D printing for printing in different materials. Fused deposition modelling (FDM) is the most common 3D printing technology. FDM is a 3D printing process that uses a filament of thermoplastic material of which polylactic acid (PLA) is the most common material used.

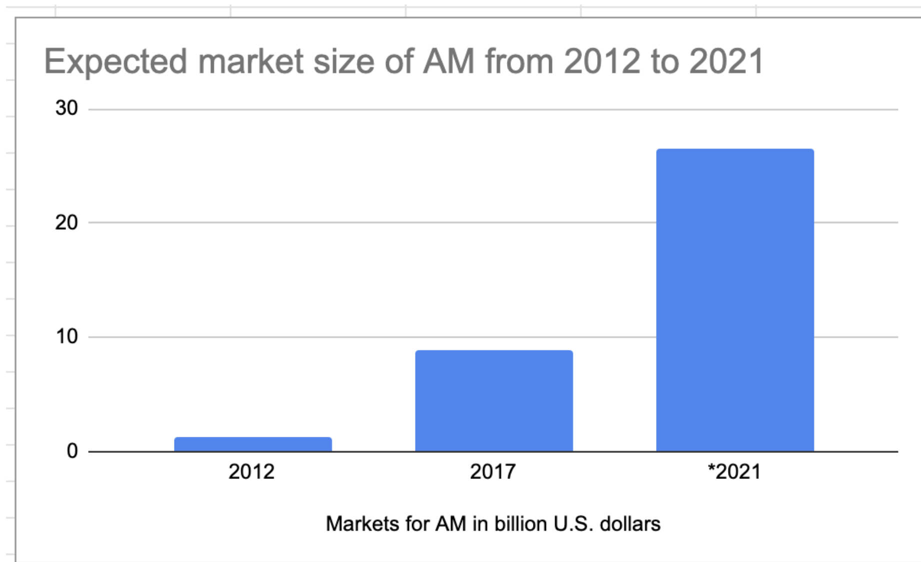


Figure 1.1: Market size of AM from 2012 to 2021[2]. (*) means expected result.

The FDM 3D printing process mainly consists of 3 steps:

- The first step is to design a 3D model of what you want to print. This is usually done using computer aided design (CAD) software like the free and open-source sculpting tool called SculptGL [51]. The model is normally saved in a stereolithography (STL) file format or an Object File (OBJ) format.
- The second step is to slice the 3D model that was created in the first step into horisontal layers and compile it into a sequence of instructions. 3D printing is an additive manufacturing method and thus the machine needs a way to know how the material should be added in a layer-by-layer process in order to achieve the desired result. This is done by using a slicing algorithm[34]. In addition to slicing the 3D model into layers, the slicing algorithm also calculates the tool-paths for each layer. The tool-path defines the path that the printer should follow in order to achieve it's goal. Finally, the tool paths are compiled into G-Code, which is a sequence of commands in which the 3D printer must execute in order to create the given 3D model. The entire sequence of

commands is stored in a file where each line in the file corresponds to a command. There are multiple different slicing softwares [52, 58] that can do this operation.

- The final step is to send all the commands to be executed to the printer. The printer has a firmware installed that can read the G-code and generate control signals to motors and heating elements as well as reading input from sensors. This part of the process may be different based on the type of 3D printing technology or the materials that are used. In most cases the 3D printing material is dispensed through a heated hot-end at a given feed-rate set by the slicer in the previous step. The melted material gets pushed through a Nozzle and builds up the physical version of the digital model layer by layer.

3D printing technology offers a set of advantages in manufacturing and rapid prototyping compared to traditional manufacturing methods [46]. Some significant advantages of 3D printing are waste reduction, the ability to create complex and detailed geometries in a short time frame and to produce useful design in multiple fields. This technology has proven to be especially useful in the medical field to design and manufacture prosthesis of different shapes and sizes [63]. However, like any other technologies, there are a series of challenges and disadvantages involved in the 3D printing process that prevents a large-scale evolution of this technology [19]. The aim of this thesis is to dive into the most common challenges related to FDM 3D printing and discuss to what extent these challenges can be taken account for by using programming rather than upgrading the machines with expensive hardware.

1.2 Problem Description

During the recent years affordable desktop sized 3D printers has been developed and are available for millions of people all around the world spanning from professional designers to manufacturing hobbyists [4]. Even though Computer-Aided Manufacturing (CAM) has been around for a long time most of the work in this field has been targeting large industries, where expensive hardware is required for higher precision and faster production [55]. Therefore, the development of desktop sized 3D printers also gives rise to new challenges to be solved.

Depending on the size, the complexity of the model and the chosen resolution during slicing, the 3D printing process may take from just a couple of hours to several days. This means that the machine will sometimes be working 24 hours a day without anyone consistently monitoring the print to make sure everything is like expected. During this time a lot of different things can go wrong. For instance, the nozzle in which the material gets extruded through, can be clogged so that there is no material flow, or the print may not stick properly to the print bed so that it starts to move. The former example may in worst case damage the machine. These are two examples of common problems in 3D printing that may result in a lot of wasted time and material depending on how late during the process it was detected. If detected early there is a small chance that the project can be paused and fixed mid-print, otherwise the print has to start from the beginning. If an error occur 24 hours into the process, but it doesn't get detected until 6 hours after the problem occurred, then there's a total of 30 hours of time-waste.

Currently the only way of detecting these errors is to manually check by the machine every now and then to make sure the print layers are correct. Even though we detect such an error only minutes after it occurred the print is most likely broken and has to start from scratch again. The aim of this project is to classify the most common challenges in FDM 3D printing and analyse each of them to see if it is possible to automatically detect them

using real time monitoring. If it is possible to detect errors automatically the print can be paused by the monitoring system to make sure minimal time and material are wasted.

1.3 Research Questions

This thesis focuses on error detection in 3D printing by using a camera and computer vision algorithms to classify defects during 3D printing. 3D printers do often run around the clock. Additive manufacturing is a time consuming process and it is not uncommon that errors and defects in the object being manufactured occurs after multiple hours of printing. It is also not common that these errors aren't detected before the object is damaged beyond repair and the whole manufacturing process must be started again from the beginning. Currently, the only way to detect these defects is by actively monitoring the process by eye, which is not a feasible solution in the long run. The research questions are therefore defined as follows:

> *Can we monitor and automatically detect the most prevalent 3D printing errors using Traditional Computer Vision Algorithms?*

The first research question focuses on using *Traditional Computer Vision* [59], which are computer vision algorithms implemented using statistical and mathematical models and image processing techniques. Traditional Computer Vision does not utilize deep learning for image classification and feature detection in images. Training a Convolutional Neural Network (CNN) to classify defects on high precision manufacturing machines like 3D printers requires a large data set for training, but are able to detect more advanced features than traditional computer vision algorithms are able to do [59]. Collecting thousands of images for training an image classification model of defects in 3D printing will be very time consuming and, to best of our knowledge, no such data sets seems to exits. Hence our second research question:

> *Can we create a method for systematic collection of labeled data sets of faults in 3D printing for training Deep Learning models?*

1.4 Method

A literature review has been used to obtain knowledge about the topic and to identify what is known and what is still unknown in the area. The literature review has also been used to gather knowledge about whether the same techniques, or similar techniques, have been used in other manufacturing industries. Normally, a literature review that lists findings in a set of selected articles would end up on the low end of the spectrum if there existed a "reliability" spectrum for the selected papers [10]. This is due to the possibility that the articles may be biased by the author, but it does not mean that the articles are inaccurate [10]. The literature review revealed that there is a lack of research in the area of automatic fault detection in 3D printing processes. Some techniques were attempted and gave some results, but the methods haven't been tested thoroughly and were prone to generating false error alerts.

Next, a feasibility study was done to answer the research questions chosen for this thesis. Since not much research exists on this particular subject, it is difficult to say if it is possible to find a feasible solution to this problem. By doing a feasibility study these questions may be answered by building a proof of concept prototype and apply it on some test case. Based on the literature review we decided to build a prototype and used different image processing techniques to create new and more consistent methods for fault detection. We used an experimental setup using two cameras monitoring the 3D printing process from different angles.

1.5 Related Work

Trends and Challenges of 3D Printing

It is important to address challenges facing the future as the 3D printing technology is developing as well as analysing how the world adapts to the technology. As stated in [19], one of the main issues regarding 3D printing are the technical problems that can arise during a print as a result of unexpected behaviour from the mechanical parts. The paper also concludes that 3D printing technology will continue to evolve in the future as many stakeholders are already seeing the potential of 3D printing. By comparing the rate of evolution of 3D printers with the adaption rate, we get a better view of the potential benefits from automatic error detection in 3D printing.

PCB Defect Detection

A similar project was conducted in [48] where computer vision were used to detect defects in the Printed Circuit Board (PCB) manufacturing industry. Image processing algorithms are applied to frames that are continuously captured by a camera in real time during production to remove background noise. A digital model of the same PCB is then compared to the actual footage of the manufactured PCB by applying the Exclusive or (XOR) operation on the images. The XOR operation will output the differences between the two images, where the differences will reflect the deviation from the desired PCB model.

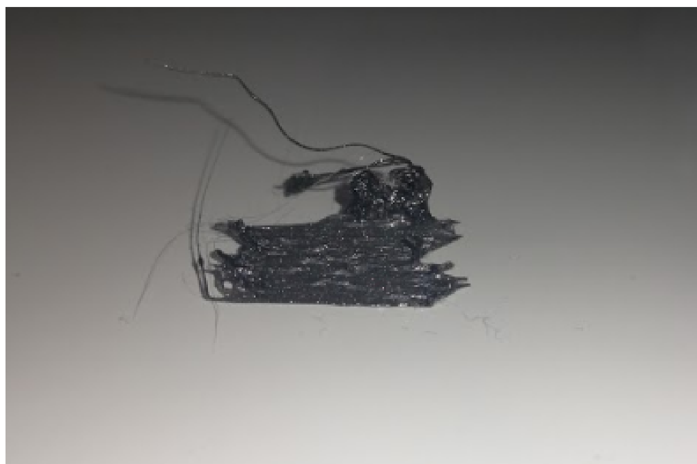


Figure 1.2: Print loosened from the print bed due to bad adhesion

Autonomous in-situ correction using Computer Vision

A lot of different errors ranging from clogged nozzle to loose mechanical parts of the printer [15] might occur in the middle of a print affecting the quality of the physical object being printed, or even cause the print to fail miserably. Section 3.2 introduces the most common and important causes to defects of a 3D printed object. However, almost all of the succeeded prints have one characteristic in common, namely a successful first layer. This also applies to failed prints; If the first layer doesn't succeed, it will most likely cause the rest of the print to fail. Figure 1.2 shows an example of a defect 3D print caused by a bad first layer to illustrate the importance of a succeeded first layer print in order to achieve an optimal result.

The first layer of a print may not succeed due to a series of different reasons, but a common cause is due to over-extrusion or under-extrusion [15]. Over-extrusion means that too much material is flowing through the nozzle, whereas under-extrusion occurs when the printer is not able to supply enough material. An autonomous system containing advanced machine learning algorithms to detect over- and under-extrusion and self-correct in real-time to be able to print reliably at fast rates and high resolutions has been developed in [26].

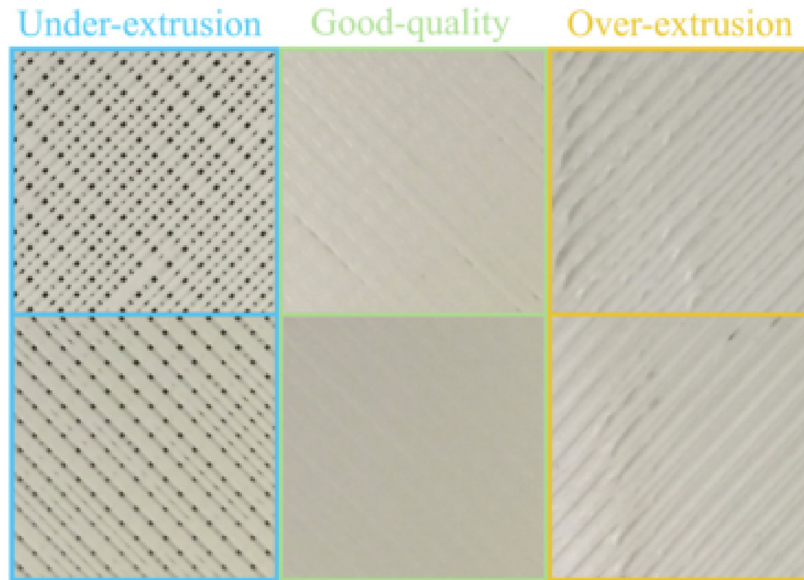


Figure 1.3: Under-extrusion, Over-extrusion and Good quality surfaces

The 3D printing system in this project, based on machine learning, consists of real-time monitoring using a mounted camera and a post-training procedure with a ResNet architecture [18] to train a Convolutional Neural Network (CNN) [28] classification model to detect over- and under-extrusion during the 3D printing process. Images are fed into the model continuously in real-time while 3D printing. If an under-extrusion is detected by the classifier, commands to increase the material flow will be sent to the 3D printer automatically by the system. Figure 1.3 shows what the trained classification model is looking for when trying to detect over- and under-extrusion. Around 120,000 images of each category are prepared to build the classification model, where 30% are used for testing and validation, and the remaining images are picked randomly to be used as training data.

Chapter 2

Background

2.1 Additive Manufacturing

3D printing is the process of creating a physical three-dimensional object from a computer aided design (CAD) model by adding material layer by layer, starting from the first layer at the bottom to the last layer at the top. This process is also called Additive Manufacturing (AM) [11], whereas the more conventional manufacturing method, also called subtractive manufacturing, is the process of removing material from a solid block of material to create the 3D object [7]. Several different materials like paper, powder filament, metals, liquid and thermoplastics can be used in additive manufacturing, where thermoplastics is the most common material used in 3D printing [32]. Which type of materials to chose depends heavily on the additive manufacturing method being used. Fused Deposition Modeling (FDM), which is the technology used in this project, is the most common 3D printing technology and the material being used in FDM printers are normally thermoplastics like Polylactic Acid (PLA) [37].

2.2 Computer Numerical Control

Numerical Control is a way to automate the control of machining tools by using computers. The Computer Numerical Control (CNC) machines are programmed using letters, numbers and symbols to define movement sequences. The letters and numbers are often referred to as *G-Code*, which is the programming language used for most CNC machines, including 3D printers.

G-code consists of a long sequence of movement commands along the X, Y and Z axis for the 3D printer. There is also a fourth axis called "E-axis", which is the stepper motor controlling the extruder of the 3D printer and the flow of material through the hot-end of the printer. The hot-end is the part of the printer in which the filament gets fed into by the E-axis motor and the melted material comes out at the bottom of the hot-end.

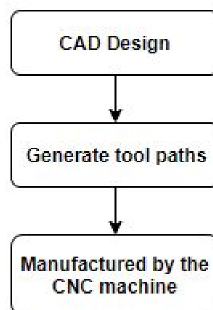


Figure 2.1: The manufacturing pipeline using CNC machines

Figure 2.1 above shows the typical pipeline for a CNC controlled machine [17]. This pipeline was briefly mentioned in chapter 1 with regard to 3D printing. A model of the object one intend to manufacture is created using a CAD software. The second step in the pipeline is generating the tool paths. In 3D printing this step is called *slicing*, where the resulting output is a G-Code file containing all the instructions for the 3D printer to manufacture the object. A detailed explanation of what G-Code is and how it is generated will be given in the next two sections. Finally, the tool paths generated in

the last step are executed as a sequence of commands to manufacture the CAD model.

2.3 G-Code

The 3D printing technology enables us to manufacture physical objects created using different types of material by using a digital model created by a computer aided design software and a 3D printer. In order for this to be possible it is necessary to develop a method to tell the printer how to create the specific object. This is where G-code comes in. G-code is the most used programming language for computer numerical control systems, thus it is also the most widely language used for 3d printers [5]. The same G-code commands may have different meanings for different types of CNC machines, and therefore this section will be focused towards G-code used in 3D printing.

G-code is usually stored in a .gcode file consisting of a long sequence of movement commands along the X, Y and Z axis for the 3D printer, but the file extension may vary for different printers whilst the code syntax remains the same. There is also a fourth axis, called “E-axis”, which is the stepper motor controlling the extruder and the flow of material through the printer’s hot-end. The hot-end is the part of the printer in which the filament gets fed into by the E-axis and the melted material comes out at the bottom of the hot-end. Each line in a G-code file corresponds to a single command, where each command may take several parameters. The most common command stored in a G-code file is usually the G1 command, which tells the printer where to move in a straight line while extruding material. The first 30-40 lines of a C-code file normally consists of a combination of different G-code and M-code commands. The M-codes are commands to set different configurations before the print starts like setting a temperature for the nozzle and the print bed, maximum acceleration along the axis, maximum feed rates and retraction acceleration. Table 2.1 shows a list of the most important G-codes and M-codes used in 3D printing.

G-Code	Description
G1	Linear Movement: The command tells the printer to move in a straight line to a specified X, Y and Z location [30]. G1 X5 Y10 Z9 E2.0 F500 tells the printer to move to position (5,10,9) while extruding 2.0 millimeters of filament with a feed rate of 500 mm/s
G28	Home: G28 performs the homing routine for the axis. If no arguments are given to the G28 command it will move the toolhead to position (0,0,0). G28 X Y will only move the X and Y axis to their respective home positions. It is important to run G28 at the very beginning of a print. When the axis are homed using the G28 command the printer can safely assume that it is in position (0,0) in the X-Y plane and use relative coordinates for the rest of the print.
G10	Retract: Retracts the filament by a length specified by the M207 command[31]. A lot of stringing during a print is usually due to poor retraction settings.
M190	Set bed temp: This command sets a temperature for the print bed and wait for it to be reached before before it do anything else[12]. It is important to wait for the correct temperature in order to get the best print results possible.
M106	Fan on: Turn the cooling fan on
M109	Set extruder temperature: Sets a temperature for the extruder and wait for it to be reached. For printing with PLA an extruder temperature around 215°C is normal and can be set using the M109 command: M109 S215
M226	Temporarily pause the printing process
M24	Start / Resume print

Table 2.1: Most common G/M-Codes used in 3D printing

We now know the very basics of what C-code are and how it works, but how do we go from a CAD model to a sequence of commands the printer has to execute in order to create the physical object? As mentioned earlier in the introduction section a model has to be sliced into layers [34]. During the slicing process of the model a slicing algorithm will generate a large file, the G-code file, that may contain hundreds of thousands of commands based on the triangle mesh of the CAD model, slicing settings and the slicing algorithm used. Listing 2.1 shows a very simplified G-code program for printing a 5x5mm square. In the next section, section 2.4, a more detailed explanation of the slicing process is given, including how decisions made during the slicing process will significantly affect the print result.

```
1
2  M104 S215 ; set extruder temp
3  M140 S60 ; set bed temp
4  M190 S60 ; wait for bed temp
5  M109 S215 ; wait for extruder temp
6
7  G28 ; Home all axis
8
9  G1 X0 Y5 E1.0 ; Move to position (0,5)
10 G1 X5 Y5 E2.0 ; Move to position (5,5)
11 G1 X5 Y0 E3.0 ; Move to position (5,0)
12 G1 X0 Y0 E4.0 ; Move to position (0,0)
13 G1 X0 Y5 E5.0 ; Move to position (0,5)
14
15 M104 S0 ; turn off temperature
16 M140 S0 ; turn off heatbed
17
18 G1 Z30.8 ; Move print head up
19
20 M84 ; disable motors
```

Listing 2.1: Example G-code program to print a simple square frame.

The first lines of listing 2.1 shows how the M-codes are used for configuration of a 3D printer before the actual printing is happening. After the bed and extruder has reached their respective target temperature G28 are executed to move all the axis to position (0,0) in the X-Y plane. This can be done using a limit switch, or an endstop[49], which are mounted at the end of each axis. When the axis touches a limit switch, the switch breaks the circuit to the motor and the print head can't move any further. Thus the printer

knows its exact location and it can begin printing. The printer will first execute the G1 command and move to location (0,5) while extruding 1mm of filament, followed by another G1 command to (5,5) while extruding another 1mm of filament until all commands are executed. At the end another set of configurations using M-codes are executed to turn off the temperature, heat bed and the stepper motors. The result is a physical object, a simple 5x5mm square frame.

