

**TRAJECTORY PLANNING AND  
SIMULATION FOR 3D PRINTING PROCESS**

**NG CHIN YONG**

**UNIVERSITI TUNKU ABDUL RAHMAN**

**TRAJECTORY PLANNING AND SIMULATION FOR 3D PRINTING  
PROCESS**

**Ng Chin Yong**

**A project report submitted in partial fulfilment of the  
requirements for the award of Bachelor of Engineering  
(Honours) Mechatronics Engineering**

**Lee Kong Chian Faculty of Engineering and Science  
Universiti Tunku Abdul Rahman**

**September 2022**

**DECLARATION**

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature :  \_\_\_\_\_

Name : Ng Chin Yong \_\_\_\_\_


ID No. : 1800216 \_\_\_\_\_


Date : 30/9/2022 \_\_\_\_\_

**APPROVAL FOR SUBMISSION**

I certify that this project report entitled “**TRAJECTORY PLANNING AND SIMULATION FOR 3D PRINTING PROCESS**” was prepared by **NG CHIN YONG** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Honours) Mechatronics Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature :   
\_\_\_\_\_  
Supervisor : Ts Dr Lee Jer Vui  
\_\_\_\_\_  
Date : 2/10/2022  
\_\_\_\_\_

Signature :   
\_\_\_\_\_  
Co-Supervisor : Dr. Chan Siow Cheng  
\_\_\_\_\_  
Date : 03/10/2022  
\_\_\_\_\_

The copyright of this report belongs to the author under the terms of the copyright Act 1987 as qualified by Intellectual Property Policy of Universiti Tunku Abdul Rahman. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

© 2022, Ng Chin Yong. All right reserved.

## ABSTRACT

The demand for Fused Deposition Modelling (FDM) 3D printing technology is skyrocketing among hobbyist makers in recent years especially during the outbreak of pandemic Covid 19. 3D printing community has increased drastically with engagement of hobbyists and users from non-engineering background. The main concern from non-engineering users are affordability, operability, and efficiency. Affordability refers to the financial burden upon purchasing large amount of printing material for reprint the printouts that are not satisfactory. Operability refers to the knowledge needed to adjust the printer's settings. Efficiency refers to the amount of time spent printing on printouts that are not satisfactory. With these feedbacks from the users, a stimulator called CY simulator is proposed in this project to resolve the non-engineering users' concerns.

The aim of this project is to develop a FDM 3D printer simulator that can generate the surface and infill of a 3D print from a STL file. 3D printing process will be simulated according to the generated vertices and printing time will be estimated. Development of the simulator was started with the development of STL file reader to extract the triangular facets' vertices from a STL file in ASCII format. The extracted vertices will be used as the input of slicer to slice the model into layers with user-defined layer height. The slicing method used is basic slicing. The slicing algorithm consists of two sections: intersection point tracking algorithm and contour creation algorithm. The output of slicing algorithm will be used as input of infill generator to generate infill vertices based on the infill density and infill pattern that decided by users.

Next, the output of slicer and infill generator will be rendered accordingly to simulate a lifelike 3D printing process. Vertices are connected by using a hollow cylinder to represent the 3D printing material. Moreover, printing time is estimated by dividing the distance between each vertex by the default printing speed. After that, several experiments were conducted to examine the feasibility of the simulator in terms of layer height, infill density, top and bottom thickness, and estimated printing time.

According to the result, the layer height, infill and top/bottom thickness generated by the simulator can achieve 100 % similarity with the actual print. Apart from that, the simulator has 100 % accuracy of estimating printing time for objects that have lesser layers. However, less accurate of estimating printing time for models that have many layers. This is because the travelling time for the nozzle to travel from the last position of the layer to the first position of the next layer is not taken into consideration of the printing time. The effect of travelling time on the estimated printing time increases with the number of layers and the distance of the travelling time from last point of a layer to the next layer.

## TABLE OF CONTENTS

<b>DECLARATION</b>		<b>i</b>
<b>APPROVAL FOR SUBMISSION</b>		<b>ii</b>
<b>ABSTRACT</b>		<b>iv</b>
<b>TABLE OF CONTENTS</b>		<b>vi</b>
<b>LIST OF TABLES</b>		<b>ix</b>
<b>LIST OF FIGURES</b>		<b>x</b>
<b>LIST OF APPENDICES</b>		<b>xv</b>
 <b>CHAPTER</b>		
<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
1.1	General Introduction	1
1.2	Importance of the 3D Printer Simulator	2
1.3	Problem Statement	3
1.4	Aim and Objectives	4
1.5	Scope and Limitation of the Study	4
<b>2</b>	<b>LITERATURE REVIEW</b>	<b>6</b>
2.1	Introduction	6
2.2	The Concept of a 3D Printing Simulator	7
2.3	STL File Reader	10
2.4	Structure of a 3D print model	11
2.5	Slicing Method	12
2.6	Intersection points' tracking method	15
2.7	Infill of 3D Printing	17
2.8	3D Graphics Engine	18
2.8.1	Rendering Line and Triangle	19
2.8.2	Rendering Pipe	20
2.9	Summary	23
<b>3</b>	<b>METHODOLOGY AND WORK PLAN</b>	<b>25</b>
3.1	Introduction	25



3.2	Project Planning and Milestone	25
3.3	Program Flow of 3D Printer Simulator	26
3.4	User Input	28
3.5	Workspace	29
3.6	Vertex Extractor of STL file in ACSII format	30
3.7	Slicer	34
	3.7.1 Case I	36
	3.7.2 Case II	37
	3.7.3 Case III	38
	3.7.4 Case IV	41
3.8	Contour Creation	42
3.9	Infill	44
	3.9.1 Infill Density	44
	3.9.2 Infill Pattern	45
3.10	Top and Bottom layer	51
3.11	Rendering Process	51
	3.11.1 Solid Body	52
	3.11.2 Filament Rendering	54
3.12	Gantries movement	62
3.13	Experiments	64
3.14	Summary	65
<b>4</b>	<b>RESULTS AND DISCUSSION</b>	<b>66</b>
4.1	Introduction	66
4.2	User Documentation	66
	4.2.1 Preparation of STL File in ASCII Format	66
	4.2.2 Input Print Parameter	67
4.3	Simulation Results	71
	4.3.1 Disk	71
	4.3.2 Cylinder	77
4.4	Cube	85
	4.4.1 Bracket	88
	4.4.2 Spoon	90
4.5	Summary	93
<b>5</b>	<b>CONCLUSION AND RECOMMENDATIONS</b>	<b>94</b>

	viii
5.1 Conclusions	94
5.2 Contributions	94
5.3 Recommendations for future work	96
<b>REFERENCES</b>	<b>98</b>
<b>APPENDICES</b>	<b>102</b>