2.4 Triangle Mesh Slicing

2.4.1 Slicing Pipeline

As mentioned earlier, FDM 3D printing is all about building up a physical 3-dimensional object by accurately applying material layer by layer. In this section we will explain how we go from a digital 3D model to G-Code commands by using a slicing algorithm to slice the 3D model into layers. Figure 2.2 below shows the pipeline from a digital model, normally stored in a stereolithography (STL) format [5], and convert it to a sequence of G-Codes stored in the resulting G-Code file containing all the instructions for the 3D printer to execute. [21]. STL files are one of the most commonly used formats to store information about 3D models created using CAD software and are therefore widely used in most of the computer aided manufacturing methods for rapid prototyping [33], including 3D printing.

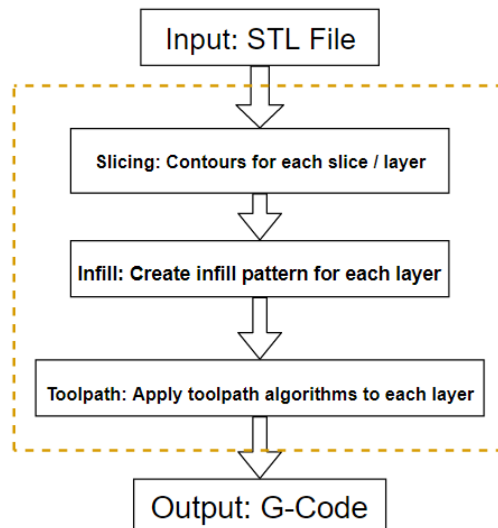
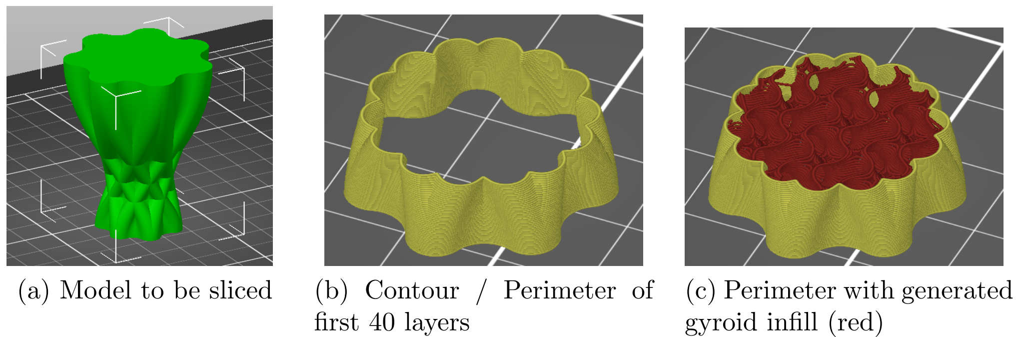


Figure 2.2: The Slicing Pipeline

As there exist multiple methods for slicing, the most common technique for slicing a 3D model stored as an STL is by finding all vertices that defines the contour of each layer and then, if Infill settings are set in the slicing software used, generate infill inside the contour / perimeter of the object.



(a) Model to be sliced

(b) Contour / Perimeter of first 40 layers

(c) Perimeter with generated gyroid infill (red)

Figure 2.3: Phases of the slicing pipeline

Figure 2.3 above shows an example using Slic3r Prusa Edition [45] to slice a model (See: Figure 2.3a) into layers and prepare it to be printed, where each layer consist of many two-dimensional coordinates defining the contour of the object. Figure 2.3b shows what the contour for the first 40 layers of the model looks like, where the first layer is the very first contour at the bottom of the model. One can therefore think of 3D printing as doing 2D printing over and over again by applying small adjustments to the contour for each layer until we get the desired shape.

When the contours are defined we move to the last processing step in the pipeline - generating infill for the contours. Infill is a term in 3D printing that refers to the structure printed inside the perimeter, or inside the object, to make the object stronger and more solid. The amount of infill ranges from 0% to 100% infill, where 0% infill means that only the contours of the object should be printed whilst 100% means that the contours should be completely filled with material to make a solid object. It is most common to use infill of around 10-20% depending on how strong one want the structure to be. Less infill means weaker objects, but the print time and material usage decreases significantly. How much infill that is being generated are set by the "infill"

parameter in the slicing software being used.

The infill inside the contour is an automatically generated pattern for each layer, or each contour, of the sliced model. If the infill parameter is set to 10%, the Slicer will then generate an infill pattern that fills 10% of each layer. There exist a variety of different infill patterns that can be chosen, where different infill patterns can drastically affect the flexibility and strength of the final print. Figure 2.3c shows the contour (in yellow) with an infill of 25% using an infill pattern called "Gyroid". Among many other infill patterns, gyriod has shown to be one of the best patterns for 3D printed models used in technological applications due to its capability to absorb energy [1].

Figure 2.4: Same model printed using 4 different infill settings. The top-left and top-right objects are printed using grid infill pattern with 20% and 50% infill, respectively. The bottom-left and bottom-right have the same infill density as the top ones, but with a different infill pattern called *gyroid*.

As mentioned in section 1.5, the first layer is the most important layer for the print to be successful as it forms the foundation for the remaining layers to be printed on top of the first layer. Therefore, it is common to print the first layer with 100% infill to ensure a good base for the print with good adhesion

go the print bed. If the first layer detaches from the print plate the object might start moving as the printer continues to deposit material onto the remaining layers, and the material will not be applied at the correct positions relative to the object. In a worst-case scenario, if the first layer detaches from the print bed, it can cause severe damage to the hot-end. Figure 2.4 shows the difference between 20% infill and 50% infill using the grid infill pattern (the top two objects) and the gyroid infill pattern (the bottom two objects). Unlike the objects printed in figure 2.4, it is also common to print the last 3 layers with 100% infill to hide the infill patterns behind solid layers which results in a smoother surface.

2.4.2 STL File Format

3D models created using CAD programs can be very complex objects containing a lot of small details at high resolutions. But how are all the details and information about the objects stored?

Usually, a model created using CAD software is stored as an STL file (a file with the .stl extension). This file contains information about all the vertices, all the triangles made up of the vertices and the normal vector, or the normal, of the triangles[22]. As with almost all cases of 3D graphics rendering, the surfaces of the 3D models are comprised by a set of triangles connected together in different angles to create the surface. This is also called a *triangle mesh*[57]. By storing the normal vectors of the triangles and the vertex positions for the triangles the model can be re-rendered from the STL file.

Information about the polygons stored in STL file format can be represented in both ASCII and binary format. An object may consist of millions of triangles and thus a binary representation is usually a more efficient way to store the triangle mesh.

```
1 facet normal nx ny nz
2     outer loop
3         vertex v1x v1y v1z
4         vertex v2x v2y v2z
5         vertex v3x v3y v3z
6     endloop
7 endfacet
```

Listing 2.2: ASCII Representation of STL file format

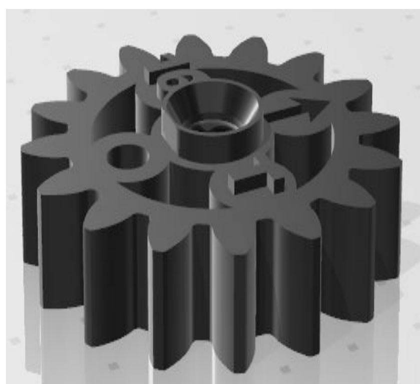
Listing 2.2 shows how the information about the triangles are organized in an STL file, where n is the triangle normal and v1, v2 and v3 are the vertices creating the triangle.

2.4.3 Slicing Software

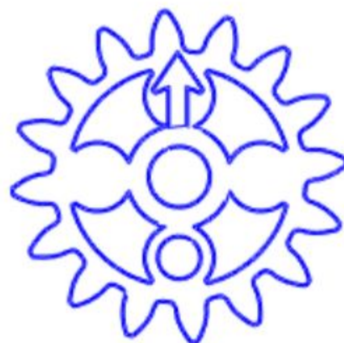
The slicing pipeline consists of a sequence of triangle mesh analyzing algorithms in order to generate the tool path for the 3D printer[34]. This includes defining the contour for each layer, generating the infill based on chosen infill settings and generating the toolpath for the 3D printer. The following two steps are the most important steps in a simple slicing algorithm:

1. **Find and store triangles:** As we know, the mesh of the model is stored in an STL file containing a list of all the triangles and its connecting vertices. The first step in the slicing algorithm is to loop through the STL file, finding all the triangles that builds up the mesh model and store them in a list.

2. **Construct contours:** For each layer of the model one needs to find the contours for the respective layers. A polygonal description of the layer is required in order to accurately construct each layer of the object. The polygons are then filled with material to make it stronger, also called *infill*. This step is important for the generation of the coordinates for the machine, since the 3D printing process consists of printing the contours for each layer in a 2 dimensional plane till the last layer, the top layer, is finished.



(a) 3D model of gear before slicing



(b) Contours for one layer of a gear model after slicing

Figure 2.5: A 3D model of a gear before and after slicing.

Figure 2.5 above shows a 3D model of a simple gear with 16 teeth before slicing (a) and the contours from one layer (b) after slicing. One can see that the contours in (b) matches the 3D model in (a) with the same contours and the same number of teeth. In this particular example, by repeating step 2 above, the contours will look exactly the same in almost every layer. At the top the slicing algorithm will only find the contours for the letter "T" to the right, number "16" to the left and the circle in the middle of the model. After these steps are done the slicing algorithm will generate the tool path for the 3D printer, which tells the 3D printer where to move and how much material to extrude while moving, for each layer. Another setting that is decided during slicing is the *layer height*, which is how far the printer should move in the vertical axis (Z-axis) when moving to the next layer. The lower the layer height is set to, the more layers the printer has to print to finish the object.

PrusaSlicer

The slicer used in this project to generate G-code for the testing objects is the *PrusaSlicer* [52]. PrusaSlicer contains all the necessary profiles and settings needed to carry out this project and is therefore the preferred slicing tool. Other slicing tools can be used as well, but some minor changes in the code for our prototype may be necessary for it to be compatible with it's functionality. The most important feature in PrusaSlicer for the G-Code to be compatible with our prototype and our simulation framework, which will be covered later in chapter 4, is the *Verbose G-Code* option, which are found under the *Print Settings* tab under *Output Options*. By enabling this feature the slicing algorithm will generate a commented G-Code file, with a descriptive text at each line explaining the exact purpose of each command. This can be used to extract key movements used in our simulator and we can choose to exclude a number of different irrelevant coordinates while simulating. When trying to simulate a 3D print by directly rendering the coordinates from a G-Code file, a lot of noise is generated as a large

number of the generated coordinates are just simple movement without any material extrusion. By generating a commented file one can identify all the noisy movement patterns and decide whether it is appropriate to render the movements for the respective commands or not.

```
G1 X124.436 Y156.057 E0.05859 ; infill
G1 X124.320 Y156.198 E0.00603 ; infill
G1 X124.223 Y156.274 E0.00408 ; infill
G1 X124.235 Y156.418 E0.00478 ; infill
G1 X125.646 Y157.829 E0.06595 ; infill

G1 X124.443 Y158.875 E-0.25010 ; wipe and retract
G1 X124.391 Y158.261 E-0.14231 ; wipe and retract
G1 X124.790 Y158.660 E-0.13035 ; wipe and retract

G1 X122.826 Y149.761 E0.07809 ; perimeter
G1 X123.031 Y149.761 E0.00644 ; perimeter
G1 X123.318 Y149.731 E0.00905 ; perimeter
G1 X123.627 Y149.631 E0.01018 ; perimeter
```

Listing 2.3: Commented G-Code file using the Verbose GCode setting in PrusaSlicer.

Listing 2.3 shows some example G-Code generated using the Verbose G-Code setting in PrusaSlicer. The lines that are tagged with *infill* and *perimeter* corresponds to the commands executed in the process of creating infill patterns and perimeters for the model. For the *wipe and retract* commands the printer receives some X and Y coordinates while extruding in the negative direction (E-0.25010). This means that the extruder retracts, or pulls in, the filament rather than pushing it out while moving to the next location. This is to make sure that no molten material accidentally flows out of the system when moving to another position. When simulating G-Code, the wipe and retract commands are also rendered, causing a lot of noise and an obscure scene. With verbose G-Code one can easily detect such noise, remove it and focus on the relevant movement commands.

2.5 Technologies Used

A variety of different libraries and frameworks are used in the development of the proof of concept prototype in this project. As the main purpose of this thesis is to explore the possibilities of using computer vision to detect defects in 3D printing, a library containing image processing and image analyzing functionality is essential for this task. As parts of the planned prototype requires simulation of G-Code executions a graphics framework is also necessary to render the simulation outcome. This section will give a brief overview of the technologies used and the reasoning behind the particular technology choices.

2.5.1 Computer Vision

The field of *Computer Vision* (CV), or *Computational Image Analysis*, is an interdisciplinary field that aims at making computers able to gain a more high-level understanding of images and video frames. CV is usually a very challenging task as it focuses on different problems like image segmentation, object tracking in a video stream, feature extraction and motion tracking [56]. It is usually a demanding task in itself to create applications that performs some of these tasks in both an efficient and a fully automated way. In many cases one need to combine multiple tasks in order for a computer to make sense of an image or a video frame. For instance, a computer vision application made to detect a moving animal needs a combination of object detection to detect the animal itself and motion tracking to track whether the animal is moving or not.

In CV, a set of different *Image Processing* and *Image Analysis* algorithms are applied to images, or frames in a video stream, to solve some of these computer vision tasks. Image Processing is, like the name suggests, processing of an image. An image processing function usually takes an image as input and outputs a new image after applying a set of different image manipulation algorithms like transformation, smoothing and color changes on

the input image. Image processing is an important step in many computer vision applications as it helps eliminate unwanted elements from an image like background noise, which enables us to focus on the more important parts in the image. In our algorithms for error detection it is crucial to be able to remove all noise in the images so that only the 3D printed model is left in the frame.

OpenCV

There exists a large amount of libraries offering functionality for different image processing algorithms and techniques. In this context, Compute Unified Device Architecture (CUDA), OpenCV, MatLab and TensorFlow are some important libraries. OpenCV is the chosen library for this project as it is the most popular library for computer vision applications and it has prebuilt packages for the Python programming language. Since there exist a Python implementation for this library it makes it easy to run it in different environments and platforms like Windows and Linux. Using *pip*, the standard package management system for python packages, the OpenCV build for python will be downloaded and installed automatically.

Listing 2.4 below shows an example using the Python build of OpenCV to read two images into memory and do a simple weighted image addition of the two images. After the weighted addition has been done the resulting image are then stored in the *addedImage* variable on line 6. On line 7 the resulting image are stored in a file called *addedImage.png*.

```
1 import cv2
2
3 img1= cv2.imread("img1.png")
4 img2 = cv2.imread("img2.png")
5
6 addedImage = cv2.addWeighted(img1,0.2,img2,0.8,0)
7 cv2.imwrite("addedImage.png", addedImage)
8
9 cv2.imshow('Added image', addedImage)
```

Listing 2.4: Simple addition of two images in OpenCV

The two input images, represented as two arrays of pixel values, in the addWeighted function are inserted into the following equation[41]:

$$addedImage = \alpha * img1 + \beta * img2 + \gamma$$

where $\alpha = 0.2$, $\beta = 0.8$ and $\gamma = 0$ in our example.

Figure 2.6: The first input for the addWeighted function.

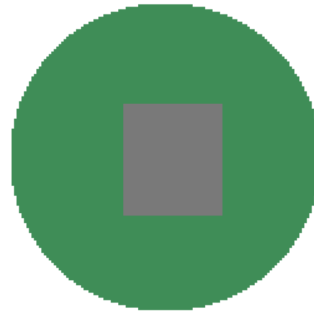


Figure 2.7: Second input for addWeighted function.

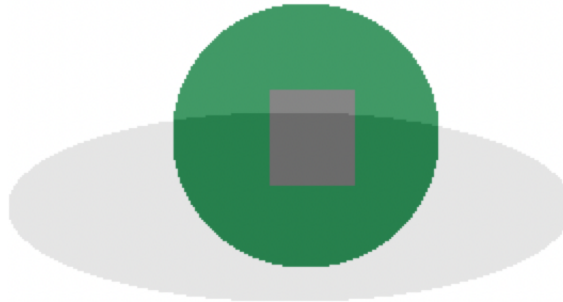


Figure 2.8: Result after doing a weighted addition operation on two images using OpenCV.

The last figure, figure 2.8, shows the final result after doing the weighted addition between the two images. This example shows that one can easily read images and store them in variables and perform different image processing algorithms on them. The *imread* function that reads an image into memory

returns a three dimensional numpy[8] array. The first two dimensions corresponds to the width and height whilst the last dimension corresponds to the number of *channels* for the respective pixels. For RGB pixels the last dimension will be three, one for each RGB color value. For instance, an image of 256 pixels by 128 pixels containing RGB colors will be stored in an array of the following format: (256, 128, 3).

The advantage of Grayscale Images

Many computer vision applications utilizing common image processing algorithms are using grayscale images rather then color images[16]. Therefore, arrays containing the color images needs to be converted to grayscale images, thus we are eliminating one dimension of the array. For the example above, the array will consist of an array of 256 by 128 pixel values after the grayscale convertion. Each value in the array describes the "light" for each pixel ranging from 0 (black) to 255 (white), and not the three values for RGB pixels. Therefore, only one value for each pixel is needed.

A good example of applications that uses grayscale images for image analysis are *Self Driving Cars*. Self driving cars got computer vision algorithms implemented into their core systems that takes grayscale images as input. For instance, it is crucial that self driving cars are able to detect and find their road lanes. *Lane Line Detection* algorithms has therefore been implemented for that purpose. These algorithms uses edge detection techniques [9] and *Hough Transformation* [39] to find and detect straight road lines in an image. When detecting edges in an image the color information for each pixel are redundant information and makes the task hard. However, we still want to keep the contrasts of all the elements in the image after removing the colors. By converting the image to a grayscale image all the contrasts are preserved and the three RGB values are redused to only a single value per pixel ranging from 0 to 255.

2.5.2 Three.js - A JavaScript 3D Library

Three.js is a cross-platform JavaScript library for creating 3D scenes directly in a web browser by rendering the 3D scene in a HTML canvas element [13]. Three.js is built on top of the Web Graphics Library (WebGL) [42] which is based on the Open Graphics Library (OpenGL) standards [42] allowing GPU accellerated 3D rendering. The benefit of using Three.js over WebGL is that Three.js has abstract implementations of many low level operation required in WebGL, making it easier to create 3D graphics directly in the web browser.

Three.js by default comes with a number of different tools for applying matrix operations like translating or rotating objects in a 3D space. It is easy to create objects with different shapes and material properties by using Three.js's pre-defined models. This includes objects like spheres, lines, custom geometries to be defined by a set of points and planes. One of the most important tools in Three.js in regards of our project is the camera system provided in Thre.js. The camera in our simulation environment must be possible to customize in terms of its angle and position relative to the sliced 3D model that is being simulated.

Chapter 3

Problem Analysis

In the two previous chapters, a brief introduction to the 3D printing world and a problem description were presented as well as some necessary background knowledge on how to translate a 3D model into tool paths and commands for the 3D printer. As mentioned in section 1.2, 3D printers may be working 24 hours a day, in which it is hard for humans to consistently monitor the 3D printer. During this time, a lot of different unexpected errors can occur, errors that might even destroy the 3D printer. As the aim of this project is to get an idea of how computer vision can be utilized to automatically detect these errors, it is therefore important to analyze the various errors that can occur and discuss to what extend these defects can be detected using only a camera and computer vision algorithms. In this chapter we will explain the most common errors that occurs in the world of 3D printing, discuss why these errors takes place and how these can be detected. At the end of this chapter, we will also discuss whether an approach using deep learning or a traditional computer vision approach would be better to solve the different classification tasks.

3.1 Why Computer Vision?

Today, many printers are equipped with a monitoring camera that can stream to a website or to a phone app. This makes it a lot easier to more consistently monitor the printer and to verify that nothing has gone wrong. However, this still requires human interaction and the process of additive manufacturing is not as automated as it potentially can be. As also mentioned in section 1.2, one can always gear the printer up with dozens of sensors in which each sensor has it's own responsibility to detect different types of errors that may occur during the print. This is an expensive approach and it requires a total redesign of the current 3D printers to fit the necessary sensors in correct positions. Another concern about this approach is that the firmware, the programming stored on the control board on the printer, must be rewritten to take account for the sensor data and assign different actions to the printer based on the sensor readings.