**LIST OF TABLES**

Table 3.1:	Gantt Chart of FYP Part 1.	26
Table 3.2:	Gantt Chart of FYP Part 2.	26
Table 3.3:	The infill gap, $d$ with respective infill percentage.	44
Table 3.4:	The combinations of 3 vertices to form 12 facets for the cube.	53
Table 3.5:	The functions defined and usage of the functions for object rendering in pipe form.	58
Table 3.6:	Debavit-Hartenberg Table.	63
Table 4.1:	Default print setting.	67
Table 4.2:	Print parameters for four disk.	72
Table 4.3:	Test result of the disks.	72
Table 4.4:	Print parameters for four cylinders.	77
Table 4.5:	Test result of the cylinders.	78
Table 4.6:	Print parameters for four cubes.	86
Table 4.7:	Test result of the cubes.	86
Table 4.8:	Print parameters for the bracket.	88
Table 4.9:	Test result of the bracket.	88
Table 4.10:	Print parameters for the spoon.	90
Table 4.11:	Test result of the spoon.	90

## LIST OF FIGURES

Figure 2.1:	Interface of ULTIMAKER CURA software.	9
Figure 2.2:	The 3D printing workflow (Materialise Software, 2020).	9
Figure 2.3:	Workflow of 3D Printing Process.	9
Figure 2.4:	STL Binary format	10
Figure 2.5:	STL ASCII format	10
Figure 2.6:	Components made a 3D print item.	11
Figure 2.7:	Xu, Gu et al. 2018 - A review of slicing methods.jpg	12
Figure 2.8:	Possible intersection cases (Topçu, Taşcıoğlu and Ünver, 2011).	13
Figure 2.9:	Algorithm for detecting intersections (Topçu, Taşcıoğlu and Ünver, 2011).	14
Figure 2.10:	Intersection points' tracking (Pan, X., Chen, K. and Chen, D, 2014).	15
Figure 2.11:	Two contour loops tracked by marking method (Pan, X., Chen, K. and Chen, D, 2014).	16
Figure 2.12:	Infill Density.	17
Figure 2.13:	Infill pattern from Makerware (Gopsill, Shindler and Hicks, 2018).	18
Figure 2.14 :	Primitives for rendering lines	19
Figure 2.15:	Primitives for drawing triangles	20
Figure 2.16:	Triangular Prism	20
Figure 2.17:	Rectangular Prism	20
Figure 2.18:	Octagonal Prism	21
Figure 2.19:	Hex decagonal Prism	21
Figure 2.20:	A vertex coordinate on a cylinder (Ahn, 2019).	21
Figure 2.21:	Extruding a pipe along a path Q1-Q2-Q3 (Ahn, 2019)	23

Figure 2.22:	Cross-section view of extruding a pipe (Ahn, 2019)	23
Figure 2.23:	Pipe	23
Figure 3.1:	Flowchart of 3D printing simulation program.	28
Figure 3.2:	A window prompt that gets user input and shows the printing info.	29
Figure 3.3:	A 3D space created for the simulation of 3D printing.	30
Figure 3.4:	Graphical visualization of a STL file.	31
Figure 3.5:	Flowchart of STL file reader.	32
Figure 3.6:	Presentation of imported vertices.	32
Figure 3.7:	Flowchart of centring a 3D model in 3D space.	33
Figure 3.8:	Position of 3D model before centring.	34
Figure 3.9:	Position of 3D model after centring.	34
Figure 3.10:	Possible intersection cases (Topçu, Taşcıoğlu and Ünver, 2011).	35
Figure 3.11:	An illustration of slicing triangular facets.	35
Figure 3.12:	Flowchart of slicing algorithm.	36
Figure 3.13:	Flowchart of Case I.	37
Figure 3.14:	Redundant points in a circle	37
Figure 3.15:	Case II.	38
Figure 3.16:	An illustration of slicing a triangle under Case III.	39
Figure 3.17:	Unwanted type of triangles.	40
Figure 3.18:	Flowchart of Case III.	40
Figure 3.19:	An illustration of triangle under Case IV.	41
Figure 3.20:	Flowchart of Case IV.	41
Figure 3.21:	Flowchart of assigning start and end point.	42
Figure 3.22:	Flowchart of Contour Creation program.	43

Figure 3.23: Vertical infill.	46
Figure 3.24: Horizontal infill	47
Figure 3.25: Flowchart of infill vertex generator	48
Figure 3.26: Line infill pattern's illustration.	49
Figure 3.27: Linear interior geometric generation.	50
Figure 3.28: Flowchart of interchanging vertices for linear infill pattern.	50
Figure 3.29: A cube that formed by GL_TRIANGLES.	53
Figure 3.30: A tetracontaoctagon	54
Figure 3.31: Correct formation of pipe	55
Figure 3.32: Failure of pipe formation	55
Figure 3.33: Possible cases for the angle between a line and the X-axis. <sup>56</sup>	
Figure 3.34: A triangle.	58
Figure 3.35: renderPrinted (matcha green), CurrPrinted (purple),renderPrinting (white) and PrintedInfill (cyan).	60
Figure 3.36: renderInfill (pink), CurrInfill (pickle green) and CurrLayer (red).	60
Figure 3.37: Preview of a sliced model in pipe form.	61
Figure 3.38: Sliced data rendered with GL_LINES.	62
Figure 3.39: Cartesian Robot Schematic Diagram	63
Figure 4.1: SOLIDWORKS interface to save CAD model.	66
Figure 4.2: Save CAD model as STL file in ASCII format.	67
Figure 4.3: Import STL file and show the default print setting.	68
Figure 4.4: Description of print parameters.	69
Figure 4.5: User Interface after the customization of print setting.	70
Figure 4.6: Disk was modelled in SOLIDWORKS.	71