A numerous of common errors and defects in additive manufacturing, especially in 3D printing, often occurs in the middle of the manufacturing process [15], and most of the defects are detected by the human eye. When the defects are detected it is most likely too late to fix the damage since it is difficult to consistently monitor the process by eye, and thus it is hard to detect the errors in time. Since most of the errors are detected by the human eye, and most 3D printers are equipped with a camera, is it then possible to use this camera as a replacement for the human eye? Is it possible to design computer vision algorithms to automatically detect the same defects as the human eye can? For the human eye it is easy to identify if a manufactured object has defects or not by just comparing it with the modeled object. On the other side, it is a very hard for a computer to understand the context of an image. A computer can only see the RBG (Red, Blue, Green) value of a single pixel at a time[53], which makes computer vision a challenging topic in itself. Another phenomenon that might occur during 3D printing is something called *microholes* (see figure 3.1), which are tiny holes that mostly appears in wall structures on the printed model.

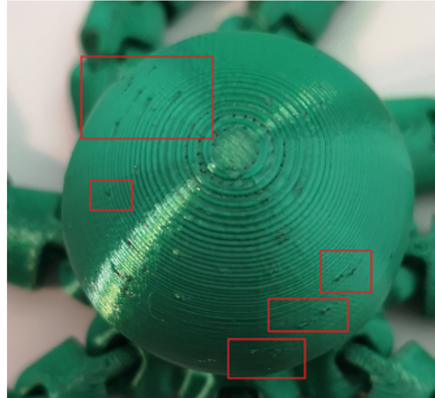


Figure 3.1: Tiny microholes might occur as a result of chosen print settings, bad filament or a partially clogged nozzle

This is an example of a deviation in the print that is not considered an error or defect, but it might be hard for a computer to distinguish between these types of deviations and other defects.

3.2 Problems in FDM 3D Printing

Rapid prototyping technologies, such as 3D printing, became very popular very quickly the additive manufacturing, design and architecture world has evolved considerably since the first printer was built. However, the wave of this popular technology brought with it many challenges that makes it less consistent and more time-consuming for anything other than just prototyping. The main problems of the 3D printing technology are divided into two categories - *Technical Problems* and *Controversies*. The technical issues are defined as deviations, or differs, between the printed object and the original digital model that was designed. Controversies, on the other side, are issues that includes using 3D printing for illegal purposes like fabrication of weapons and drugs [19]. However, we will not be focusing on the controversies issues in this thesis. In this project, we focus on minimising the risk of wasting time by trying to detect the technical problems as early as possible. This way, we increase the chance of not having to start the entire printing

process from scratch and start at the first layer. According to the work done in *The Trends and Challenges of 3D Printing* [19], 3D printing technology will continue to evolve in the future and due to its promising benefits many stakeholders are adopting to this technology. This will result in a gap of knowledge, and as technical issues occurs, the stakeholders might not have the on-site knowledge required to fix it. Therefore, it is important to develop a framework that can adapt to most of the 3D printers to detect these kind of issues before it is too late.

3.3 Technical Problems in FDM 3D Printing

All the errors that takes place in rapid prototyping can be classified into three grades of severity: Catastrophic failure, complete failure and partial failure [24]. Failures that causes damage to the 3D printer are placed in the *catastrophic* category. *Complete* failures are errors that causes so much damage to the object that it is not longer possible to fix the print object.*Partial* printing failures are usually only cosmetic defects on the print object and it can be repaired after the print has finished. Figure 3.1 from section 3.1 is an example of a partial failure, where the failure isn't actually affecting the functionality of the printed object. We will cover the most common complete and catastrophic errors as these has the most significant effect on the efficiency. Table 3.1 gives an overview of the errors[23] that we will explore in more details later in this chapter.

Error	Description
Clogged Nozzle	The material flow gets blocked and there is no material coming out from the nozzle.
Object not sticking	The object detaches from the print bed and starts moving. The melted material may in worst case stick to the heated nozzle and cause a fire.
Extruder Blobs	The filament wraps around the extruder and grows larger as the printer continues to extrude filament. Can potentially cause severe damage to the printer if it's not detected early.
Warping	The 3D printed object starts to deform when printing larger objects.
Over/Under extrusion	Too much or too little material is flowing through the extruder.

Table 3.1: Overview of common 3D printing failures

3.3.1 Clogged Nozzle

During a 3D printer's lifetime it will melt many kilograms of material. Some larger 3D printers might even melt thousands of kilograms of filament. The filament is inserted into the *extruder* and gets melted as it travels through the heated part of the extruder called *hotend* [20]. Finally, at the end of the extruder the melted filament comes out from a tiny hole in the last component of the extruder, the *nozzle*. The nozzle has a diameter ranging from only 0.2mm to 0.5mm, which makes it particularly vulnerable to a Clog. A clog occurs when the filament moving through the extruder gets blocked and the material gets stuck somewhere in the system. The severity of a blocked system depends on how early it was detected during the printing process and in which part of the system that is clogged. The extruder is usually the most advanced part of a FDM 3D printing system, and one wants to prevent any damage that can potentially damage the extruder.

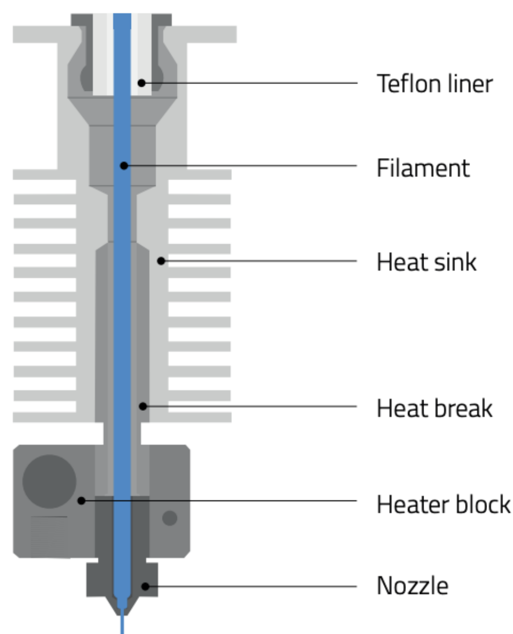


Figure 3.2: An extruder is a complex component consisting of multiple parts. Source [20].

As shown in figure 3.2 an extruder is a complicated part made up of multiple

components. The extruder in figure 3.2 is a typical extruder designed for 3D printing where filament are pushed into the nozzle through a teflon tube and out of the nozzle at at the other end. The following parts are typically used to build an extruder for 3D printing:

Teflon liner: The material, or filament, that is used for printing are usually coiled around a spool that stands next to the printer while printing. The material has a radius of only 1.75mm which makes it necessary to lead it through a teflon tube in order to safely transfer the material to the 3D printer. If the material is attached directly to the extruder, and not through a teflon tube, it may break during a print.

Heat sink: The heat sink is responsible for transferring heat away from hot components in order to cool down the cold side of the extruder. This prevents the filament from melting uncontrolled until it hits the heater block.

Heat break: Where cold meets hot. The heat break is the last component on the cold side of the extruder the material flows through before it starts melting. It connects the heat break to the heater block.

Heater block: The heater block is where the part that melts the filament. It contains a thermistor and a heater cartridge. A heater cartridge is the heating element that heats the hot-end to around 210°C

Nozzle: The nozzle is the part where the molten filament comes out. The nozzle size is usually around 0.4mm, which is the diameter of the opening where the filament comes out. The filament that comes in is usually 1.75mm in diameter, whereas the melted filament tapers down to the nozzle size of only 0.4mm.

As we mentioned earlier the nozzle is significantly narrower than the rest of the extruder components which makes it vulnerable for dusts and other particles that might find its way through the system. A clogged nozzle is a common problem in FDM 3D printing and it can potentially cause severe damage to the components in the extruder. If the material flow gets interrupted in the heated part of the extruder where there are molten material,

the motors will still continue to press material through the system. As the hot end gets overfilled by molten material, the molten material will eventually find its way up through the heat break and into the heat sink before it cools down. Now, as a result of the clogged nozzle, we also have a completely blocked heat sink and heat break caused by the molten material that has been cooled down and stiffened in those areas. Usually, if a nozzle is clogged, it can be unclogged by heating it up and insert a tiny needle through the extruder to try and force the material through. However, if the heat sink and the heat break is blocked as well as the nozzle, it is much harder to fix the clogged nozzle. If one tries to heat up the system, the heat sink and the upper part of the heat break won't reach a high enough temperature to melt the material again. Thus, unless the extruder motor is turned off as soon as the clog appears, the extruder may have to be replaced. It is therefore important to detect a clogged nozzle as soon as it appears.

Many 3D printers have optical sensors installed to monitor both the movement of the filament through the system and the presence of filament. If the sensors detect that the material is missing or not moving when it should move, the print will pause. Even if these sensors are installed, they are usually installed in the upper part of the extruder, namely the cold side. In the case where the molten material starts moving upwards through the heat sink, as mentioned in the last paragraph, the material movement won't stop until the system is completely blocked. The sensors might not detect the clog before it has damaged the extruder, but it is an effective way of saving the print if a clog appears. If the clog is repairable, it is most likely possible to continue the print afterwards with only minor defects. Therefore, if we are able to identify a clog using a camera it can extend the lifetime of a 3D printer significantly when used in combination with already pre-installed optical sensors.

3.3.2 Warping and Deformation

Warping is one of the most common errors that causes deformation and failures in 3D printing and it is directly related to the properties of the material being used as filament [3]. Warping occurs when the parts that are being printed begins to curl in the middle of the process and the object gets deformed. It is typically a corner of the printed object that detaches from the bed and it begins to bend the bottom of the print away from the print bed. This results in a gap between the print bed and the warped corner. As the part of the objects that starts warping are usually printed at the lower layers earlier during the print, it is very likely that the parts are irreparable even if the process is interrupted.

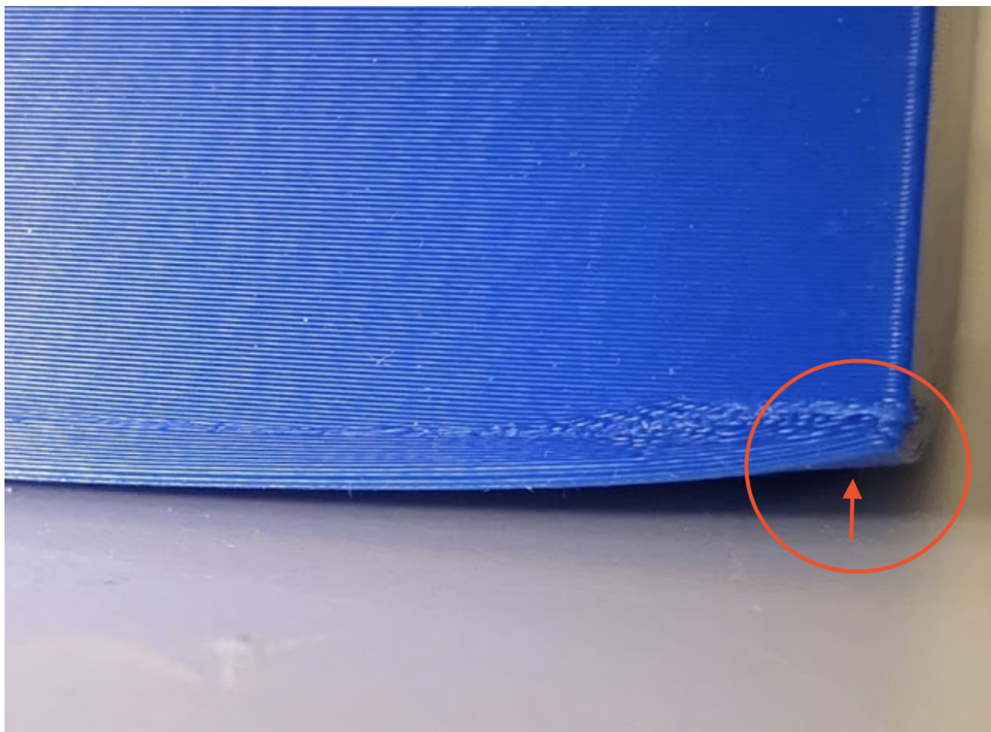


Figure 3.3: Shrinking in the material causes warping

Figure 3.3 above shows a warping that occurred on a larger print. Depending on the size of the model the severity of the deformation might vary. In this case, the print is large enough that the warping does not affect the rest of

the object. The printer is therefore able to finish the process without any further deformations.

The deformation, if severe enough, can cause parts of the object to detach from the print bed and the entire print will fail. This type of failure is relatively unpredictable compared to other potential failures, as it typically occurs later during the print even though the first couple of layers successfully adhered to the print bed [23].

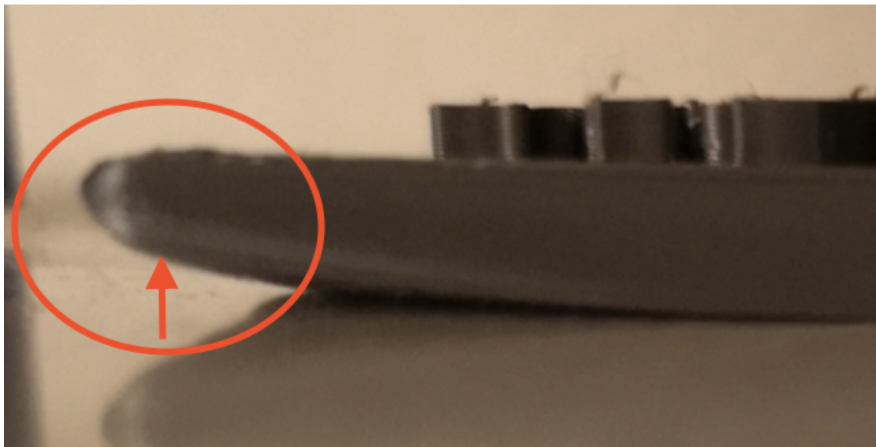


Figure 3.4: Warping on a smaller model causes the entire model to bend

Figure 3.4 above shows a smaller object exposed to deformation as a consequence of warping. If the model is too small, like in this example, the warping causes the entire model to bend upwards. The surface at the warped location is now closer to the nozzle than the rest of the model, and the heated nozzle will touch the surface and cause it to melt in an attempt to print at the warped surface. As we mentioned, it is unlikely that the warping is repairable by interrupting the print when detected, and in most cases the printer can finish the task without further deformations. However, if warping occurs, the current state of the warping should be assessed manually in case the nozzle will touch the surface if the printing process continues. This can prevent any potential damage to the printer caused by warping. If the nozzle touches the surface while printing as a result of warping a clog in the extruder may occur as the material flow gets blocked by the surface. Even though it is unlikely that a warped model can be repaired, or if the 3D printer manages to finish

the process, one might still find it worth to develop an algorithm to detect these deviations due to the potential danger for the printer.

What Causes Warping?

One of the disadvantages of using FDM 3D printing technology is that material starts to behave different as it heats up and cools down again. In a layer-by-layer process like 3D printing, the objects are built by placing layers of molten material on top of each other. When the molten material cools down and solidifies, the properties of the material intervene and it begins to shrink [62]. For instance, if one decide to print using Acrylonitrile Butadiene Styrene (ABS), which is a common thermoplastic material used in 3D printing, it may shrink by almost 1.5% as it cools. If the material shrinks with almost 1.5%, then larger prints may shrink by several millimeters. As the material shrinks the adhesion to the build plate are tested, but unfortunately the adhesion isn't strong enough in many cases. The adhesion strength for a particular build plate will decrease over time as dusts and particles get stuck on the plate.

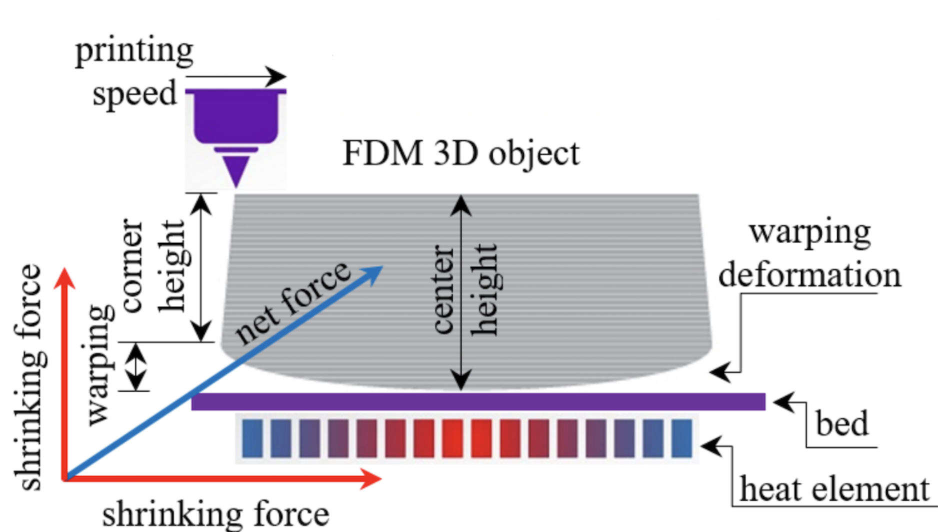


Figure 3.5: Forces applied on the print object as it cools down. Source [3]

Figure 3.5 shows the forces acting on the object as the material is cooling

down causing the warping deformations. As the material contracts towards the center of the object it pulls the corners of the objects with it creating a net force towards the top center of the object at both sides of the object [3]. This is a problem in FDM 3D printing that is hard to deal with and it is unpredictable as it most often occurs on larger objects with sharper corners.

Prevention of Warping

Today, many printers have been equipped with a heated build plate. Before the print starts the print bed gets heated to around 60°C. The heated build plate drastically reduces the shrinkage rate of the material by slowing down the cooling process by a large amount. However, warping still occurs when the adhesion to the build plate isn't strong enough to keep the material from detaching. It is therefore hard to completely prevent the warping from happening during a 3D print.

As far as our knowledge goes, there aren't many publications assessing warping of objects with different materials in 3D printing using systematic methods. Therefore, most of the succeeded prints comes from experience by testing different techniques with different settings. An experiment in [3] shows that there is a relation between the combination of print speed and print temperature and warping. The printing speed is the speed at which the printer moves while extruding material. The printing speed is measured in Millimeters Per Second (mm/s). However, the best combination of temperature and print speed weren't enough to reduce the amount of warping in an efficient way. At this time, it seems that the most effective way of dealing with this problem is to reduce the chances of the machine getting damaged by detecting warping and interrupting the print process as warping takes place. Detecting the problem will not solve the warping problem itself, but it will prevent the printer from continuing on an already deformed object.

3.3.3 Print detachment

Print detachment are identified by horizontal movement in the object as the 3D printer is adding layers and is often a result of bad adhesion to the build plate. The adhesion is the 3D printed material's ability to stick to the print bed during printing. Some typical consequences of horizontal object displacements during printing is *layer shifting* and a phenomenon called *Spaghetti Print*. Layer shifting is, as the name implies, a displacement of one or more layers. If one move a model to the left when it is being printed the next layer will then be printed with an offset to the right relative to the model. If the model is displaced too much there might not be any surface for the machine to build on, and the printer starts extruding material on nothing but air.

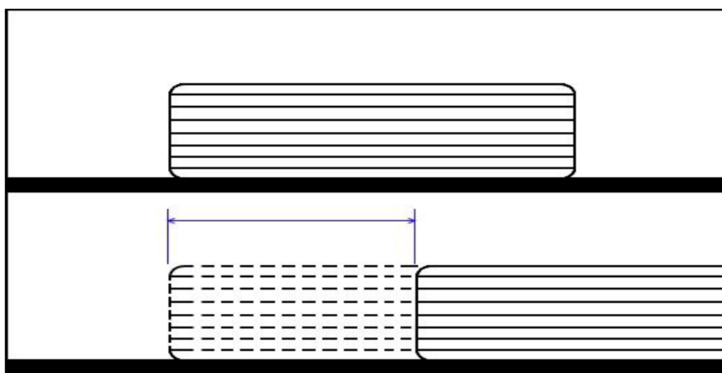


Figure 3.6: Object displacement during printing. Source [6].

Figure 3.6 illustrates an object detachment in two sketches. The object at the top represents the objects to be printed and is attached to the print bed. The sketch at the bottom shows a displacement of the object to the right, where the original position is marked with dashed lines. An attempt to detect such displacement were conducted in [6] by implementing different image analysis algorithms for *Blob Detection*. Blob Detection algorithms aims at detecting regions of interests in images that differs from other segments of an image, like different color brightness compared to surrounding pixels. However, as stated in [60], blob detection algorithms liable to fail and catches a lot of

noise making it hard to find the contours of the blob. In our project we will utilize a similar algorithm by applying more robust algorithms and methods rather than applying blob detection methods.

Figure 3.7 shows an example of how spaghetti print can occur when printing in the air with no surface for the material to attach to. In this case the printer managed to attach the last layers to the surface, but very often the production of "spaghetti production" won't stop until the printer is stopped resulting in a large amount of unorganized material on the print bed. The strings may also attach to the moving nozzle that can cause additional issues for the 3D printer like clogging the nozzle or damaging wires. Using traditional computer vision algorithms and methods it is hard to directly detect these strings and classify them as spaghetti print. A project called *The Spaghetti Detective* [25], which is a plugin made for 3D printers that supports *Octoprint*, uses deep learning algorithms to detect these features. However, many printable models contains features that are mistaken as spaghetti by the spaghetti detective. An example of this is the *gyroid infill* (See figure 2.3c), where the gyroid infill pattern can be detected as strings.

Figure 3.7: Printing on overhang without support results in spaghetti print

Chapter 4

Design and Implementation

4.1 The Prototype

The target of this project is to investigate the possibilities of using computer vision to automatically detect errors and making the 3D printing process more robust. Some of the methods mentioned in 1.5 can detect some specific errors using machine learning algorithms. However, a lot more training data for the machine learning algorithm is needed in order to make the detection more consistent. Some of the machine learning algorithms might also detect different infill patterns as a defects. In this chapter we present our proposed method for process monitoring and automatic fault detection in 3D printing using traditional computer vision algorithms and image processing algorithms. The prototype is divided into three different parts each with its own responsible to accomplish their own tasks. The first two tasks are responsible for monitoring and detecting errors in the first layer of a print and the nozzle during the rest of the print. The last task is to use the already mounted cameras to collect labeled datasets for future development using more advanced approaches by combining machine learning, deep learning and computer vision algorithms in future applications.

The fault detection task is further divided into multiple sub tasks:

- **Snapshot:** Activate the top camera to take a snapshot of the first layer when it is printed
- **Simulation:** Simulate the first layer to create a digital model for the verification process
- **Detect contours:** Find the contours of both the simulated first layer and the snapshot of the printed layer
- **Verify the first layer:** Apply the same image processing algorithms on both images to detect deviations
- **Activate the nozzle analyzer:** When the first layer is verified, the camera in front of the 3D printer is capturing frames and tracking the nozzle as the printing process progresses

In the following sections of this chapter we give a more detailed overview of the prototype. First, we present a simplified overview of the system and its main components showing how the components are related to each other. Next, we present all the algorithms and methods used for the first layer verification and the nozzle analyzer.

4.2 System Overview

Our prototype is implemented using a Client-Server Architecture [54]. The Client-Server model is an architecture made up of one or multiple clients and a server. The clients send requests to the server and the server responds accordingly based on which communication protocol is used. This architecture also provides a mechanism called *Inter Process Communication* (IPC). IPC allows different processes to communicate with each other using either *Shared Memory* or *Message Passing*.

In our prototype we have one server and one client, where the server handles all the image processing, the verification algorithms and the simulation. The client is responsible for controlling the camera and is communicating directly with the printer. When the client has recorded a frame, the frame is sent to the server which responds with a simple boolean value telling the client whether it has detected a fault or not. If a fault is detected in multiple frames, the client parses the G-Code through the serial communication between the client and the 3D printer to pause the printing process.

The communication between the server and the client is fairly simple. We are using the Python Socket API to create network sockets. Both the client and the server are implemented as two sockets. The client socket is directly connecting to the server sockets IP address and the same port the socket runs on. Listing 4.1 shows a basic example of a client-server communication from a clients perspective. The client is trying to connect to the address "127.0.0.1" and the port "65432", which is the same port number and IP address as the server runs on. If the server running on the same address and port, we are able to connect to the server and send and receive data. The address used in this example is the IP address used for *localhost* and not a real IP address, so the processes can only communicate when they are running on the same network.

```
1 HOST = '127.0.0.1'
2 PORT = 65432
3 with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
4     s.connect((HOST, PORT))
5     s.sendall(b'Testmessage')
6     data = s.recv(1024)
7 print('Data Received:', repr(data))
```

Listing 4.1: A socket in python acting as a client

Listing 4.2 shows a simple implementation of a server socket. The server runs on the same address as the client so that the client are able to connect to the server using the localhost address. When the client is connected the server can receive and respond to the messages sent from the client. Both the process and the server must run in different terminals, or processes or threads.

```
1 HOST = '127.0.0.1'
2 PORT = 65432
3 with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
4     s.bind((HOST, PORT))
5     s.listen()
6     print("Waiting for connection..")
7     connection, address = s.accept()
8     with connection:
9         print('Connection received from: ', address)
10        while True:
11            data = connection.recv(1024)
12            if data:
13                connection.sendall(data)
```

Listing 4.2: A socket in python acting as a server

The client in our system is designed to run on a microcomputer called *Raspberry Pi*, which is a small single-board computer that are able to run operating systems like Linux. Since the Raspberry Pi can read data from multiple different low level sensors like temperature sensors, humidity sensors and cameras, it works excellent as a client in our prototype as we want to capture video frames through a camera. The Raspberry Pi is also able to handle standard USB connections, making it possible to establish a connection directly to the 3D printer through serial communication. Figure 4.1 presents a simple graphical overview of all the main components involved in this project.

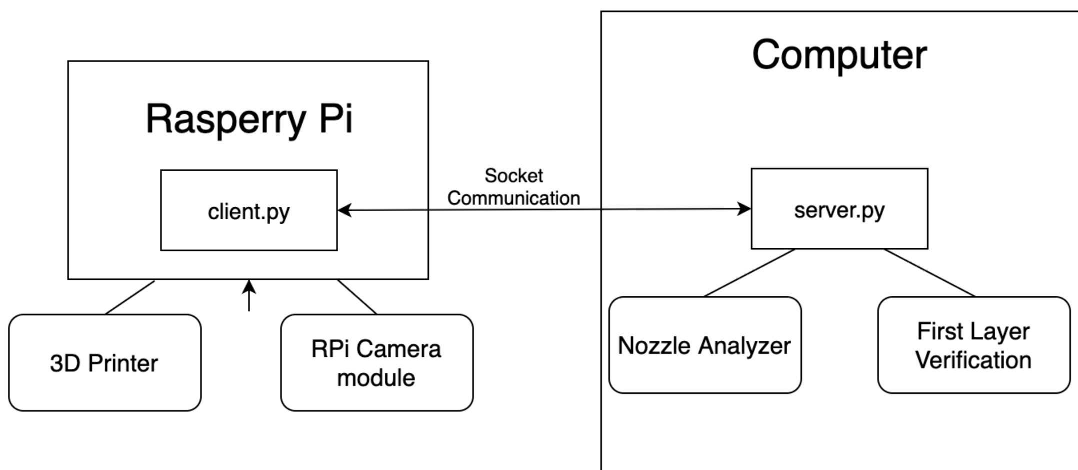


Figure 4.1: A graphical overview of the main components in our Client-Server Architecture

In figure 4.1 we can see all the components in our project and which components that are communicating with each other. The Client receives video frames from the RPi Camera module, which is a camera module designed to be used together with a Raspberry Pi. The client is also communicating with the 3D printer, which is able to receive G-Code commands through serial communication. In serial communication messages are sent one byte at a time. When a frame is recorded it is sent to the server socket, which runs on a normal desktop computer. A normal computer has better hardware and thus better computing power making the image processing much faster. Therefore, it is necessary to implement the Client-Server architecture, as the

Raspberry Pi won't be able to efficiently apply the image processing algorithms by itself. When a frame is received by the server, depending on the state of the printing process, it is consumed by either the nozzle analyzer or the first layer verification (FLV) application. If we are on the first layer doing first layer verification of the frame, it is consumed by the FLV application. When the first layer is verified the rest of the frames will be received from the front camera that tracks the nozzle.

As we have mentioned earlier it can be hard to find a large dataset containing training data for different types of defects occured while 3D printing. A large dataset is necessary in order to be able to train a deep neural network or a convolutional neural network to accurately classify a 3D print as a failure or as a success. Therefore, all the frames that are received by the server are stored locally and will be labeled and stored in a database in the cloud when the printing process has finished. The system we implemented to label the data are isolated from the Client-Server architecture and will be discussed later in section 4.6.

4.3 First Layer Verification Process

4.3.1 What are we looking for?

It is vital to get the first layer as perfect as possible for the print to succeed [43] as this builds the foundation for the rest of the 3D print. There are mainly two issues related to the first layer: Under- and Over-extrusion and the first layer not sticking to the print bed, where the latter one is one of the most common problems that occurs in 3D printing [29]. Detection and autonomous correction of under- and over-extrusion is done in [26]. However, if the printer starts to under- or over-extrude it is a sign that there is a mechanical failure, or maybe a minor clog in the nozzle, that is causing it. Changing the the extrusion multiplier, which are done in [26], can cause even more issues if the settings are tuned too much. For instance, if the extrusion multiplier are too high it will eventually cause a clog in the nozzle and there will be no material flow. Deep learning are used to detect such issues due to the difficulties in designing statistical models for traditional computer vision algorithms that are able to reliably detect these patterns.

In our FLV application we focus on detecting the perimeters of the first layer to verify that all parts of the first layer are printed. The goal is to implement computer vision algorithms that can detect errors that occurs during the first layers of the printing process. This is particularly useful when printing parts that has multiple layers that are not connected to each other, like models that are twisted around each other and locket together as joints to make the objects flexible. Figure 4.2 shows a 3D printed model built from multiple parts that are printed together as joints.

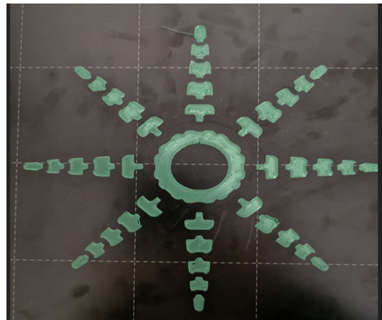
Figure 4.2: A 3D printed octopus with flexible legs

When 3D printing parts that are connected through such joints, the first layer usually contains multiple parts, where each part are connected by a joint. One can then say that we have multiple first layers that might fail. If one of these layers fail the entire print will also most likely fail due to one part moving around and disturbing the rest of the print. In figure 4.3 (c) and (d) we intentionally reduced the adhesion to the bed raising the nozzle by 0.2 millimeters. We can see that 4 of the contours have detached and been dragged around as the nozzle travelled between its coordinates. One of the contours has also melted into the nozzle which can potentially cause severe damage to the printer if not detected early.

Figure 4.3 (a) shows how a typical 3d printed first layer for a multi-joint model looks like. The first layer consists of multiple contours that are not connected to each other and the chances of a detachment increases with each contour. Figure 4.3 (b) shows the final result. In our prototype we want to trigger a first layer verification at this point before the 3D printer gets a signal to continue the process if the first layer gets approved by the computer vision algorithms. If the algorithms detects that something is off by a certain threshold, we want to send a signal to pause the print.

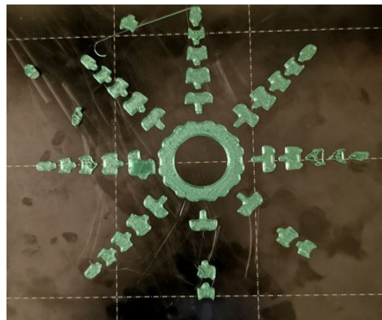
It is a hard problem to detect whether the first layer is loose or not since

the object doesn't necessarily move. In that case it isn't a critical problem and the process will most likely finish without any problems. Therefore, the algorithm in our prototype is designed to look for deviations in the contours of the first layer. For instance, if the algorithm can only find 6 contours when the model is supposed to have 7 or more contours, then we have detected a deviation from the original model. To achieve this we need an image of what a successful first layer should look like in order to assess whether it has deviations or not. A basic simulator that takes the generated G-Code file, the same file that is sent to the 3D printer, as input was built to simulate a 3D print of the first layer. The simulator loops through the commands sequentially and builds up the same object layer by layer based on the coordinates generated from the slicing pipeline. The object is rendered in a 3D scene using a web browser 3D graphics framework called *ThreeJS*. ThreeJS is a framework built on top of the Open Graphics Library[42, 13], a cross platform API used for interacting with a Graphics Processing Unit (GPU). Thus, by using the simulator, we have a template to compare the actual print to and it is possible to assess the first layer using different image analysis techniques and algorithms.

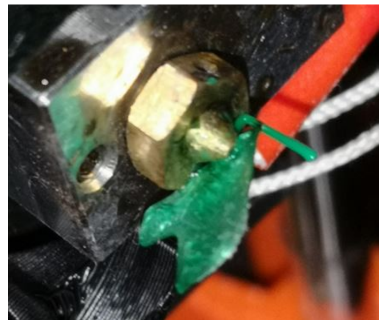


(a) Multiple contours that are not connected

(b) The finished model with flexible joints



(c) Some of the contours detached during the middle of a print



(d) One of the first layer contours melted and got stuck into the nozzle

Figure 4.3: The first layer showing all the joints and the finished result

4.3.2 A modified 3D printing pipeline

The normal 3D printing pipeline involves creating a design using CAD software, store the model as a STL file, slice the STL file and send the resulting G-Code file from the slicing pipeline directly to the printer. With our prototype a rework of the entire 3D printing pipeline is necessary to make our system communicate properly with the 3D printer.

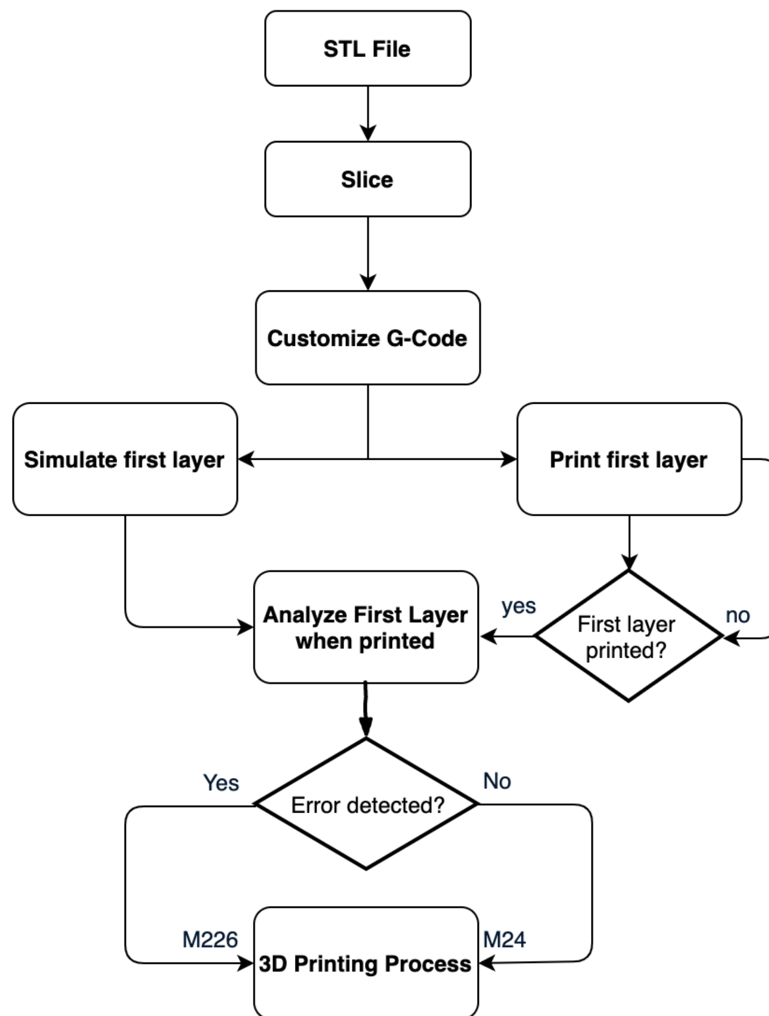


Figure 4.4: A flow chart for the first layer verification prototype

Figure 4.4 shows a simple overview of the re-designed 3D printing pipeline using our prototype to analyze the first layer before the printing process

continues. Compared to the original pipeline shown in figure 2.1 that is normally used in additive manufacturing[17] discussed in section 2.2 we have added an extra layer of complexity to make this process as autonomous as possible. The first two steps in the pipeline are the same as for the original pipeline, where the STL file designed using a CAD application are sliced into layers which in turn are converted into movement commands for the 3D printer. However, this time the G-Codes are not sent directly from the slicing pipeline to the 3D printer. The file containing the sequence of G-Code commands is the only source of information about the state of the current state of the 3D printer. Instead of parsing all the commands into the machine's memory and let it execute the program in its own environment, we have added a step to customize the G-Code so that it is possible to keep track of which layer the printer is currently working on at any given time. The customized G-Code is used by our simulator to extract necessary data from the model to render the chosen parts of the model. However, at this time we are only interested in knowing when the first layer is completed so that it can be verified. The adaptation of the G code will therefore only be with respect to the first layer of the print. This includes information about which commands belong to the first layer, how many contours exist in the first layer and infill. Sometimes the first layer consists of multiple layers to make a more solid foundation, and this is also necessary information in order to make an appropriate simulation of the first layer.

Once the customization is completed, the customized file containing G-Code commands will be sent to both the printer and the simulator. By sharing the modified file with the printer it is possible to obtain knowledge about the state of the current process; thus a pause command can be triggered when the first layer has been printed to analyze it using a camera. When a pause command has been executed we know that the first layer is printed and we are ready to move to the next step, the first layer verification process.

A frame from the 3D scene that is rendering the first layer simulation are stored as a temporary image in the file system and later used as input to the first layer analyzing algorithm. From the camera attached to the top of the

printer a snapshot of the current layer is also taken, which in theory should be equal to the simulated first layer as it is simulated using the exact same G-Code as the 3D printer. In order to get this as accurate as possible, a camera that is well configured and calibrated for use in computer vision[47] is necessary for it to work. The snapshot from the simulated 3D scene and the physically printed layer are then given as the two inputs to our deviation detection algorithm.

Algorithm 1 Detect deviations in first layer

1: $cv2 \leftarrow import(opencv)$
2: $np \leftarrow import(numpy)$
3:
4: **function** LAYER_HAS_DEVIATION (snapShot, firstLayer, thresh, pVal)
5:
6: $s, f \leftarrow scale(snapShot, firstLayer, 700)$
7: $sGray, fGray \leftarrow grayScale(s, f)$
8:
9: $t1, t2 \leftarrow threshold(sGray, fGray, thresh, pVal, cv2.BINARY_INV)$
10: $sBlur, fBlur \leftarrow filter2D(t1, t2, -1, np.ones(5, 5))/25$
11:
12: $sContour, fContour \leftarrow findContours(sBlur, fBlur)$
13:
14: **for all** s in sContour **do**
15: **if** hasSameContour(s, fContour) **then**
16: **return** *True*
17: **end if**
18: **end for**
19: **return** *False*
20: **end function**

Algorithm 1 above describes the key steps in the first layer analyzing part of the pipeline for finding and comparing the contours between the simulated virtual first layer and the actual footage of the printed first layer. Our

function takes in the two images as well as the threshold and pixel value[40]. First, the two input images are scaled to make some of the image processing algorithms to work more efficiently on the images. The scale function returns the scaled version of *snapShot* and *firstLayer* respectively and store them in the abbreviated variables s and f. A grayscale conversion is applied on line 7 to make the pixel values range from only 0 (black color) to 255 (white color) instead of using the standard Red, Green and Blue (RGB) pixel values. As objects in a grayscale image usually differs substantially in gray levels from the background of the image, it is now easier to apply image processing algorithms for image segmentation like thresholding algorithms[50].

4.3.3 Thresholding the First Layer

In section 2.5.1 we briefly introduced *Computer Vision* as the ability for computers to gain a high-level knowledge of the content contained in images. A 3D printer is just following a sequence of commands while extruding material, *blindly* trusting the movement pattern it has received through the G-code. We still want the 3D printer to rely on the G-Code commands received from the slicing pipeline, but we want to make it *less* blind.

Before the first layer can be verified we need to implement algorithms such that a 3D printer is able to *see* the first layer of the print. The ideal result would be a binary image, an image containing pixels that can only have *one* of exactly two colors, where one of the colors represents the printed first layer. Algorithms to obtain segmentation in a snapshot of the first layer are implemented using a combination of different techniques. The main methods used for the segmentation are *Thresholding* and *Masking*.