Figure 4.7:	Disk is presented in simulator with layer height of 0.12 mm.	73
Figure 4.8:	Disk is presented in simulator with layer height of 0.16 mm.	73
Figure 4.9:	Disk is presented in simulator with layer height of 0.20 mm.	74
Figure 4.10:	Disk is presented in simulator with layer height of 0.28 mm.	74
Figure 4.11:	Infill of disk with layer height of 0.12 mm.	75
Figure 4.12:	Infill of disk with layer height of 0.16 mm.	75
Figure 4.13:	Infill of disk with layer height of 0.20 mm.	76
Figure 4.14:	Infill of disk with layer height of 0.28 mm.	76
Figure 4.15:	Section view of the disk.	77
Figure 4.16:	Cylinder with infill density of 10 %.	78
Figure 4.17:	Measurement of 4 mm infill gap for 10 % infill density in actual printing.	79
Figure 4.18:	Cylinder with infill density of 20 %.	79
Figure 4.19:	Measurement of 2 mm infill gap for 20 % infill density in actual printing.	80
Figure 4.20:	Cylinder with infill density of 30 %.	80
Figure 4.21:	Measurement of 1.3 mm infill gap for 30 % infill density in actual printing.	81
Figure 4.22:	Cylinder with infill density of 40 %.	81
Figure 4.23:	Measurement of 1 mm infill gap for 40 % infill density in actual printing.	82
Figure 4.24:	Cylinder with infill density of 50 %.	82
Figure 4.25:	G-code viewer to verify actual infill gap for 50 % infill density.	83
Figure 4.26:	Cylinder with infill density of 70 %.	83

Figure 4.27:	G-code viewer to verify actual infill gap for 70 % infill density.	83
Figure 4.28:	Cylinder with infill density of 90 %.	84
Figure 4.29:	G-code viewer to verify actual infill gap for 90 % infill density.	84
Figure 4.30:	Infill changes direction on alternate layer.	85
Figure 4.31:	Cube is rendered in simulator with layer height of 0.20 mm.	86
Figure 4.32:	Top and bottom layers of the cube in simulator.	87
Figure 4.33:	Actual top and bottom layers of the cube from 3D print.	87
Figure 4.34:	G-code viewer to verify the real line gap of top and bottom layer.	87
Figure 4.35:	View of the bracket in simulator.	88
Figure 4.36:	Isometric view of the bracket's shell.	89
Figure 4.37:	Isometric view of the bracket's infill.	89
Figure 4.38:	Actual 3D printed bracket.	90
Figure 4.39:	Spoon is rendered in solid body mode.	91
Figure 4.40:	Appearance of the 3D printed spoon in the simulator.	91
Figure 4.41:	Shell of the spoon in the simulator.	92
Figure 4.42:	Infill of the spoon in the simulator.	92
Figure 4.43:	Actual 3D printed spoon.	93



**LIST OF APPENDICES**

Appendix A: ULTIMAKER CURA	102
Appendix B: Coding	104
Appendix C: Motion of Ender 3 3D Printer	168

## CHAPTER 1

### INTRODUCTION

#### 1.1 General Introduction

Back to the time of World War II, the concept of computer simulation is proposed by two mathematicians – Jon Von Neumann and Stanislaw Ulam and others who were faced with the problem of analysing the diffusion of neutrons. A simulator of the process was developed to solve the problem safely with minimal cost. This alludes that the capability of a simulator to solve engineering problems safely and efficiently. However, the simulator industry was not arisen due to the shortage of skilled men and experts in particular field. Today's, the obstacle has been reduced due to the evolution of technology. Reduction of manufacturing costs, training costs, and preventing defective items are the main concerns of a business in recent days. Therefore, the demand of simulation software has increased drastically and getting important in the today's and future.

Additive manufacturing (AM) or additive layer manufacturing (ALM) which is the industrial production name for 3D printing. Additive layer manufacturing which facilitates the manufacturing industry a new method of fabrication. This method of converting raw material to an object reduce material waste, unlike the traditional manufacturing methods which are subtractive, forming, and casting. Over the last decade, the popularity of additive manufacturing has been inclined exponentially. The hardware for 3D printing is called 3D printer. The 3D printer is widely used in industrial production, and it is available for personal use and for SMEs to use for small-scale projects. The increasing demand of additive manufacturing increases the necessary for having simulator to increase quality of product. Moreover, to control and estimate the material cost and manufacturing time.

In this study, our major focus will be on the simulation of 3D printing. The 3D printer simulator is to create a visualization platform of the layered object. The simulator reads a 3D object and render it in a 3D space layer-by-layer with specified layer height.

This study is to design and develop a simulator with capability to render the trajectory of 3D printing process progressively and to visualize the movement of the 3D printer. The layered 3D object will be rendered in a window after the user inputs a STL file of a 3D object with the nozzle of 3D printer to visualize the printing process.

## **1.2 Importance of the 3D Printer Simulator**

The rise of 3D printing in this decade has facilitated several industries such as aerospace, automotive, manufacturing, robotics, construction, healthcare. According to the statistic, the market of 3D printing bringing to an industry has grown from 4.4 billion USD in 2013 to 21 billion USD in 2021 (Evans, 2021). The number of manufactures that adopt the 3D printing technology for industrial-scale production has doubled in between 2018 and 2019. It is stated that there are 40% of the manufacturers implant this technology into their manufacturing in 2019 (Aaryaman Aashind, 2021). This clearly illustrates that the massive adoption of 3D printing and it can also be implemented to manufacture variety of products and goods due to its flexibility. In 2020 and 2021 – the occurrence of global crisis of COVID-19 pandemic, 3D printing technology was grown drastically. The community of 3D printing increases, they shared their knowledge of 3D printing to help in terminating this crisis. With the knowledge, this technology contributed to solve the shortage of much-needed medical supplies and the disruption of logistics. 3D printing is used to fabricate medical devices such as face shield, ventilator valve, non-invasive PEEP mask. Personal protective equipment (PPE) like respirators and metal respirator filters, and testing devices – nasopharyngeal (NP) swab are fabricated through 3D printing technology as well (Choong et al., 2020). A survey was conducted and found that the highest application of this technology is prototyping which consists of 72%, followed by research and development (44%), repair (43%) and the production of parts (39%) (Aaryaman Aashind, 2021). Other than that, 66% of companies stated that the reason of using 3D printing is provides a higher productivity of manufacturing, while 61% declared that the ability of customizing parts is the main reason of using the technology (Aaryaman Aashind, 2021).