Thresholding

Thresholding is an effective technique in image processing to perform image segmentation based on the pixel values. Segmentation allows us to remove everything that we aren't interested in and isolating objects that we want to

focus on. This is typically a good way to remove background noise and shadows from the image that may interfere with our image analysis applications. The result after applying thresholding is typically a binary image, where the pixel values above the threshold are converted to black whilst the pixel values beneath the threshold are converted to white. As only tiny deviations from the original model can be crucial enough to make the entire print to result in a defect model, we are dependent on an accurate object isolation in our image analysis algorithms.

To find a good threshold value that is good enough to isolate the first layer from the rest of the image can be a challenging task, especially when there are a lot of reflection from the model making it appear more white on some parts of the object. Depending on the light, the lower part might for instance be a lot brighter than the upper part of the same model. In figure 4.3 (a) we can see that shadows on the lower part make the first layer appear darker whilst the upper part reflects some bright light making it look more white. This effect can cause the model to obtain the same gray levels as the white dotted squares on the print bed making it harder to isolate the printed layers from the rest of the image. Even if we manage to find a good threshold for a particular case, the lightning will with high probability be slightly different during the next print. With different lightning a different threshold is needed, otherwise the same image processing and analysis algorithms will produce different results each time, making our fault detection very inconsistent.

In figure 4.5 we are applying thresholding algorithms to a snapshot of the first layer of the same octopus model used earlier in this thesis. This model is a great model to experiment with as it contains multiple geometries that are not connected to each other. By being able to detect and isolate all the parts of such a complex model we create a solid baseline for succeeding on less complex models in the future as well.

Our goal is to eliminate every pixel from the image that is not a part of the printed first layer. We can see in figure 4.5 that applying thresholding algorithms to the image of the first layer results in binary images containing only two pixel values. As mentioned, some parts of the 3d printed first

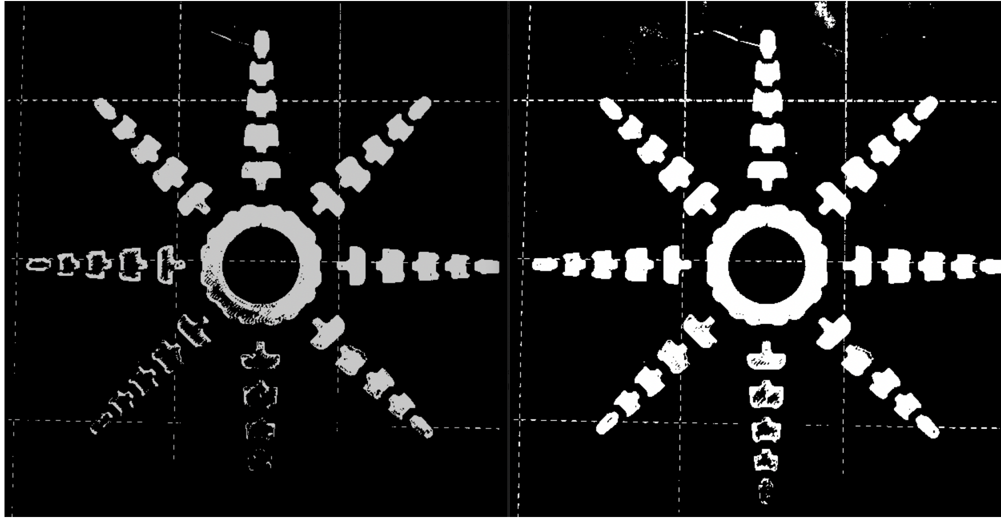


Figure 4.5: Thresholding algorithms applied using two different thresholding values. The image to the right has a lower threshold value allowing darker pixel values.

layer appears brighter than other parts due to the lightning, which may create an interference in the thresholding algorithm. In the image to the left the darker part that were exposed to shadow did not reach the threshold and got excluded from the binary image. By reducing the threshold value, meaning that darker pixel values gets included, the shadow exposed parts of the first layer appears in the image to the right. However, with more forgiving threshold values comes more noise as more pixels gets included. We can see that the dotted lines from the print bed are much sharper as well as some other reflections from the build plate. The image situated to the right got a threshold reduction of only 15, from a value of 120 to 105. Compared to the maximum value of 255, which is the largest pixel value in a grayscale image, this is a small reduction that only makes 6% out of the total pixel range from 0 to 255.

By using traditional computer vision algorithms to detect deviations from the printed first layer and our digitally created first layer it is hard to accurately perform image analysis on images containing a high level of noise. In figure 4.5 every single pixel above the threshold, the white spots in the image, will

be treated as a part of the first layer. This means that the reflection from the print bed, the dotted lines and cosmetic damage to the print bed that passes the threshold check will be interpreted as a deviation from the model. However, what if we accept a certain threshold of errors that are allowed to occur before we trigger an error? By allowing a certain percentage of deviations to occur we take account for the redundant pixels. The problem with this method is to find a threshold that is generating the same results even with different lightning. With more lightning, more reflections will occur creating more deviations.

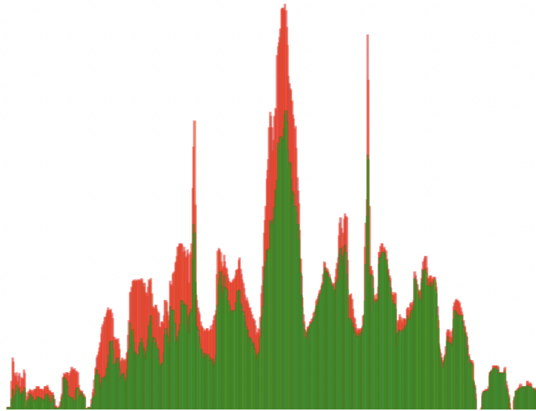


Figure 4.6: Histogram of the binary images in figure 4.5. The red graph reflects the image with lowest threshold value.

We created a histogram analysis of the two images with different threshold values from figure 4.5. By creating a histogram of both images we get an accurate description of the pixel distribution that we can compare. Figure 4.6 shows the histogram created for the two binary images in the previous example. Despite the small change in threshold value the results are significant. The green graph represents the image with the largest threshold value of 120 whilst the red one represents the image with the lowest threshold value of 105. We know by default that our example model got a symmetric shape which should be reflected by the pixel distribution in our histogram. An interesting case to note is that for the image with largest threshold value has most of it's bright pixels shifted towards the right, while the second image

experiences the opposite effect. We can also see that the green graph maintains a better symmetry than the red graph does, as the red one generally is shifting more towards the left side. This shows that even a small change in the projected lighting can change the perceived symmetry of the geometry completely. The human eye have the ability to easily detect whether a part of such a binary image is noise or not by comparing with the digitally created model. However, for a computer that only is able to see one pixel at a time through a camera, it is hard to determine determine whether the current pixel is a part of the object or not.

	Threshold: 120	**Threshold: 105**
White pixels	36851	48363
Total pixels	415829	415829
Percentage of white pixels	8.86%	11.63%

Table 4.1: Pixel distribution of the binary images in figure 4.5.

Table 4.1 above shows a detailed overview of the counted pixels taking place in both binary images. The number of white pixels increases drastically by more than 10.000 pixels when reducing the threshold value by 15. This means that 2.8% of the total numbers of pixels got converted from dark pixels below the threshold to brighter pixels above the threshold value. This is equal to a 31.2% increase of white pixels. If we, for instance, decide to set the threshold for our deviation detection algorithm to accept and ignore all detected deviations below 5-10%, the noise generated by the change in lighting conditions can quickly reach this threshold generating a false deviation detection. By increasing the deviation detection limit we reduce the chances of the reflection generated noise to reach the new threshold. However, this also increases the chances of classifying real defects as a false detection. For this method to be successful, optimal lighting conditions and maximum contrast between the material being printed and the build plate are indispensable.

Even with constant lighting, some objects might be printed on some of the spots of the print bed that reflects most light which in turn will contribute

to a small change in lighting conditions. Another problem by utilizing a light sensitive method is that our method will only work on specific types of material, removing the ability to print with light absorbing materials. Artificial Neural Networks (ANN) and Convolutional Neural Networks (CNN) are commonly used for detecting objects in images for classification purposes [38] and object segmentation by displaying the object and its position by *masking* it. These ANNs and CNNs are pre-trained models to detect known shapes and patterns in images. In 3D printing, the shapes and patterns are not known, and therefore it is hard to train a network to mask the objects. We use a similar *masking* technique used in most CNNs for image segmentation by looking for simple features that takes place in most of the first layer prints, namely the *colors* of the print material.

Masking the First Layer

Creating object segmentation algorithms that are based purely on the contrasts of the image, which are done on many computer vision applications, comes with some challenges in detecting the first layer in 3D printing. The main challenge is that the level of contrasts are very inconsistent and the contrast of the surroundings can very quickly reach an approximated contrast at the same level as the object we want to isolate. As a solution to this we implemented another layer of image pre-processing utilizing a masking technique that is based on the RGB colors of the image, before we threshold the image. This allows us to limit the area of the image to a *Region of Interest* (ROI). The pixels included in the resulting image after performing the masking operation will have a color similar to the material of the first layer being printed. This means that in most scenarios, unless the material has a similar color to its surroundings, the resulting image will contain only pixels that are related directly the first layer, called *Superpixel*.

As mentioned earlier, the masking is used to display objects detected by the segmentation algorithms. Usually CNNs or ANNs are used for this purpose as the majority of images contains different colors and shapes that are hard to

create consistent statistical models and algorithms for. In our case, we only need our segmentation to isolate two segments of the image - the first layer and the background. The algorithm only need to find the segment containing the first layer, as everything else in the image will be grouped together as the background, or the *second segment* in the image. Our advantage is that the first layer will consist of a single color and not multiple different colors. Our prototype is therefore built on the assumption that the first layer is built using only one type of material with a single color.

Algorithm 2 Algorithm for creating the mask used for segmentation

$cv2 \leftarrow import(opencv)$
$np \leftarrow import(numpy)$

function CREATEMASK (image, lowerCRange, upperCRange)
  $hsv \leftarrow rgbToHsv(image)$
  $mask \leftarrow inRange(hsv, lowCRange, upCRange)$
  $isMask \leftarrow newList()$
  for all m in mask **do**
    if $m \geq 1$ **then**
      $isMask.add(True)$
    else
      $isMask.add(False)$
    end if
  end for
  $fLayerPix[isMask] \leftarrow image[isMask]$
  $res \leftarrow bitwise_and(image, fLayerPix, mask)$
  return res
end function

Algorithm 2 shows our algorithm for generating the mask for the first layer. First we import the python modules to be used in our application, *opencv* and numpy[8], as these modules contains all the functionality for handling the lists and the pixel operations. The function takes in an image, which

is a numpy array and two three dimensional vectors describing the HSV (Hue, Saturation, Value) [27] values for the lower HSV and the upper HSV boundaries. The input image is originally in RGB format. The first line of the function converts the RGB image to a HSV image, converting all the pixels to HSV values instead of RGB values. The HSV format is the preferred format in most object detection application as it is a decomposition of an image providing necessary analysis tools for classification and image segmentation [27]. Figure 4.7 shows how the colors changes as the HSV values changes.

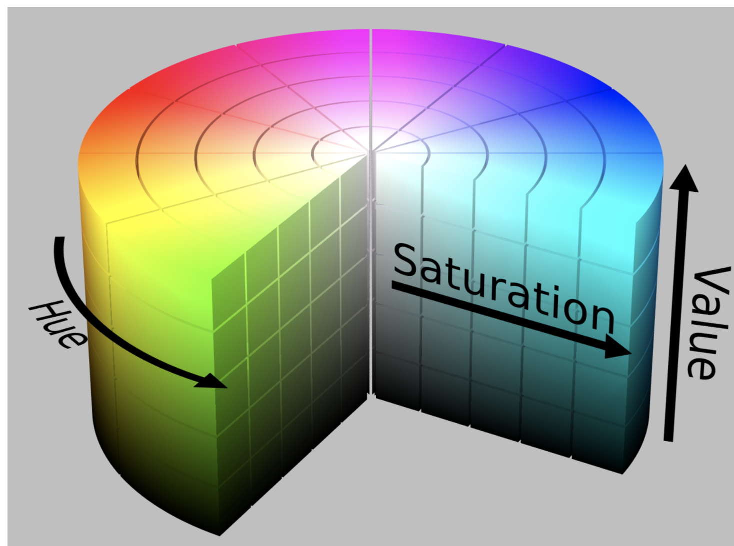


Figure 4.7: HSV Color Space [From: Wikipedia]

Figure 4.7 shows how the colors act as we change the HSV values. The *Hue* value ranges over the entire spectrum of colors, and we call this the *wavelength* of the perceived color as it is the most dominant factor. The *Saturation* value can be changed to manipulate the strength of the color. The last parameter, *Value*, determines the brightness. A low value will give a darker appearance of the pixel.

After converting the image to an array of HSV pixels, we create a new array with the same size, changing all the pixels that has a value in the range between the lower color and the upper color boundaries to white. The pixels outside the boundaries are set to black. As the image only contains black and white pixels, we have a new binary image of the model, but this time we

have removed all the background noise. By utilizing the HSV color model we are able to implement a segmentation algorithm. Figure 4.8 shows the result of the mask after applying algorithm 2 on the image.

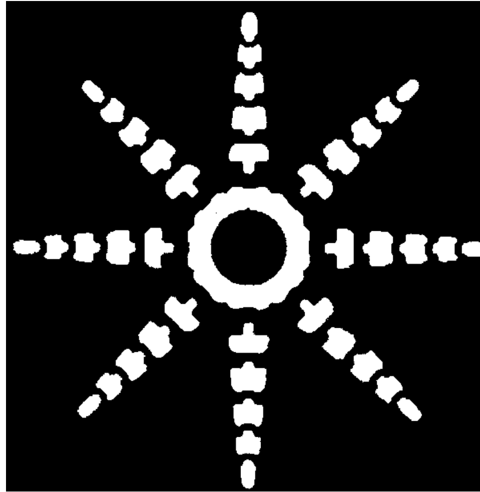
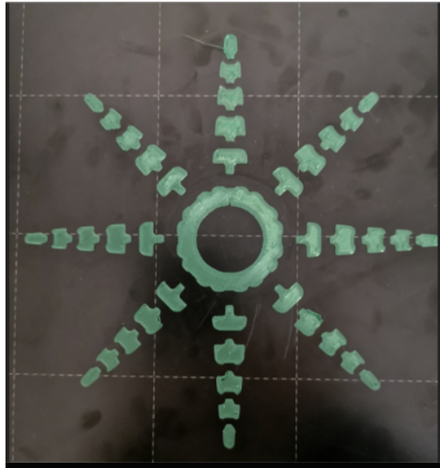


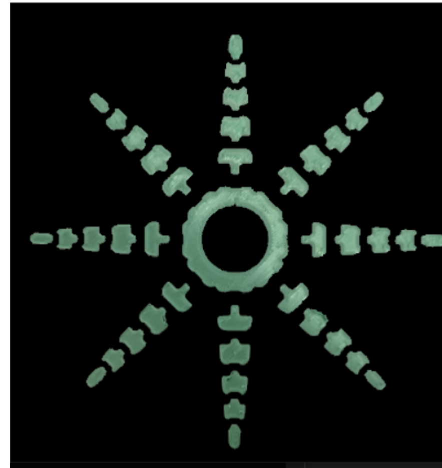
Figure 4.8: The resulting mask after applying algorithm 2 on the image

Now, we have an almost perfect binary image of the first layer. However, we want the image containing the full RGB color space with the noise excluded. Thus, the final step in our algorithm is to render all the pixels from the original image with the corresponding pixel indexes from the *isMask* array that contains the boolean value *True*. The boolean *AND* operation is performed on the image with the binary image applied as a mask, telling the algorithm which pixels to do the boolean operation on. This operation is also called *Bitwise AND operation* as it is performing the boolean operation on each pixel in both images that has a matching indexes. If pixel $(x_i), y_j$ in both images has the same pixel value we keep it, otherwise we set it to 0.

The final result can be seen in figure 4.9. We now have a clear segmentation of the first layer in the full RGB color model. By converting it back using all three color channels we are able to apply better image analysis algorithms directly on the segmented image.



(a) Snapshot of the first layer



(b) Image segmentation algorithms are applied to remove background noise from the snapshot

Figure 4.9: Image segmentation algorithms are applied on a snapshot of the first layer print.

We have successfully implemented a segmentation algorithm to create a threshold for the first layer snapshot and we have been able to isolate a snapshot of a 3D printed first layer. As we can see in the first algorithm we introduced in section 4.3.2, algorithm 1, we have implemented the threshold function on line 9, but we have only applied it to the snapshot and not to the simulated first layer. However, the same algorithms will be applied to both the digitally rendered image and the snapshot. The next step in algorithm 1 is to find the contours of the first layer, which requires accurate pre-processing algorithms like thresholding to isolate the objects we want to analyse. The resulting pre-processed image from algorithm 2 will be the first input for the function that will search for the contours whereas the simulated layer will be the second input.

4.3.4 Contour Approximation

We have done an image segmentation of the first layer snapshot, changing all the background pixels to black leaving us with an image of only extruded material. For the first layer verification process the pixels need to be organized. Now, the image is only a two-dimensional array containing pixel values ordered from the top-left corner to the bottom-right corner. It is hard to do any sort of shape analysis on an array containing only color information about each pixel. A good way to do shape analysis on figures from a two dimensional view is by finding the *contours* of the shape on wants to analyse. If we go back to figure 4.9 we see that this model has multiple potential contours, where each green region that is not connected to another green region may have its own contour.

A contour is simply an approximated curve that joins all the continuous points, typically around a boundary. OpenCV provides an implementation of Douglas-Peucker Algorithm [61] which can be used to find points around a boundary to create contours. The algorithm does an approximation of the points and does not include every single point around the border. This algorithm is highly sensitive to interference from other pixel values, requiring a successful pre-processing of the image before one can approximate the contours. From the image processing done in last subsection we are able to find all the contours for the first layer. Figure 4.10 below shows the result after applying OpenCV's implementation of the contour finding algorithm.

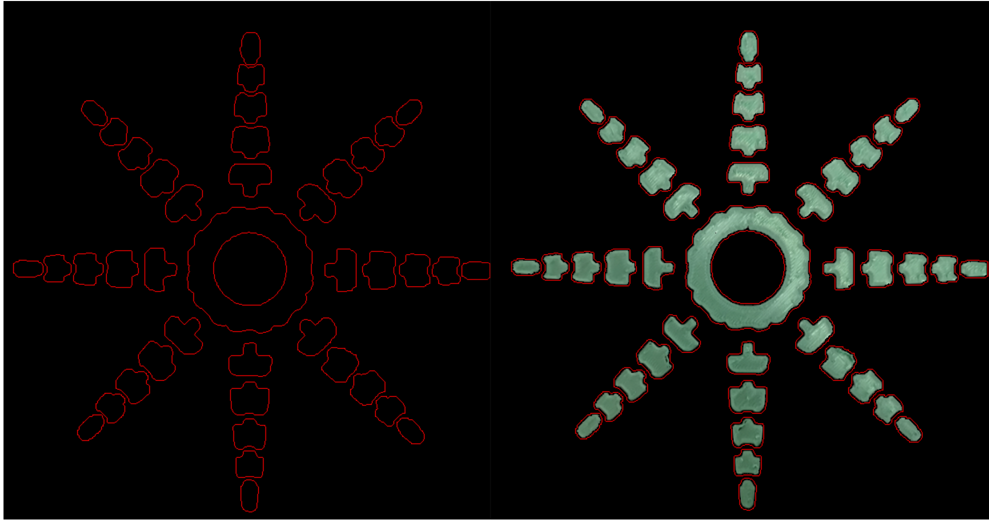


Figure 4.10: Approximation of contours are shown in red

In the image to the left all the contours are drawn on a black background. In the image to the right the contours are drawn in the same image as the algorithm was applied to. We can see that the approximated contours fits well to the 3D printed first layer shapes, but there are some slight deviations in some of the shapes. Each contour is an array of points that lies on the contour curve, and all the contours are stored in a list. By being able to find the contours of the first layer enables us to apply more advanced image analysis techniques on the first layer. We are for instance able to count the total number of contours stored in the and thereby counting the number of objects that has been printed.

To be able to count the number of printed objects is in itself is a big step towards moving from a list of pixel values to a higher level of knowledge about an image. In the next subsections we have implemented a simulation framework that takes the sliced 3D model as input and simulate the 3D printing process layer by layer. The simulation and the result from figure 4.10 above will be compared to verify the first layer.

4.4 Fault Detection Simulation Framework

In the last section we managed to find all the contours and organize them in a list. Each index of the list contains information about one contour. However, this information is useless for our purpose without any knowledge about the intended shape. We want to verify whether the first layer has deviations to the intended model. The existing tools for analysing a sliced 3D model before printing is not suitable for our purpose as they generate a lot of noise and are hard to customize. Therefore, we created a simulation framework that is highly customizable which analyses the generated G-Code directly to find information about the model.

Our Simulation Framework is implemented using NodeJS, which is a JavaScript runtime environment, and the high level graphics library *ThreeJS* for rendering the sliced model in a 3D scene. A simple server hosting our web interface, the ThreeJS application, is built using a web application framework for NodeJS called *ExpressJS*. ExpressJS is used to process and respond to HTTP requests. Therefore, our simulation framework can be accessed directly in a web browser by navigating to the server address which will render the ThreeJS scene in a web browser.

The framework is split into two main parts: The 3D Scene for rendering the model and a back-end which accesses and analyses the G-Code before it is sent to the front-end application for rendering. As we mentioned earlier, the 3D printing pipeline using first layer verification is slightly modified, and requires the 3D printer to pause after first layer is printed. The back-end service for our simulation framework automatically adds two G-Code commands to the G-Code file before it is sent to the 3D printer. The commands are added right after the G-Code that defines the end of first layer. The added G-Code commands are *G28* and *M226*. Since the camera is mounted on top of the printer angled directly down to the build, the camera view will get blocked by the moving print head. Sending G28 command to the printer before taking a snapshot of the first layer will solve this as it tells the printer

to move the axis to their home position. When the axis are placed in their respective home positions the M226 command is triggered telling the printer to temporarily stop the process. The process continues when the printer receives the M24 command which are sent after the first layer is verified. It is also important to note that these commands are not supported all 3D printing firmwares. If the commands are not supported one must change the code to use the commands that apply for that particular 3D printer.

The ThreeJS Scene

Figure 4.11 below shows all components and their child components in the ThreeJS scene.

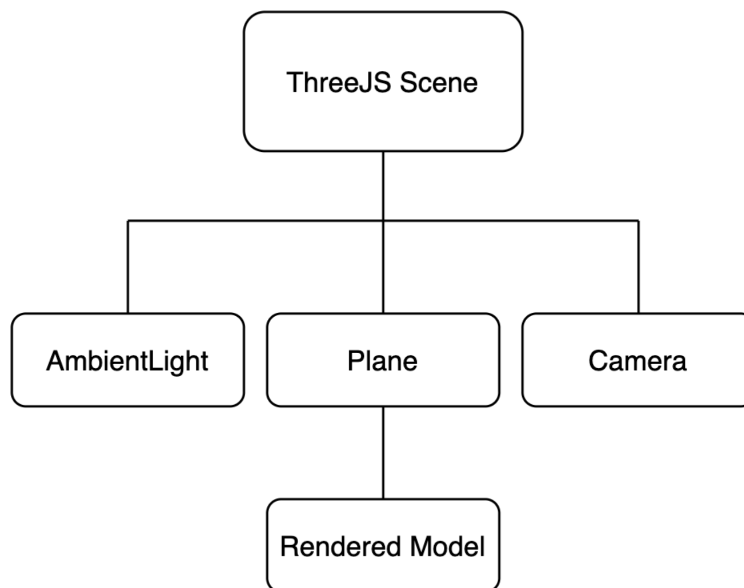


Figure 4.11: Object in the ThreeJS scenegraph

The scene has three child objects: AmbientLight, Plane and Camera. A light source is used to simulate lighting effects in ThreeJS to illuminate objects and make them visible. By using the AmbientLight object all objects in the scene are illuminated equally in the scene, so no noise due to lighting

are generated. A plane geometry is a rectangular shape used to illustrate the build plate of the 3D printer. The Camera is the virtual camera in the 3D scene. The *Rendered Model* is a child of the Plane component, meaning that whenever a translation or a rotation is applied to the plane, the same operation will be applied to the model making it move relative to the plane.

A simple user interface are placed at the top of the ThreeJS application where the user can chose between different options for simulation. For instance, one can set a texture on the plane illustrating a real build plate and apply realistic lighting to evaluate error detection algorithms without printing. Another important option is what information about the model one wants to render in the scene. For instance, in our case we want to analyze the contours of the sliced model and therefore the option for *perimeters* are selected. One can also define which layers to be analyzed. When all options are set the front-end application triggers a HTTP request containing the option parameters to the back-end server. Listing 4.3 below shows the request object sent from front-end. In this example the front-end asks for contours of layer 1-4 in the *multi_joint.gcode* file.

```
1 {
2     "filename": "multi_joint.gcode",
3     "startLayer": 1,
4     "numLayers": 4,
5     "features": ["contours"]
6 }
```

Listing 4.3: Request object sent from the front-end application

Back-End Server

The Back-End receives a request from the front-end triggering a function to analyse and extract the requested information from the G-Code. First, the G-Code file is searched for the layer height. For instance, if the layer height is 0.2mm we know that we are on the first layer if the nozzle height is set to 0.2. The next layer will have a nozzle height 0.4 mm increasing by 0.2mm for each layer. Using this information we can identify each layer from the G-Code file.

However, a G-Code file contains different procedures built up by movement commands for different purposes which creates a lot of noise in a G-Code file. For instance, when a contour is printed the printer do not move directly to the next location. It performs a *wipe and retract* movement, which consists of multiple commands executed to retract the filament from the nozzle and wipe the nozzle against the printed contour to *clean* the nozzle for any melted material that may have attached to the nozzle. A procedure like this can be identified in the G-Code by a *change in feed-rate*, which changes velocity of the axis. Perimeters and infill are printed using a constant feed-rate.

The algorithm strips all coordinates from G-Code commands containing a constant feed-rate until a change in feed-rate is detected. When a change is detected the algorithm continues to the next G-Code commands with the normal feed-rate and repeats the process until maximum layer height has been reached. The extracted coordinates for infill and perimeters are packed into a multi dimensional array where each index contains infill and perimeter information for the respective layer. The array is then sent back to the front-end as a response.

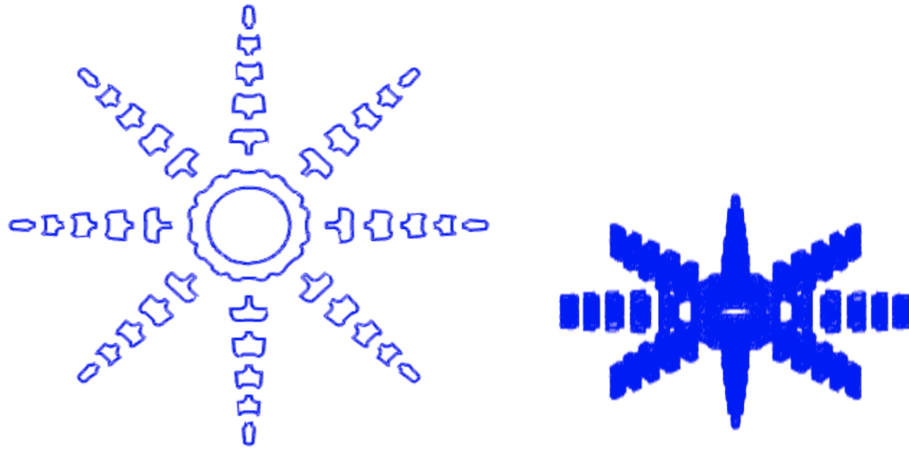
Rendering the Extracted Coordinates

Rendering of the coordinates are done in the front-end application when it receives the response from server. The response will contain information of either contours, infill or both. The rendering process is the same regardless. The rendering process goes as follows for the perimeters:

1. Loop through the list of perimeters
2. Create a basic ThreeJS geometry model containing an empty list
3. For each point in the perimeter push it to the list from previous step.
4. Create a ThreeJS Line Geometry and define its shape using the points from last step
5. Add the Line Geometry to the scene

6. Repeat from step one until all perimeters are processed

7. Render the scene.



(a) Simulated G-Code seen from a top-down view

(b) Simulated G-Code seen from an inclined camera angle

Figure 4.12: The simulated G-Code from two different camera angles

Figure 4.12 shows the result of our simulation framework. In figure 4.12a we can see that the rendered contours from our G-Code simulation matches the contours found in the last section. Our fault detection simulation framework can be used to compare these two images in order to verify the first layer. We can, for instance, count the number of contours and check whether they have the same contours or not. In figure 4.12b we have changed the camera angle slightly so that we can see the layers of the model. In the next section we present a method to detect errors after the first layer has been verified by using another camera view. Changing the camera angle in our simulation framework so that it matches the angle of our monitoring camera can give the monitoring algorithm useful information like how many objects are expeted to be seen from a particular angle.

The next figure, figure 4.13 shows the results after comparing two images of the same model, but one is missing some of the contours.

The image to the left is missing some contours and we detect it as a *polygon error*, as there are some missing polygons in the picture. For models that do

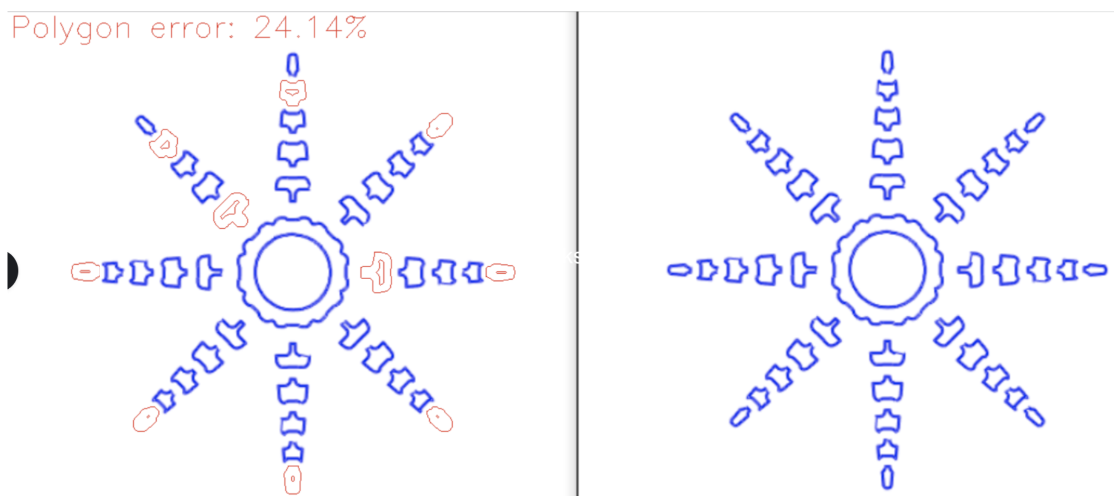


Figure 4.13: Deviation detected between two images

not contain multiple contours in their first layer we have implemented image analysis techniques to calculate the aspect ratio of the bounding box around the entire model. The fact that we detect a discrepancy between height and width in the image may be a sign that an error has occurred in the first layer. This can be caused by the 3D printed layer loosening from the build plate and has been rotated, warping in one of the corners or missing material extrusion.

4.5 Nozzle Analyzer

In section 3.3 we discussed different situations in which technical problems arises during the 3D printing process. A clogged nozzle, warping, deformation of the printed object and object detachment were the most common failures that occurs in the middle of a print. In an earlier subsection, section 4.3.3, we mounted a camera on top of the printer to detect deviations in the first layer. Printing the first layer is usually the hardest part as that's where problems usually occurs. By verifying the first layer, most of the automatic fault detection is therefore already completed. However, we still want to be able to detect if the object that is being printed has detached from the print bed or if there is no filament flow in the extruder. In [6] a similar project using similar techniques was conducted. However, their algorithms focuses on detecting the largest *blob* found in a frame, and the algorithm wont be able to detect any deviations while printing multiple objects.

4.5.1 Camera Position Relative to Y-Axis

In order to set up real-time monitoring of the object that is being printed, the camera angle from the first layer verification process must be changed. For the first layer verification a snapshot were taken from above, which requires the printer to home the X-axis before the snapshot is taken. As the object is being printed, the view of the printing process gets blocked by the print head making it hard, or even impossible, to monitor the process from above. Depending on the how the Y-axis for the 3D printer is mounted, a different setup for this implementation may be required. The printer used in this project has the print bed mounted on its Y-axis, making the object move in the direction of the Y-axis as it gets printed. Therefore, accurately monitoring the process in real-time is a challenging task as the camera may lose focus of the print object as the printer is progressing. The camera is mounted in the front of the 3D printer to make the movement of the Y-axis parallel to the camera projection, making the Y-axis either move towards

and away from the camera, not left and right. Figure 4.14 shows the setup with a Raspberry Pi controlled camera mounted to a tripod in front of the 3D printer.

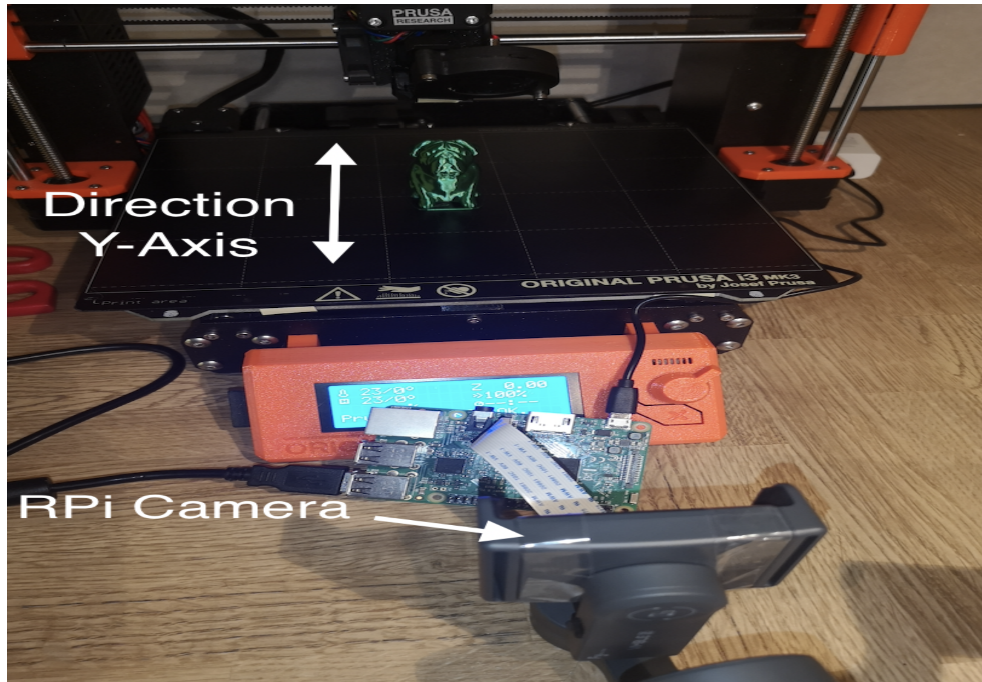


Figure 4.14: 3D Printer setup with a Rasperry Pi controlled camera for monitoring

In figure 4.14 we can see that the mounted camera is alligned in parallel to the direction of the Y-axis. This eliminates the problem in which the object moves left and right on the axis making it hard to automatically analyze the image properly due to the camera projection angle.

Figure 4.15 simulates an example of a setup that we want to avoid in our prototype. A 3D scene with a camera and its perspective projection are simulated using the cross-platform game engine called *Unity3D*. The camera is rendered as a white object in the lower-left corner. In front of the camera are three equally sized objects placed with an equal distance between each other. One can see the projection lines for the camera in which everything inside the lines will be rendered in front of the camera. The frame in the lower-right corner displays the view from the cameras perspective. If we look

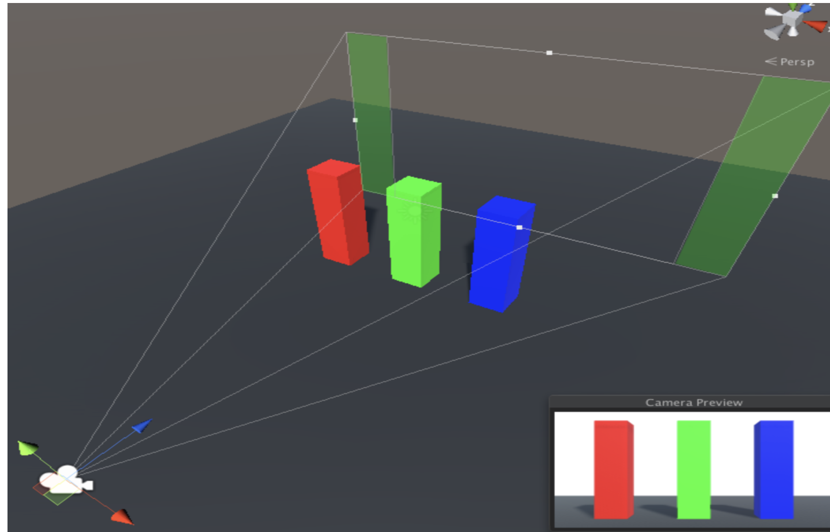


Figure 4.15: Simulating a camera projection in Unity3D

carefully at the camera view in the lower-right corner, one can see a small part of the left side for the blue object and the right side of the red object due to their position relative to the camera. We want to avoid this in our prototype for now, as this will be interpreted as a significant change in geometry by the image analysis algorithms. We have therefore placed our camera directly in front of the object that is being printed so that the projection is in parallel with the Y-axis to avoid this effect.

As we have eliminated the factor in which the object moves sideways enables us to easily detect if the object has lost the adhesion to the print bed. For each frame sent to the server we have to check if the object has changed its center of mass since the last frame that was analysed. If there is a significant change in the center of mass we can confidently conclude that a detachment has occurred, and the server will respond with a false-flag to the client which will pause the process. However, as one challenge gets solved another challenge arise. The distance between the nozzle and the object that is being printed changes slightly from the perspective projection seen in an image. This means that the pixel-distance between the nozzle and the printed material will be lower when the Y-axis moves away from the camera and vice versa. This is a scenario that has to be taken account for in the nozzle tracking algorithm

as a consequence of the chosen camera position.

4.5.2 Region of Interest

In the previous subsection we explained the experimental setup for our prototype discussing the importance of obtaining a good position and angle for the camera. The view from the camera position is shown in figure 4.16. We have marked three reference points in the image, marked as purple circles, which will define the Region of Interest (ROI) of the image. ROI can be defined as smaller samples within a larger dataset that is under consideration. For instance, in a computer vision application we may want to consider a smaller set of pixels in a large array of color pixels. We strategically chose four points in our image to define a rectangular box that defines the boundaries for our ROI. To find our region of interest we apply the same masking algorithms as we did in the last section to find the positions for the reference points. The two bottom reference points will decide the minimum and the maximum value for the X-axis, as well as the maximum value for the Y-axis. The lowest reference point will define the largest Y-value since the top-left corner is, as opposite to most 2D coordinate systems, defined as the origin in most computer graphics libraries, including the OpenCV library used in our project. Following the same procedure, the reference point on the nozzle will be used to set the minimum Y-value. The reference point attached to the nozzle will also be used to track the movement of the nozzle to calculate the distance between the nozzle and model.

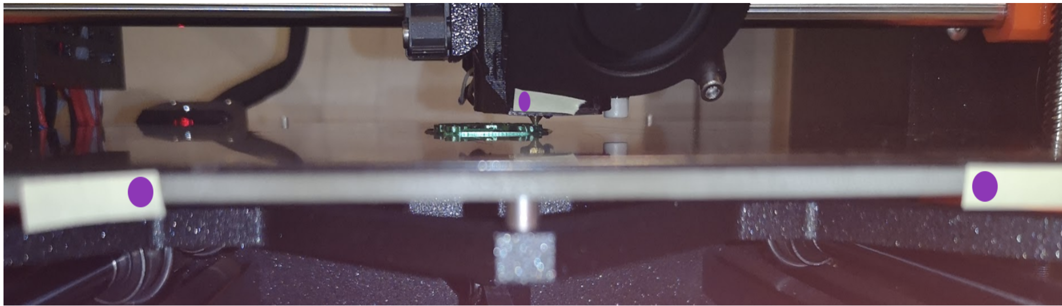


Figure 4.16: Build plate with reference points seen from the front camera setup

Algorithm 3 is applied on the image to calculate the values for x_1, x_2, y_1 and y_2, which are the values for our ROI coordinates.