The evidence above has proven the rise of 3D printing and the demand of it in the near future which will be widely applied in various of industries. The rising of demand of 3D printing making the growing importance of simulation of 3D printing. The simulation is vital for engineers to visualize the upcoming printing process and validate the orientation of the object on the printing platform. Different orientation of the object will affect the amount of material consumed and printing time. Moreover, it can avoid some printing issue like warping and overhanging in Fused Deposition Modelling (FDM) type of printing. Thus, it is important to analyse and study with a simulator before executing a print. On top of that, visual tool like 3D printing simulator will be an educational tool for public to learn and understand the 3D printing technology. To get the public prepare for the coming Industry 4.0.

### **1.3 Problem Statement**

In this research, the simulation of 3D printing will be mainly focusing on the Fused Deposition Modelling technology. FDM allows rapid prototyping and part manufacturing in industries. The demand for FDM 3D printing technology is skyrocketing among hobbyist makers in recent years especially during the outbreak of pandemic Covid 19. 3D printing community has increased drastically with engagement of hobbyists and users from non-engineering background. The main concern from non-engineering users are affordability, operability, and efficiency. Affordability refers to the financial burden upon purchasing large amount of printing material for reprint the printouts that are not satisfactory. Operability refers to the knowledge needed to adjust the printer's settings. Efficiency refers to the amount of time spent printing on printouts that are not satisfactory.

Simulation provides a solution to reduce and minimize the problems encountered for FDM. With simulation software of a FDM 3D printing, the object can be viewed layer by layer which ease the engineers to identify the potential regions for warping and overhanging to occur. The engineers can whether redesign the regions by adding support structures or adding chamfer/fillets features to eliminate the overhanging problem. Besides that, change the orientation of an object to reduce or eliminate the problems. With 3D printing

simulator, the problems will be overseen and eliminated. Simulation helps the industry to reduce the error of printing and increase the efficiency of rapid prototyping and part manufacturing.

#### **1.4 Aim and Objectives**

In this project, the aim is to develop a FDM 3D printing simulator that can be used to slice a STL file (ASCII format) for 3D print and simulate a 3D printing process. The objectives of this project are:

1. Develop a STL file reader to extract the triangular facets' vertices from an ASCII format STL file. Slicing algorithm is developed to slice the triangular facets into layers with user-defined layer gap. Contour creation is developed to organize the slice vertices orderly.
2. Design and develop an infill generator for interior of a 3D print object with user-defined infill density and infill patterns. Develop a skin generator for exterior of a 3D print object with user-defined thickness.
3. Develop a render program to simulate the realistic 3D printing process and present a lifelike 3D print object with user-defined rendering mode. Develop a simulator to estimate the printing time.
4. Conduct experiments with different layer height, infill line gap, top/bottom thickness, to tune the simulator's print parameters to match with real print parameters.

#### **1.5 Scope and Limitation of the Study**

The scope of this project is to design a FDM 3D printer simulator that can render the surface and infill of a 3D print object layer by layer. The first part of this project is to develop a STL reader that extract vertices data from STL file in ASCII format and the vertices data will then be analysed and sliced into layers. Then, develop an infill generator to fill in the interior of the object. The second part of the project is to simulate the printing process and estimate the printing time.

Limitations of this project:

- 1) Generating multiple layers for the wall of a 3D print object can be complex and computational expensive. The algorithm requires to compute the offset of the contour that formed in Contour Creation algorithm. Complexity of the algorithm subject to the complexity of the outer layer's shape.
- 2) Multiple choices of infill pattern can be difficult, complex and computational expensive. Thus, infill patterns of the project are limited to linear and line infill pattern.
- 3) Polygon infill algorithm can be complex and difficult to develop. Hence, the simulator is limited to non-hollow object.

## CHAPTER 2

### LITERATURE REVIEW

#### 2.1 Introduction

3D printing simulations aid in the understanding and visualisation of the complex thermo-mechanical phenomena that occur during manufacturing, resulting in high-quality, high-accuracy components (Vaissier, n.d.). In the research literature, simulation of Flexible Manufacturing System (FMS) has gotten a lot of interest. The flexibility aspect of FMS according to Chang et al. (1986), makes the design of such systems exceedingly complicated, and their possible simulation an appealing design and analysis tool. The fundamental problem with traditional analytical approaches is that they require the use of simplification assumptions that are best suited to the analysis of systems in a steady state, but this is not possible in the case of flexible production platforms (Avventuroso et al., 2017). Today, simulation is regarded as a critical technique for gaining a better understanding of system behaviour. It is incredibly significant since it provides for an analytical understanding of the system (Hajihosseini et al., 2009). Simulation models have been used to test system performance outcomes achieved in mathematical models since the 1970s (ElMaraghy, 1982). By nature, 3D printing is time and material consuming. For example, in Fused Deposition Modelling (FDM), the filament goes through two cycles of fusion (semi-liquification of the filament) and deposition (cooling and solidification of the material), each of which takes a long time (Luo et al., 2014). It is expected that 3D printing will take hours, if not days, to complete. There is no method to make a partial adjustment. If a fully printed model is not satisfactory, it must be reprinted with same amount of time and expense as starting from scratch (Luo et al., 2014). It shows the needs of 3D printer simulator to help the 3D printing process to address the affordability, operability, and efficiency challenges.

In the following section, the aim is to introduce the concept and theoretical foundation of 3D printing simulator based on the prior published literature.

## 2.2 The Concept of a 3D Printing Simulator

Simulators have a long history in industrial manufacturing, particularly for Computer Numerical Control (CNC) systems. Prior to the rise of 3D printing, CNC systems mostly used for cutting, milling, and routing operations to create real items from digital models (Luo, 2014). The simulator for CNC systems helps users to prepare tool path, allow operator to test the NC codes. These simulations are based on real NC codes. It simulates the movements of a CNC machine. All forms of collisions, model gouges, quick cuts, and NC code problems may be easily identified in a simulation environment (Manus, 2015). Simulator like CNC simulators provide a safe and educational platform for students to possess the knowledge. Moreover, users can reduce production time and costs by using virtual simulations of both complex and simple processes. Different solutions and machine settings can be compared in a short amount of time in order to choose the best one. Interferences and conflicts between mobile and fixed parts can be detected and removed (Lo Valvo et al., 2012). As a result, simulator of CNC systems emphasizes the importance of process simulation for 3D printing with the statement made by Lo Valvo et al. (2012) that simulation is a tool capable of analysing various machining strategies not only quickly but also without causing any damage, risk, waste, or breakdowns.