Algorithm 3 Calculate ROI points for print area

```
1:  function CALCULATE_ROI_POINTS (image, lowerBound, higherBound)
2:
3:      x1 ← image.width
4:      y1 ← image.height
5:      x2 ← 0
6:      y2 ← 0
7:      hsv ← convertToHsv(image)
8:      mask ← inRange(hsv, lowerBound, higherBound)
9:      contours ← findContours(mask)
10:
11:     for all c in contour do
12:         xc, yc ← calculateCentroid(c)
13:         x1 ← min(x1, xc)
14:         y1 ← min(y2, yc)
15:         x2 ← max(x2, xc)
16:         y2 ← max(y2, yx)
17:     end for
18:
19:     return x1, x2, y1, y2
20: end function
```

The algorithm takes an image and an upper and lower limit for the HSV color range used in the segmentation of the reference points. After the segmentation is done and the reference points are isolated in the mask, we are looping through all the detected reference points, calculate the *Centroid* of each reference point and then the minimum and maximum values for the X and Y coordinates are calculated. These coordinates are also called *Extreme Points* [44]. In *Linear Programming* an extreme point is usually called a *vertex* or a *corner point* in a convex set. A *Centroid* is, in mathematics and physics,

defined as the geometric center of a shape in a plane. The extreme points will define the extreme north, east, south and west centroid coordinates for all the contours that are detected in algorithm 3.

Finally, the following points will define our ROI: (x_1, y_1), (x_2, y_1), (x_1, y_2) and (x_2, y_2). The ROI coordinates will be used to *crop* the image, removing all the pixels that are not found to be inside the ROI boundary, eliminating a lot of noise from the image. As the image cropping provides a much smaller area to work on, a large amount of pixels that may interfere with further image processing are now excluded, increasing our chances to detect errors.

Figure 4.17 shows the resulting image after cropping the original image based on the coordinates from algorithm 3. The printed object is marked in a red box whilst the nozzle is marked in a green box. By applying the same segmentation algorithms as we did earlier to isolate the first layer we are now able to isolate both the nozzle and the object on the build plate.

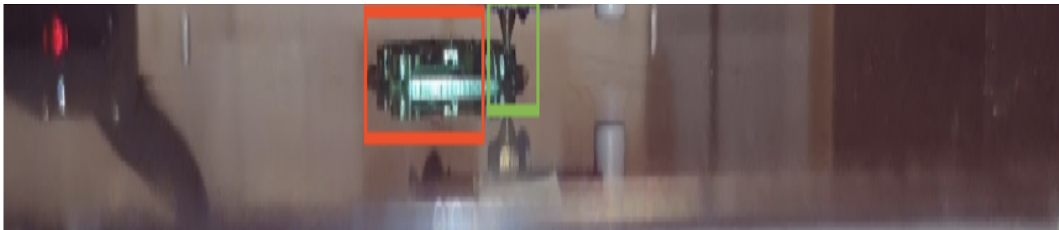


Figure 4.17: Frame from the front camera are cropped to the reference points creating a ROI

With both the nozzle and the 3D printed object isolated from the background, applying image processing and image analysis techniques directly to the nozzle and the model without any interference from the environment is possible.

4.5.3 Detection of Extruded Material

This part of the process may arguably be the most difficult and error-prone step of this module. As seen in the captured frame in figure 4.17 from last subsection, the light that hits the build plate in our experimental setup is bouncing off and creating reflections. The reflections creates a number of pixels with values that are relatively close to the color range we use for image segmentation. We solve this problem by increasing the similarity distance between similar pixels. This will result in an increased contrast between the pixels making the segmentation more consistent and less error-prone to pixel errors.

The cropped RGB image is converted to the *CIELAB Color Space* [35] (LAB), which is often used as a base for color vision. Lab expresses colors as three values: L for the lightness, A for the color from green to red and B from blue to yellow. To increase the contrast for the frame an image processing technique called *Adaptive Histogram Equalization* (AHE) [64] was applied to the L-channel of the LAB-image. However, this algorithm was shown to also amplify the noise, creating almost the same relative distances between many of the pixels that interfere with the image segmentation as the original image. Another variant of AHE called *Contrast Limited Adaptive Histogram Equalization* (CLAHE) [65] prevents this problem as it is designed to limit the amplification.

A 3D printer often prints a different model each time a new printing process is started, making it difficult to find a common feature that consistently occurs in almost every model. If a box of rectangular shape was printed each time it would have been possible to implement a statistical model to detect all the features of a rectangular shape. Since that is not the case, the only feature that we can be sure exists is the color of the filament used to print the model. By increasing the contrast using the CLAHE image processing technique we can try to find a color range in the HSV color space that only applies to the material being printed. However, this limits our image analysis algorithms to only work with materials that has a certain level of contrast compared to

the environment.

Finding the HSV Color Range

Finding the correct color range in the HSV color space that only applies to the objects that we want to detect in our image can be time consuming. Using the OpenCV library it we can create a User Interface (UI) and pass different values directly into our algorithms using UI sliders to change the values. By using the UI we set a minimum and a maximum value for all three HSV values to find the range that fits best to isolate the 3D printed object from the rest of the image. Figure 4.18 below shows the UI used to find the HSV range. The values are applied to the cropped image from figure 4.17. As we can see the only visible part on the image is first couple of layers that has been printed by the printer whilst everything else has been erased. The sliders below shows the minimum and the maximum values for the HSV color space to obtain this segmentation. For this specific material, lighting condition and background the HSV range should be in between (78, 23, 174) and (96, 74, 255). As the values changes the image will be re-rendered after the algorithms has been applied with the new input values.

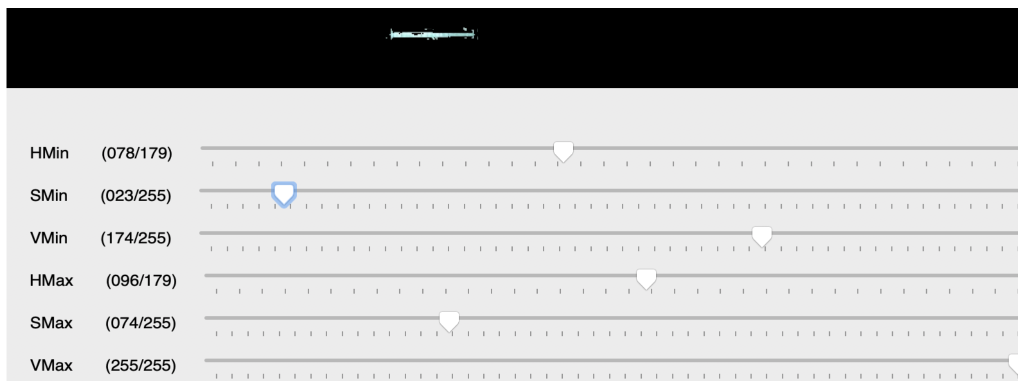


Figure 4.18: UI to find a suitable HSV range for image segmentation

2D Convolution Matrix and Image Smoothing

By looking carefully one can identify groups of pixels that are not connected in the binary image shown in figure 4.18 above. When we try to find the contour of the printed object, the pixels that are not connected to the rest of the object will be perceived as isolated objects by the algorithm. Therefore, multiple contours will be detected. Smoothing the image will solve the problem with multiple contours. This is done by passing a 2D averaging filter to the image where a *Kernel* is convolved with the image. A Kernel is usually a relatively small matrix that is much smaller than the image to be processed. The matrix below shows an example of a 5x5 kernel filled with ones:

$$k = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The kernel is placed on top of a pixel in the image to be smoothed. All the pixels in the kernel now has a respective pixel in the image. The pixels in the image are multiplied by their respective cell in the kernel matrix, added together and then divided by 25 to get the average. Finally, the pixel in the middle of the matrix will be replaced with the new average value. The operation is repeated on every pixel in the image, and the result will be a smoothed image. With a smoothed image one can calculate the contour lines of the detected material. Nonetheless, the matrix presented above will extend the pixels in both directions in the two dimensional plane, which makes it challenging to detect a potential clog of the nozzle. To detect a clog the distance between the 3D printed object and the nozzle is measured. If a clog occurs it will be detected as the distance between the nozzle and the object is increasing each time the nozzle is moving up one layer.

If the image is smoothed in the vertical direction it can be hard to mea-

sure the distance consistently as the segmentation algorithm might segment different pixels each time due to external influences like change in lighting or adjustments of the focus by the camera lens. In our prototype we have therefore applied the 2D convolution on the image using a 1x10 kernel matrix for the averaging process instead of a 5x5 matrix. A 1x10 matrix will only perform the averaging operations in the x-axis, and not the y-axis. As we eliminate the problem with the interference of the distance measurement, there is also a downside with using only a single row for averaging the pixels. If there exist a pixel that does not have any close white pixels as neighbour in the horizontal axis, it will affect the averaging significantly, thereby the smoothing may not be effective enough when calculating the contour. By increasing the size of the row from 5 to 10, the smoothing is more aggressive and may compensate for the loss in vertical averaging reducing the chance of having any *lonely* pixels. The result of the horizontal smoothing is shown in figure 4.19. The top image shows the raw segmentation whilst the bottom image shows the result after smoothing.

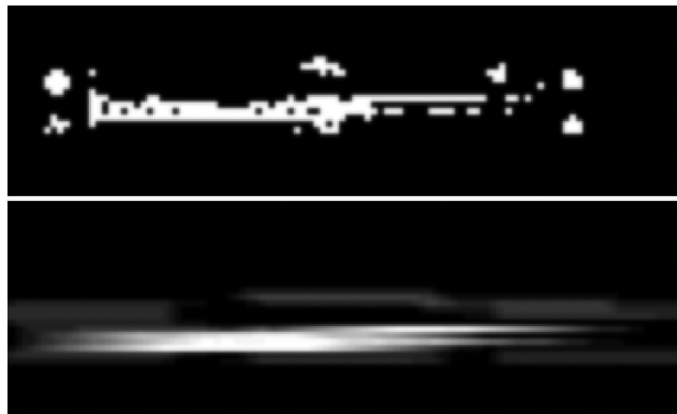
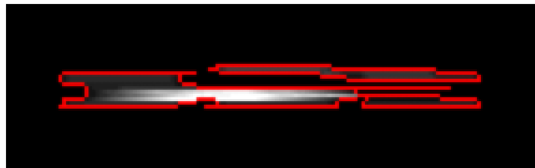


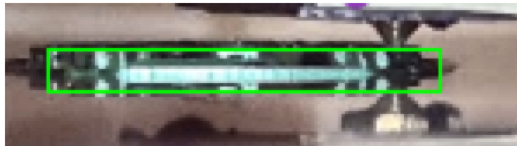
Figure 4.19: Smoothing the binary image for better contour calculation. The bottom image shows the result after smoothing the upper image.

4.5.4 Contour Approximation and Distance Measuring

Using OpenCV's implementation of the Douglas-Peucker Algorithm [61] we can find a contour approximation for the smoothed image. With the contours of the binary image one have access to a series of different features that can be used for error detection. Examples of features that can be calculated are area, centroid and moments of the shape.



(a) Contour approximation for the smoothed binary image



(b) Bounding rectangle around the detected filament

Figure 4.20: Calculating a bounding rectangle around the contour

In figure 4.20 we can see the results after applying the contour approximation algorithm and drawing a bounding rectangle around the contours. The bounding box is used to verify that we have calculated the correct extreme points for the 3D printed object. For each frame of the image we calculate new extreme points for the filament. The points that were calculated in the previous frame are also stored in variables allowing us to compare the current calculated points with the previous calculated points. There are three points in particular that are interesting to compare: The point that are furthest to the left, the center point and the rightmost point. If there is a significant change in the value of the X-axis for these points from frame to frame we can conclude that movement of the model has been detected. This is a sign that during the printing process the object has detached from the build plate and

the Rasberry Pi sends the M226 G-Code command to the printer, telling it to go in pause mode.

To find the relative distance between the nozzle and the 3D printed filament we need to know the position of the nozzle. The nozzle matches too many colors colors with the surrounding environment making it difficult to find a color range to isolate it through image segmentation. When the ROI was selected, a reference point were placed on top of the nozzle enabling us to crop the image where the nozzle is included in the ROI. We will use the already calculated center of the nozzle-reference point to measure the relative distance. We calculate the *Euclidian Distance* between the top point of the bounding rectangle around the filament and the center of the nozzle-reference point. The X-coordinate is the same in both points and is set to the X-coordinate of the reference point.

$$dist = euclideanDistance(X_{nozzle}, Y_{material}, X_{nozzle}, Y_{nozzle})$$

By only changing the Y-coordinate the calculated distance will reflect the relative *height* difference between the nozzle and the filament. As the nozzle's reference point is used to measure the distance and not the nozzle itself, the calculated distance between filament and nozzle won't reflect the *true* distance. As we are only interested in detecting a *change* in the relative distance, and not interested in the distance itself, the algorithm will work by detecting the change in distance between the reference point and the object. However, some calibration is needed since we need to find a threshold for which distances to be accepted.

The threshold is found by applying the algorithm to an image of which we know is a successful print. Figure 4.21 shows the result after applying the algorithm to a frame of a successful print. By calculating the distance between the bounding box and center of the reference point we find that the Euclidian distance is 22 pixels, which is the threshold value in this example. The threshold distance may vary slightly due to changes in pixel values inside the HSV range as a result of reflection and other external interference. With

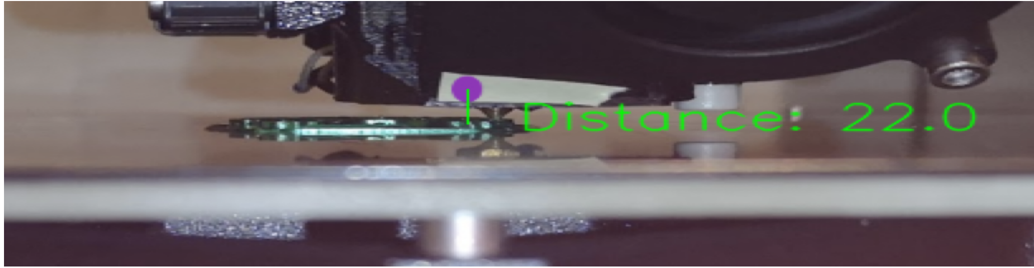


Figure 4.21: Distance between the nozzle and the 3D printed object

the threshold value decided, algorithm 4 is used to verify that the distance between the nozzle and the filament is correct.

Algorithm 4 Verify nozzle distance

```
1:  function VALID_NOZZLE_DISTANCE (p1, p2, prevDistance, threshold)
2:
3:      distance ← euclidianDistance(p1, p2) − threshold
4:
5:      if prevDistance ≠ null then
6:          if (distance − prevDistance) ≥ threshold ∗ 0.5 then
7:              return False
8:          end if
9:      end if
10:     if distance ≥ threshold ∗ 0.5 then
11:         return False
12:     end if
13:     return True
14: end function
```

The nozzle distance verification function is applied to the distance measured in every video frame received from the client. A new distance from the points in current frame is calculated whilst the distance from the previous frame is preserved. If the calculated distance between the two frames are larger than the threshold divided by two, we classify it as an error. We chose the accepted value to be less than half the distance measure from the nozzle

reference point and the boundary box to account for potential pixel errors.

In order to detect detachment and movement in the object itself the same technique with a slight modification is applied. For each frame received from the server the centroid position of the bounding box is calculated. If the centroid has change significantly from the previous frames that has been analysed we classify it as a detachment from the print bed. In our experimental setup we chose the threshold for allowed movement to be a distance of 25% of the total width of the 3D printed object. If the detected movement is above this threshold a pause in the 3D printing process is triggered. For larger models the pixel errors may increase as the object covers a larger area which may change the brightness in the image. Therefore, the selected threshold is a percentage of the model to compensate for potential pixel errors.

4.6 Training Data Collection for CNNs

We are only able to detect a limited types of potential 3D printing defects that may occur in the middle of the printing process using traditional computer vision. By using deep learning in computer vision we allow networks to automatically detect, learn and extract features from raw image data. In this project we investigated the possibilities of using traditional computer vision algorithms to find deviations and detect errors since it will be very time consuming to collect the necessary training data needed for training a neural network. Deep learning models like CNN are usually trained using big data for its extreme precision in image classification [59]. The challenge of applying DL for image classification and fault detection in 3D printing is the lack of dataset. Therefore, as a part of this project, we implemented an API for systematic training data collection and data labeling for use in future work.

From our fault detection prototype all the frames captured during the 3D printing monitoring process are stored in the file-system. These frames contains useful information that can be utilized to train deep learning models for fault detection in FDM 3D printing. By extracting labeled data from the video stream after a print has finished, every single object that is being printed will contribute to create a large training dataset for increased accuracy in fault detection using DL. If our camera system is capturing a video stream with a frame rate of 30 Frames per Second (FPS) we get 30 frames each second. As we have mentioned earlier, it is not uncommon that a printing process lasts for more than 10 hours. A 10 hour printing process will produce 1.08 million frames using a camera that captures images at 30 FPS. However, all of these frames should not be used to train a network. As 30 frames are recorded each second the frames will not differ very much. If we store and label a frame for every 1000-3000 frames that are captured, depending on how large the printed object is, a 10 hour print will produce between 300-1000 images that can be used to train a neural network for image classification.

Training Data Client

For the labeling process there are different categories the data can be labeled as. Each type of the most common errors that can occur during a 3D printing process has its respective category. *Successful Prints* is also a category since a deep learning model must be trained on these as well. A Command Line Interface (CLI) client is developed to process captured frames after a model has been printed. First, the client prompts user to enter whether the print was successful or not. If an error occurs, the user is asked to give a time period in which the error is visible in the video stream. The CLI will process the video by slicing it into frames within the given time periode and move them into a new folder named after the error that took place. Finally, the client asks the user to review the folder containing all the frames to ensure that the defects are visible in all frames. When all images are reviewed and approved the images are sent to the server through a RESTful API endpoint which handles the cloud storage.

RESTful API and Cloud Storage

Storing images and information regarding the image requires a database and a server for talking to the database. We implemented a RESTful API Service using NodeJS, a cross-platform JavaScript Runtime Envronment, to receive the images through an API endpoint and store them in a cloud database. By creating the backend as an API service the resources stored in a database can be accessed and modified by multiple clients through simple HTTP requests.

A NoSQL Cloud database called *MongoDB* [36] is used in this project for simplicity as no particular database setup is necessary and it is commonly used combined with the NodeJS environment.

Figure 4.22 above shows the two API endpoints for fetching and creating new API resources. The API resources are the captured video frames stored in a cloud database. Three different client are communicating with the API service sending different requests. Two clients has video frames of defect 3D

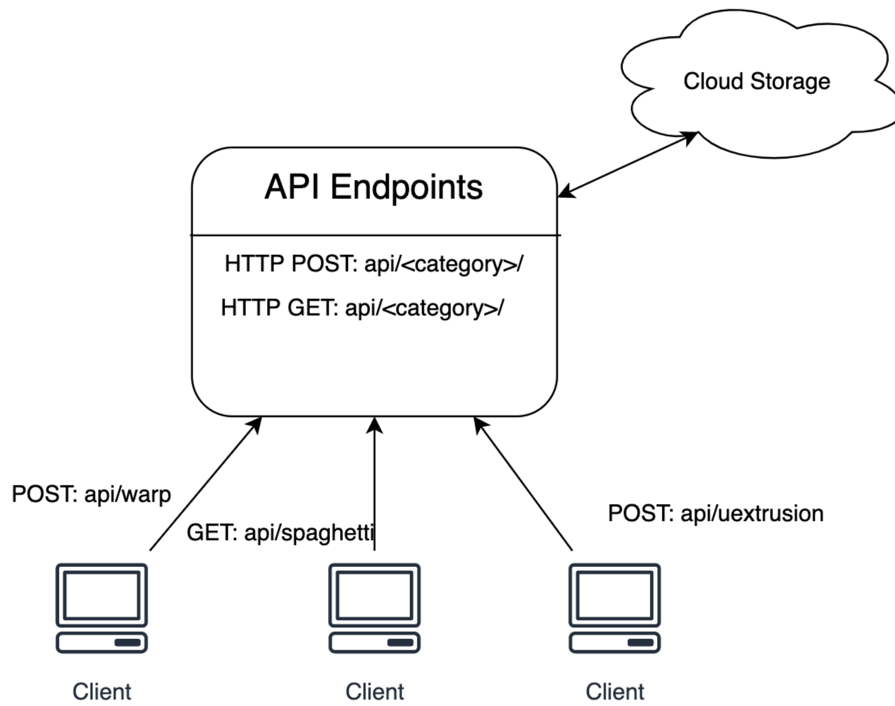


Figure 4.22: Clients posting and requesting training data through HTTP endpoints

prints and sends a POST request to the API endpoints to store the frames in a cloud database. New training images labeled as *warping* and *under extrusion* are stored in the cloud database. The client in the middle is sending a GET request to fetch all images labeled with *spaghetti*, returning the entire data set of images containing *spaghetti print*.

Chapter 5

Results

In this project we focused on the possibilities of using computer vision as a replacement for active monitoring by eye of 3D printing processes. A prototype were built using camera monitoring from two different angles. The prototype consists of the following two main modules:

- First Layer Verification
- Nozzle Analyzer

In section 3.3 we presented a table, table 3.1, which gives a brief overview of common failures that occurs in 3D printing. Our prototype was able to detect several defects covered in the table using traditional computer vision algorithms. However, both of the modules has some weaknesses causing inconsistent image segmentation and inaccurate fault detection in some cases.

5.1 First Layer Verification

A camera was placed on top of the printer facing down towards the build plate. When first layer was completed a snapshot was taken through the camera and sent to the server for verification. In the verification process we managed to successfully compare the 3D printed first layer with the first

layer of the 3D printed model and to detect significant deviations between the images. It is a challenging task to find deviations in highly accurate manufacturing machines by comparing the result with a model of the intended object using computer vision. A consistent method for image segmentation and precise simulation of the sliced model is necessary in order to perform a shape analysis on the first layer. The segmentation requires a certain contrast between the print material and the environment. For our experiment we used a dark colored build plate which makes the segmentation algorithms more reliable.

The developed prototype for verification of first layer were applied to five 3D prints for testing. One of the objects signaled a false fault detection in the first layer. The image taken by the camera were blurry making it difficult to do a proper segmentation of the image. The reason for this false detection are believed to be because the camera were focused on the print head in the moment of capturing the frame to be analysed.

In the second test object we moved the printed first layer manually to the left in order to simulate a detachment of the first layer. Our simulation framework is calibrated with the build plate of the 3D printer creating the same relative distances in the simulation environment as on the real build plate. Our algorithm was able to detect that the centroid had an offset by more than 15% and triggered a fault detection signal.

In the third test object the same model as we used in the previous examples through the thesis. This model contains multiple small contours reducing the bed adhesion significantly. In figure 4.13 we showed that the algorithm is able to find missing contours. However, detached objects does not necessarily move out of the camera's vision range. In this test we instead detached one of the outer contours closer to the center. A defect were detected by calculating the aspect ratio of the models, finding that the 3D printed object had a higher ratio than the simulated object. However, the algorithm were not able to find any deviations buy moving any of the inner contours as it will find that the aspect ratio and the number of contours are matching.

In the last two tests no errors were introduced and passed the first layer verification as expected.

5.2 The Nozzle Analyzer

A camera was placed in front of the 3D printer as shown in figure 4.14 to track the nozzle movement, measure the distance between the printed object and the nozzle and detect any movement in the object itself. As we are only interested in tracking the position of the 3D printed object the shape of the object is irrelevant, thus we have only used a single test object for the nozzle analyzer.

Four different tests were applied on the object that is being printed. In the first test we moved the nozzle manually away from the 3D printed object as soon as the first layer was printed. A signal of no filament flow was triggered after skipping around 2mm, which corresponds to around 10 layers. However, in the second test we printed around 140 layers before we apply the same test again. 140 layers correspond to a 7cm tall object with a layer height of 0.2mm per layer. In this test the algorithm detected that there was no filament flow significantly slower than in the first test. The 3D printed object was taller than the camera lens that was mounted on a tripod making it hard to detect the relative distance between the nozzle and the object.

In the third test we moved the object slightly to the left and to the right to test the object tracking and the movement detection. The movement detection algorithm were able to detect a movement and signaling a detachment when moving the object only 5mm in both directions. The downside with this method is that it is only able to detect movement in the west and east direction and not south and north direction relative to the camera. However, if a detachment occurs and the object gets stuck to the nozzle, a detachment is very likely to be detected as the object will with great certainty move in every possible directions on the build plate. In the last test we tried using the nozzle analyzer to detect molten material that was stuck on the nozzle.

This requires the algorithm to detect material that both belongs to the object that is being printed and material that got stuck on the nozzle. We manually attached some material to the nozzle by heating it up so the material melted and stuck to the nozzle immediately. The goal of this test is to find out if the algorithm is able to detect extruder blobs which also was introduced in table 3.1. It turns out that this was harder than first expected. The reason being is that molten material that is stuck on a heated nozzle changes colour and the image segmentation did not manage to isolate all the material but instead found multiple smaller contours. However, as more material got stuck on the nozzle the more consistent the fault detection was.

Chapter 6

Conclusion

In this thesis we investigated the possibilities of automatic fault detection in additive manufacturing, more specifically 3D printing, by means of traditional computer vision algorithms. A brief overview of the initial literature review done in the beginning of this project shows that affordable desktop sized 3D printers are becoming more popular creating a knowledge gap between the end users. Methods for process monitoring in the manufacturing industry exists, but there is a lack of research in this area for 3D printing. There has been some attempts trying to create methods for automatic fault detection in 3D printing, but these methods have shown to be very specific and will in many cases generate a false fault detection alert which in the long run may be more time consuming than saving time by detecting errors.

6.1 Automatic Fault Detection Prototype

In this project we designed a prototype consisting of two main modules for automatic fault detection and process monitoring of FDM 3D printing processes. The two modules for fault detection is the *First Layer Verification*(FLV) module and the *Nozzle Analyzer*. The FLV module creates a checkpoint in the G-Code before it is sent to the 3D printer, telling it to

pause when the first layer has been printed. A snapshot of the first layer is then taken using a camera mounted on top of the printer. In earlier projects image analysis techniques were applied directly on to the picture taken of the first layer, calculating average RBG values for each checkpoint and then compare the results using supervised learning. However, these methods have not been properly tested and are not able to detect specific defects making them prone to false fault detection alerts.

In our FLV module we investigated the possibilities of comparing the 3D printed first layer with the sliced version of the model that is being printed. A simulation framework for executing the G-Code commands in a 3D environment was created for this purpose. By being able to simulate the exact same layers as the snapshot of our 3D printed model we can accurately detect several faults by applying shape analysis algorithms on both the simulated model and the printed model to detect deviations. With our FLV module we are able to detect missing contours (i.e missing objects in the image) in multi-part prints, detached first layers and potentially deformations depending on how severe the deformation is.

The second module focuses on monitoring the process after the first layer has been verified. This module uses a camera placed in front of the printer and are able to detect missing material flow by measuring the distance between the nozzle and the upper layer of the printed object. We are also able to detect extruder blobs and detached objects by tracking the movements in the video stream. Being able to detect extruder blobs can increase the lifetime of the 3D printer components significantly as this can potentially cause severe damage to the print head.

A third module was also built using our existing monitoring system to slice the video streams into multiple frames at strategical time-periods of the captured video for systematic labeling of training data. As mentioned earlier, we have focused on applying traditional computer vision algorithms for automatic error detection and thus excluding the use of deep learning algorithms for that purpose. This limits our ability to detect other features that can be classified as a fault like the *spaghetti print* from figure 3.7. It is hard to cre-

ate statistical and mathematical image analysing algorithms for detection of these features. Many 3D printed objects also contains features that may be mistaken as a defect by approximating a model for detecting such features. The *Spaghetti Detective* project [25] has shown that deep learning algorithms are able to detect these features by training a deep learning model, but it often generates a false detection due to lack of training data. Therefore, we created a method to systematically collect labeled training data to use in training of deep learning models for different defect categories in future applications.

As these methods were able to detect several common errors that occurs in 3D printing they are very sensitive to changes in lighting conditions and other environmental changes. A good camera calibration is needed for this system to work properly as well as constant lighting. The algorithms for detachment detection are implemented with a tolerance value, meaning that a movement over a certain level must be detected before an alarm is triggered. A consequence of this is that some obvious errors that should have been detected do not reach the movement tolerance for a detection signal.

6.2 G-Code Simulation Framework

A tool for simulating G-Code execution in a 3D environment was developed for our system to gain knowledge about how a manufactured object was intended to look like. We were able to accurately re-create the sliced model in a 3D scene and simulating different camera angles to analyse the model from roughly the same angle as a real camera.

As this framework serves it's purpose, it turns out that our simulation framework also can be used for introducing realistic and random fault detection in the models. The simulated object is re-created by extracting it's 3D coordinates from the G-Code file containing all information about the sliced model and organizing it into a list of *infill* commands and *perimeter* commands. In a multi-part model, like the example in figure 4.3, where one model consists

of multiple parts we are now able to isolate each part and threat is as an isolated model. In a normal 3D printing environment every component of a model are together considered as only one mode. This enables our simulation framework to apply transformation and translation operations directly to a single part of the model. With this data structure we can make the simulation framework to introduce faults like detachment by translating one of the components to another position, or bad extrusion by removing some of the infill. Using our Simulation Framework it is possible to evaluate and test future fault detection algorithms by introducing, either specific or random, errors in the simulated model and apply fault detection algorithms directly into the simulation environment.

Chapter 7

Future Work

Our developed prototype for automatic fault detection in 3D printing works surprisingly well, but it is only built as a proof-of-concept prototype to answer our questions for the feasibility study. The implemented algorithms are able to detect several of the most common errors in 3D printing, but should be extended further to support more consistent error detection with a minimized tolerance level for signaling a fault. This require a more consistent algorithm for segmentation.

For the FLV module it will not detect a detachment from a multi-part model where the detachment occurs for a part that does not affect the aspect ratio. We propose implementing a shape analysis algorithm that takes account for a deviation in the relative distance between all the detected contours. Another extension that should be implemented for the FLV module is symmetry detection in both the simulated model and the 3D printed first layer. If a deformation that does not change the aspect ratio of the model occurs the degree of symmetry in the image will change and can potentially be detected.

With correct training, labeling of data sets and selection of parameters *Convolutional Neural Networks* has shown to be very accurate in solving hard image classification problems and for image segmentation problems [59]. With our implementation using traditional computer vision algorithms for image

segmentation the prototype is very sensitive to external interference. A slight change in the camera angle may cause the camera to perceive another pixel color as the perceived lighting will also change causing the segmentation to be inaccurate. Therefore, we propose incorporating a method using deep learning for improved accuracy of the image segmentation in the future.

As CNN has shown to be accurate in image classification, an attempt to train a CNN using images of printed models with defects as training data is highly encouraged. We believe that by collecting enough training data a CNN will be able to detect these errors with greater precision than using traditional computer vision algorithms.

Bibliography

[1] Diab W. Abueidda, Mohamed Elhebeary, Cheng-Shen (Andrew) Shiang, Siyuan Pang, Rashid K. Abu Al-Rub, and Iwona M. Jasiuk. Mechanical properties of 3d printed polymeric gyroid cellular structures: Experimental and finite element study. *Materials and Design*, 165:107597, 2019.

[2] Challenges of additive manufacturing. `https://www2.deloitte.com/content/dam/Deloitte/de/Documents/operations/Deloitte_Challenges_of_Additive_Manufacturing.pdf`, 2019. Online, Accessed: 2019-11-17.

[3] Mohammad Alsoufi and Abdulrhman El-Sayed. Warping deformation of desktop 3d printed parts manufactured by open source fused deposition modeling (fdm) system. *International Journal of Mechanical & Mechatronics Engineering*, 17:7–16, 08 2017.

[4] Patrick Baudisch and Stefanie Mueller. Personal fabrication. *Foundations and Trends in Human–Computer Interaction*, 10(3–4):165–293, 2017.

[5] Felix Baumann, Martin Schuermann, Ulrich Odefey, and Markus Pfeil. From gcode to stl: Reconstruct models from 3d printing as a service. *IOP Conference Series: Materials Science and Engineering*, 280:012033, 12 2017.

[6] Baumann, Felix and Roller, Dieter. Vision based error detection for 3d printing processes. *MATEC Web of Conferences*, 59:06003, 2016.

[7] Joseph Beaman, Clint Atwood, Theodore Bergman, David Bourell, Scott Hollister, and David Rosen. Additive/subtractive manufacturing research and development in europe. page 155, 12 2004.

[8] Ekaba Bisong. *NumPy*, pages 91–113. 09 2019.

[9] Farid Bounini, Denis Gingras, Vincent Lapointe, and Herve Pollart. Autonomous vehicle and real time road lanes detection and tracking. pages 1–6, 10 2015.

[10] I. Cooper. What is a “mapping study?”. *Journal of the Medical Library Association : JMLA*, 104:76–78, 01 2016.

[11] Robert Dancel. Case study paper on additive manufacturing (3d printing technology), 04 2019.

[12] G-Codes and M-Codes for 3D printing. `http://dfabresearch.com/xo15/wp-content/uploads/2015/01/Marlin_GCodes7.pdf`. Online, Accessed: 2019-09-25.

[13] J. Dirksen. *Learning Three.js – the JavaScript 3D Library for WebGL - Second Edition*. Community experience distilled. Packt Publishing, 2015.

[14] Dana Goldberg. History of 3d printing, it’s older than you are. https://www.autodesk.com/redshift/history-of-3d-printing/, 2018. Online, Accessed: 20.04.2019).

[15] Kadir Gunaydin and Halit Türkmen. Common fdm 3d printing defects. 04 2018.

[16] Ali Güneş, Habil Kalkan, and Efkan Durmuş. Optimizing the color-to-grayscale conversion for image classification. *Signal Image and Video Processing*, 10 2015.

[17] Martin Hardwick and David Loffredo. Step-nc: Smart data for smart machining. 01 2007.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[19] Edna Ho and Sameer Kumar. *The Trends and Challenges of 3D Printing*. 06 2018.

[20] Muzammal Hoque, Md. Hasnat Kabir, and Mehedi Jony. Design and construction of a bowden extruder for a fdm 3d printer uses 1.75mm filament. *International Journal of Technical Research & Science*, 3, 10 2018.

[21] Jing Hu. Study on stl-based slicing process for 3 d printing. 2017.

[22] Cătălin Iancu, Daniela Iancu, and Alin Stăncioiu. From cad model to 3d print via “stl” file format. 2010.

[23] Print quality troubleshooting guide. `https://www.simplify3d.com/support/print-quality-troubleshooting/`, 2019. Online, Accessed: 2019-08-09.

[24] Nebojsa Jaksic. What to do when 3d printers go wrong: Laboratory experiences. *ASEE Annual Conference and Exposition, Conference Proceedings*, 122, 01 2015.

[25] Kenneth Jiang. Tsd - the spaghetti detective. `https://github.com/TheSpaghettiDetective/TheSpaghettiDetective`. (Accessed: 24.09.2019).

[26] Zeqing Jin, Zhizhou Zhang, and Grace X. Gu. Autonomous in-situ correction of fused deposition modeling printers using computer vision and deep learning. *Manufacturing Letters*, 22:11 – 15, 2019.

[27] C. Junhua and L. Jing. Research on color image classification based on hsv color space. In *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 944–947, Dec 2012.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira,

C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[29] Maciej Kujawa. *The influence of first layer parameters on adhesion between the 3D printer's glass bed and ABS*, pages 76–81. 09 2017.

[30] Gcode Index. `http://marlinfw.org/meta/gcode/`. Online, Accessed: 2019-09-13.

[31] Gcode Index m207 - set firmware retraction. `http://marlinfw.org/docs/gcode/M207.html`. Online, Accessed: 2019-09-13.

[32] 3d Printing Materials Guide-All You Need to Know. `https://all3dp.com/1/3d-printing-materials-guide-3d-printer-material/`, 2019. Online, Accessed: 2019-10-02.

[33] Hugo Medellín-Castillo and Joel Torres. Rapid prototyping and manufacturing: A review of current technologies. volume 4, 01 2009.

[34] Rodrigo Minetto, Neri Volpato, Jorge Stolfi, Rodrigo Gregori, and Murilo da Silva. An optimal algorithm for 3d triangle mesh slicing. *Computer-Aided Design*, 92, 07 2017.

[35] Wojciech Mokrzycki and Maciej Tatol. Perceptual difference in l * a * b * color space as the base for object colour identfication. 08 2009.

[36] Mongodb. Available at `https://www.mongodb.com/cloud/atlas` (2020/02/01).

[37] K. Madhavan Nampoothiri, Nimisha Nair, and Rojan P John. An overview of the recent developments in polylactide (pla) research. *Bioresource technology*, 101:8493–501, 11 2010.

[38] Tung Nguyen, Ayumu Shinya, Tomohiro Harada, and Ruck Thawonmas. Segmentation mask refinement using image transformations. *IEEE Access*, PP:1–1, 11 2017.

[39] Frank Nielsen. Detecting lines in images: The hough transform. 01 2011.

[40] Opencv - image thresholding. https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html, 2019. Online, Accessed: 2019-11-10).

[41] Opencv - operations on arrays. https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html, 2019. Online, Accessed: 2019-15-12).

[42] Opengl - the industry's foundation for high performance graphics. https://www.opengl.org/, 2019. Online, Accessed: 2019-10-12).

[43] Perfecting the first layer. https://www.simplify3d.com/support/articles/perfecting-the-first-layer/, 2019. Online, Accessed: 2019-11-09.

[44] R.R Phelps. Dentability and extreme points in banach spaces. *Journal of Functional Analysis*, 17(1):78 – 90, 1974.

[45] Prusa Research. PrusaSlicer. https://www.prusa3d.com/prusaslicer/. Accessed: 14.05.2019.

[46] Alexandru Pîrjan and Dana-Mihaela Petroşanu. The Impact Of 3d Printing Technology On The Society And Economy. *Romanian Economic Business Review*, 7(2):360–370, December 2013.

[47] Wang Qi, Fu Li, and Liu Zhenzhong. Review on camera calibration. pages 3354 – 3358, 06 2010.

[48] Fa'Iq Raihan and Win Ce. Pcb defect detection using opencv with image subtraction method. pages 204–209, 11 2017.

[49] Mechanical Endstop - RepRap Wiki. Online https://reprap.org/wiki/Mechanical_Endstop, 2015. Online, Accessed: 2019-09-25.

[50] Bulent Sankur and M. Sezgin. Image thresholding techniques: A survey over categories. *Pattern Recognition*, 34:1573–1583, 01 2001.

[51] SculptGL - A WebGL Sculpting Tool. `https://stephaneginier.com/`. Online, Accessed: 22.05.2019.

[52] Slic3r. `https://slic3r.org/`. Online, Accessed: 13.05.2019.

[53] Ahmed Solyman. *INTRODUCTION TO COMPUTER VISION (Computer Vision and Robotics)*. 02 2019.

[54] Shakirat Sulyman. Client-server model. *IOSR Journal of Computer Engineering*, 16:57–71, 01 2014.

[55] Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, pages 6.329–6.346, New York, NY, USA, 1964. ACM.

[56] Joao Tavares. Image processing and analysis: Applications and trends. *AES-ATEMA International Conference Series - Advances and Trends in Engineering Materials and their Applications*, 01 2010.

[57] Robert F. Tobler and Stefan Maierhofer. A mesh data structure for rendering and subdivision. 2006.

[58] Ultimaker Cura. `https://ultimaker.com/en/products`. Online, Accessed: 13.05.2019.

[59] Joseph Walsh, Niall O' Mahony, Sean Campbell, Anderson Carvalho, Lenka Krpalkova, Gustavo Velasco-Hernandez, Suman Harapanahalli, and Daniel Riordan. Deep learning vs. traditional computer vision. 04 2019.

[60] L. Wang and H. Ju. A robust blob detection and delineation method. In *2008 International Workshop on Education Technology and Training 2008 International Workshop on Geoscience and Remote Sensing*, volume 1, pages 827–830, Dec 2008.

[61] Shin-Ting Wu, Adler C. G. da Silva, and Mercedes Rocío Gonzalez Márquez. The douglas-peucker algorithm: Sufficiency conditions for non-self-intersections. *J. Braz. Comp. Soc.*, 9(3):67–84, 2004.

[62] W. Z. Wu, P. Geng, J. Zhao, Y. Zhang, D. W. Rosen, and H. B. Zhang. Manufacture and thermal deformation analysis of semicrystalline polymer polyether ether ketone by 3d printing. *Materials Research Innovations*, 18(sup5):S5–12–S5–16, 2014.

[63] Keaton J. Young, James E. Pierce, and Jorge M. Zuniga. Assessment of body-powered 3d printed partial finger prostheses: a case study. *3D Printing in Medicine*, 5(1):7, May 2019.

[64] Youlian Zhu and Cheng Huang. An adaptive histogram equalization algorithm on the image gray level mapping. *Physics Procedia*, 25:601–608, 12 2012.

[65] Karel Zuiderveld. Contrast limited adaptive histogram equalization. *Graphics Gems IV*, pages 474–485, 12 1994.