Manufacturing parameters must be integrated to adequately define the geometry of the virtual model in order to construct a visual simulation tool according to the rapid prototyping process regulations (Jee & Sachs, 2000). In fact, the simulation paradigm of 3D printing proposed in this project cannot cover all the issues in the rapid prototyping. For instance, issues encountered in additive manufacturing but yet to be provided in present simulator are material, surface property, and dimensional tolerance (Jee & Sachs, 2000), layer misalignment, layer shift, and lower part shrinkage (Tractus3D, n.d.). 3D printer simulators that developed by companies such as Amphyon, Siemens, MSC and Materialise possess more features for analysis of the printing process.

The additive manufacturing simulator from the Siemens have been developed to estimate the material consumption, the build time, region that required support structure. The Siemens simulator can generate support structure for those regions that are overhanging. The simulator estimates the



build time and material consumption for the support structure as well (Anand et al., 2018). On top of that, the simulator provides visualization of the object for different build orientation, layer height, density of the support structure, infill density, support pattern and pattern of adhesion structure. All these are significant manufacturing parameters for the rapid prototyping. These parameters can affect the material consumption, productivity, quality of the items. Not only that, but they can also affect the subsequent tasks such as the removal of the support structure after the item is printed (Anand et al., 2018). Other than Siemes, ULTIMAKER CURA also provides a simulation tool for students and hobbyists who desired to engage in additive manufacturing field. Figure 2.1 shows the interface of ULTIMAKER CURA and the sliced object. In Figure 2.1, the light blue circular line shows the adhesion structure, the yellow structure is the infill of the object, the shell of the object is shown in red and inner wall is shown in green. The right bottom side shows the estimated time spent to print an item and material required which measured in grams.

Furthermore, the workflow of the simulator that have developed from Materialise is illustrated in Figure 2.2. First of all, STL file of the object is imported into the simulator. Next, the simulator will analyse the vertices data of the item and store them in another array for 3D printing preparation. After that, execute the design analysis by tuning the parameters that mentioned above to optimize the printing process. Then, ensure all the region are well-prepared for printing. For instance, the regions that are overhang are added with support structure. After analysing and the model is well-prepared then slice the model into layers and print. This is the proper workflow for a 3D printing engineer to carry out in a simulator before 3D print an object. Therefore, a 3D printer simulator has to possess similar workflow. In Figure 2.3 shows the general workflow of 3D simulation. The method of import STL file, read the data from STL file, slicing the object and rendering are the main concern in this literature review. The research studied and conducted on STL file is stated in Chapter 2.3. While the data slicing and rendering are mentioned in Chapter 2.5 and Chapter 2.8, respectively.

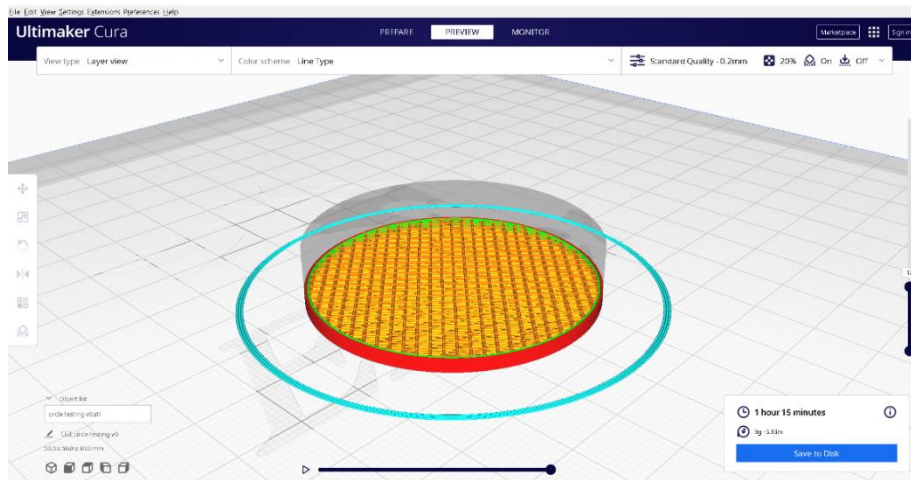


Figure 2.1: Interface of ULTIMAKER CURA software.



Figure 2.2: The 3D printing workflow (Materialise Software, 2020).

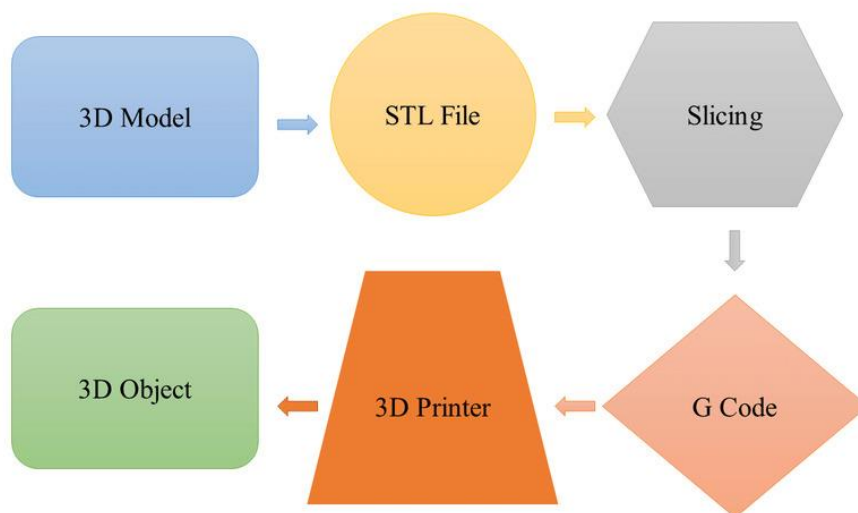


Figure 2.3: Workflow of 3D Printing Process.

### 2.3 STL File Reader

In 3D printing, the STL (STereoLithography) file format is widely regarded as the de-facto additive manufacturing (AM) data interchange standard. The conversion entails running a surface triangulation procedure, which is commonly used in finite element analysis (Topçu, Taşcıoğlu and Ünver, 2011). However, before the contour projection operation can begin, the STL file must be sliced into layers of 2D contours (Adnan et al 2018). The STL file production method mostly turns the CAD file's continuous geometry into a header, small triangles, or a coordinates triplet list of x, y, and z coordinates, as well as the normal vector to the triangles (Wong and Hernandez, 2012). There are two types of STL format which is binary and ASCII. The binary STL file's syntax is shown in Figure 2.4 while Figure 2.5 shows how data of a 3D object is stored in an ASCII format of STL file.

Bytes	Data type	Description
80	ASCII	Header. No data significance.
4	unsigned long integer	Number of facets in file
4	float	$i$ for normal
4	float	$j$
4	float	$k$
4	float	$x$ for vertex 1
4	float	$y$
4	float	$z$
4	float	$x$ for vertex 2
4	float	$y$
4	float	$z$
4	float	$x$ for vertex 3
4	float	$y$
4	float	$z$
2	unsigned integer	Attribute byte count

Figure 2.4: STL Binary format

```

solid name
  {
    facet normal  $n_i n_j n_k$ 
    outer loop
    vertex  $v1_x v1_y v1_z$ 
    vertex  $v2_x v2_y v2_z$ 
    vertex  $v3_x v3_y v3_z$ 
    endloop
  } endfacet
endsolid name

```

Figure 2.5: STL ASCII format

## 2.4 Structure of a 3D print model

Traditional manufacturing does not distinguish between the model's internal and exterior portions. Instead, each part is a single full or hollow body. In contrast, because the machine produces the two sections in completely different ways, the inner and outer 3D printed parts are theoretically independent.

A 3D print's interior is referred to as the infill, while its exterior is referred to as the shell. The infill can be printed in a range of distinct architectures and densities ranging from 0% (hollow) to 100% (solid). But the shell is printed entirely solid.

The shell not only stands out the most in a 3D print, but it also significantly affects the mechanical characteristics of the model (e.g. strength). It consists of a print's walls, top layer, and bottom layer. The top and bottom layers cover the horizontal regions, while the former makes up the vertical outside sections that span the height of a print.

The walls encircle the print's horizontal boundary and continue upward along the Z-axis, making these two sections unique from one another. The top and bottom layers completely enclose the horizontal space inside the boundaries set by the walls.

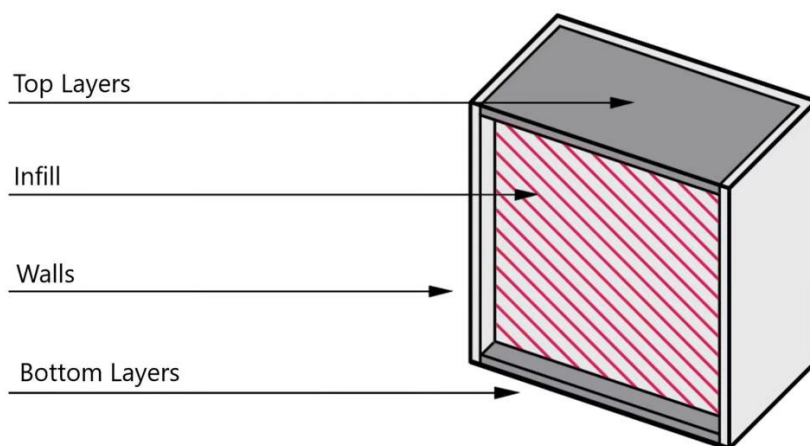


Figure 2.6: Components made a 3D print item.

The shell of a 3D print model can be generated by slicing the triangular facets into layers. This can be done through several type of slicing method which will be discussed in Chapter 2.5.

## 2.5 Slicing Method

The steps for slicing the part's triangulated surfaces, tool path generation for each layer, and tool path data conversion to G-codes are all part of the process planning (Topçu, Taşcıoğlu and Ünver, 2011). More attention should be made to refining the slicing procedure in order to improve contour accuracy, surface quality, and reduce the requirement of support structures (Xu et al., 2018). Traditional, multidirectional, and non-layer wise slicing methods are the three types of slicing methods. Traditional slicing method is expanded into basic slicing and adaptive slicing. Figure 2.7 shows a summary of the slicing methods.

Table I A summary of the published slicing methods

Category	Method	Feature	Advantage	Disadvantage
<b>Basic slicing</b>	Cusp height method	Constant layer height	Easy application High versatility	Rough accuracy Can handle only simple geometric parts
<b>Adaptive slicing</b>	Contour extrapolation Stepwise uniform refinement Local adaptive slicing Accurate exterior, fast interior	Varied layer height with regard to surface complexity or accuracy requirement	Better surface quality Implement complexity in build direction Easy applicable	Cannot handle overhang structures Complexity is still restricted
<b>Multi-direction slicing</b>	Silhouette edges Transition wall Decomposition-regrouping	Multi-direction deposition Less support structure	Reduce reliance on support structure Implement complexity in several directions	Complex algorithm May need manual inspection
<b>Non-layerwise slicing</b>	Overhang structure segmentation Centroidal axis extraction Cylindrical coordinate slicing	Considering geometric information No support structure	Can reduce or eliminate staircase effect theoretically No need for support structure Implement complexity in infinite directions	Complex algorithm Specialization, lack of universality

Figure 2.7: A review of slicing methods (Xu, Gu et al., 2018)

Basic slicing – a sequence of parallel planes with a constant layer height slices the input CAD model, which is commonly in STL format, into subsequent layers (Wong and Hernandez, 2012). The basic slicing method results a uniform layer thickness for input STL file (Mohan Pandey, Venkata Reddy and Dhande, 2003). The staircase effect is caused by the basic slicing procedure, which increases surface roughness and reduces the precision of the produced parts (Singh and Dutta, 2003). Adaptive slicing adjusts layer thickness along the build direction based on the geometry of the CAD model to increase surface finishing quality and reduce construction time (Xu et al., 2018). For complicated designs, the multi-direction slicing method is proposed with the goal of achieving a better approximation of the complex geometric surface, improving surface quality, increasing contour accuracy, and eliminating support structures (Xu et al., 2018).

The slicing method that is focused in this project is the basic slicing method. The slicing method identifies intersection locations between the imported STL data of the solid body and equally spaced slicing planes. In the basic slicing algorithm proposed by Topçu, Taşcıoğlu and Ünver (2011), the gap between each slicing plane is  $z_{step}$  which is a constant value. Figure 2.8 reveals potential connections between the slicing plane and a model facet.

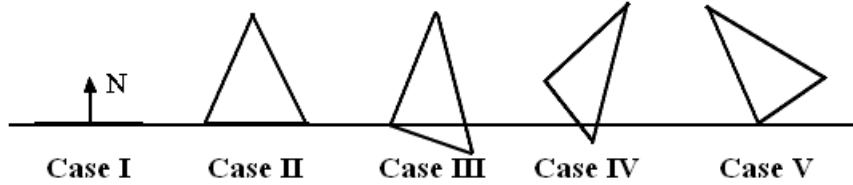


Figure 2.8: Possible intersection cases (Topçu, Taşcıoğlu and Ünver, 2011).

This facet will now be referred to as  $F$ , its vertices as  $V_{1,2,3}$ ,  $(x_{1,2,3}, y_{1,2,3}, z_{1,2,3})$ , and its normal vector as  $N$ ,  $(x_n, y_n, z_n)$ . Vertices are used in the slicing algorithm to compute the intersection points of each triangular facet. On the other hand, normal vectors are not used in slicing algorithm as they are useful for the graphical representation of the solid mode rendering. As illustrated in Figure 2.8 and described below, there are five distinct positional relationships between the slicing plane and the associated facet depending on the  $z$ -axis value of the slicing plane (Topçu, Taşcıoğlu and Ünver, 2011):

1.  $z_1=z$  &  $z_2=z$  &  $z_3=z$ .
2.  $z_1=z$  &  $z_2=z$  &  $(z_3 < z \parallel z_3 > z)$ .
3.  $z_1=z$  &  $((z_2 < z \text{ \& } z_3 > z) \parallel (z_2 > z \text{ \& } z_3 < z))$ .
4.  $((z_1 < z) \text{ \& } ((z_2 > z \text{ \& } z_3 > z)) \parallel ((z_1 > z) \text{ \& } ((z_2 < z \text{ \& } z_3 < z)))$ .
5.  $z_1=z$  &  $((z_2 > z \text{ \& } z_3 > z) \parallel (z_2 < z \text{ \& } z_3 < z))$ .

According to Topçu, Taşcıoğlu and Ünver (2011), since all three of the  $F$ 's vertices serve as the intersection points in CASE I, where  $F$  is parallel to the base, there is no need to compute the intersection points. In CASE II,  $F$ 's two vertices make contact with the slicing plane, and these two spots are where the intersection occurs. These vertices' values and the remaining

vertex's  $z$  value, which will be used to decide whether to keep or discard the data, comprise the stored information.

One of the  $F$ 's vertices touches the slicing plane in CASE III. The common version of the well-known line equation (2.1), where the slope is  $m$  and the two variables are  $x$  and  $y$ , must be used to compute the other intersection. The associated points'  $z$  values are both kept as "-1" in CASE III. The program can identify the obtained line data in subsequent sections of the code.

$$y = mx + c \quad (2.1)$$

In CASE IV, none of the vertices of the  $F$  intersect the slicing plane, yet whether viewed from the  $x$ - or  $y$ -axis, one of the vertices is on the opposite side of the plane. This situation results in two intersecting points, as expected. The label "-1" is likewise present on the obtained line data.  $F$  hits the slicing plane in CASE V only once, which is unnecessary information for the remaining code. All of the points on the slicing plane with the designated categorization reference values are the outputs of the intersection point detection technique.

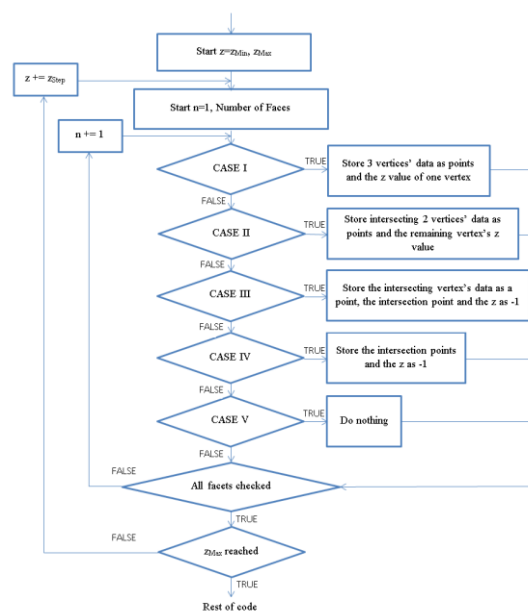


Figure 2.9: Algorithm for detecting intersections (Topçu, Taşcıoğlu and Ünver, 2011).

## 2.6 Intersection points' tracking method

The original intersection points linked list contains the intersection points between the slicing plane and all facets in a disordered state. According to Pan, X., Chen, K. and Chen, D (2014), it is suggested to order the intersecting points using the tracking method, which may then be used to create linked lists of ordered intersection points (also known as contour loops). The method determines which vertices are closest to one another, and then joins their intersection points to form a line. This line is then connected to the other points of intersection to form a polygon (Brown and Beer, 2013).

The slicing plane intersects facet 1 and facet 2 as seen in Figure 2.10. The locations of intersection are, correspondingly,  $X_1$ ,  $X_2$ , and  $X_3$ ,  $X_4$ . They are all kept in the original linked list of intersecting places, where  $X_2$  and  $X_3$  meet. The "next" pointer of  $X_1$  can be used to track  $X_2$  while building an ordered linked list with  $X_1$  as the beginning point. Once it is determined that  $X_2$  and  $X_3$  coincide,  $X_3$  can be tracked. The ordered linked list now includes  $X_3$ . Track  $X_4$  next using  $X_3$  as "next" pointer. The ordered linked list can be obtained by repeating the aforementioned procedures (Pan, X., Chen, K. and Chen, D, 2014).

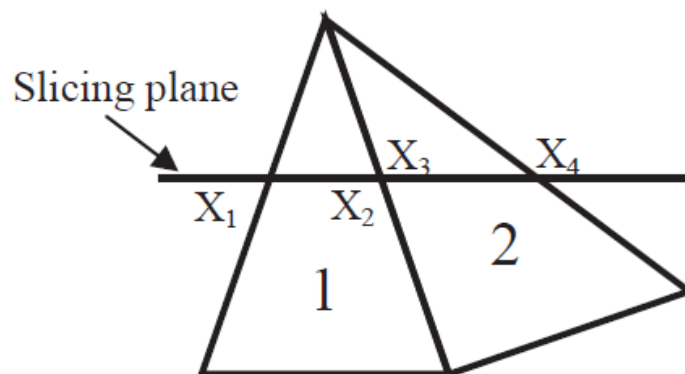


Figure 2.10: Intersection points' tracking (Pan, X., Chen, K. and Chen, D, 2014).

However, the subsequent ordered linked lists would not be produced if there were not a single contour loop on a layer because there would not be enough data to continue tracking after the first ordered linked list was created. In order to avoid repeating tracks when creating subsequent ordered linked



lists, intersection points that have previously been saved in the ordered linked list must be marked.

There are two contour loops on the layer, as seen in Figure 2.11. If you start with  $X1$  and follow each intersection point in turn to create the ordered linked list ( $X1\sim X8$ ), that is contour loop 1. The intersection locations  $X1\sim X8$  and those indicated by the associated "next" pointers should then be marked. In doing so, the value of "flag" will change to 1, signifying that the intersection point has been reached (Pan, X., Chen, K. and Chen, D, 2014). The indicated junction points will not be included when creating the subsequent ordered linked list.

In order to monitor and build the subsequent ordered linked list (such as  $X9 \sim X14$ ), which is contour loop 2, it will start at the first unmarked intersection point (such as  $X9$ ) whose value of "flag" is 0. The intersection locations  $X9 \sim X14$  will then be marked, together with the intersection points indicated by equivalent "next" pointers (flag = 1) (Pan, X., Chen, K. and Chen, D, 2014). The indicated junction points, including the intersection points marked previously, will not be tracked while creating the future ordered linked lists. Therefore, by precisely tracking with the intersection points' marking approach, all the ordered linked lists may be produced regardless of how many contour loops there are on a layer.

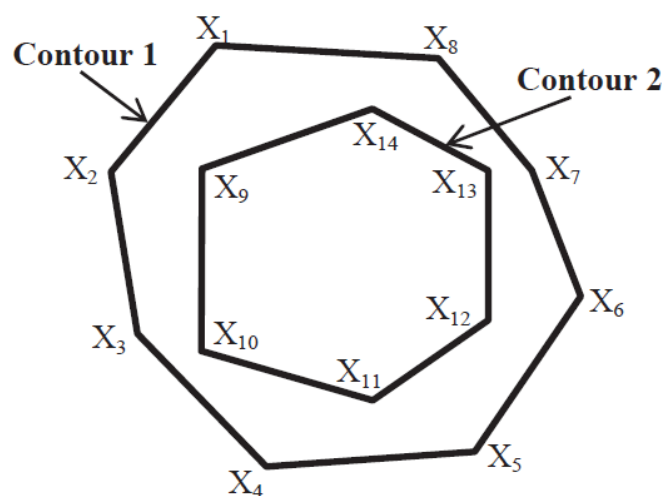


Figure 2.11: Two contour loops tracked by marking method (Pan, X., Chen, K. and Chen, D, 2014).

## 2.7 Infill of 3D Printing

The infill is significantly more dynamic and affects a part's strength, weight, structure, buoyancy, and other properties greatly. The type of infill applied for a product can be controlled by a number of parameters in 3D printing. The most significant of these factors are infill density and infill pattern.

The infill density specifies how much material is used on the interior of the print. A tougher object will result from a higher infill density due to the additional material inside the print. For prototypes with a visual purpose, an infill density of about 20% is adopted. For end-use components, higher infill densities may be used. For a prototype object where form or shape takes precedence over strength, an infill percentage of 18% to 20% may be adequate. For a weight-bearing device, like a bracket, that same infill % will, however, be wholly insufficient. In general, the infill percentage utilised during printing directly correlates with the strength of an FDM object. As an illustration, a part with 50% infill is roughly 25% stronger than a portion with 25% infill. Figure 2.12 shows an illustration of infill density from 0% to 100%.

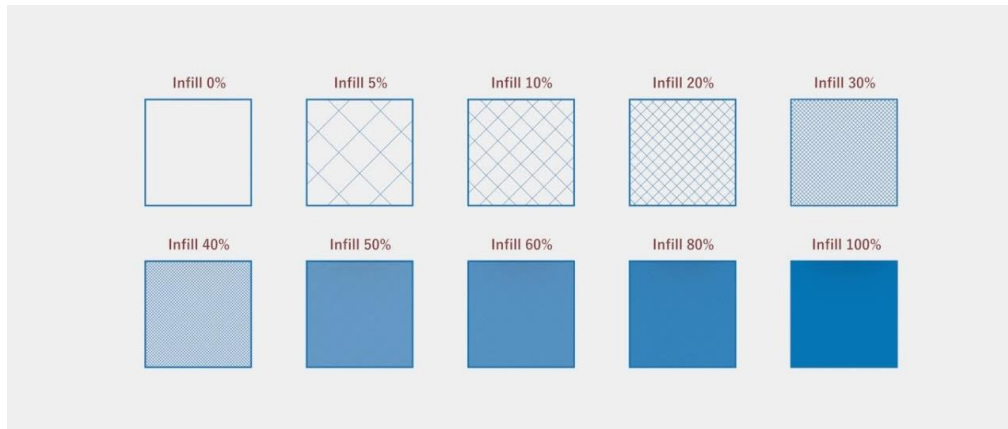


Figure 2.12: Infill Density.

For interior geometric (infill) generation, there are vast of distinct infill pattern available and proposed in the market as shown in Figure 2.13. Distinct infill patterns cause different performance for the printed objects. Manufacturers frequently aim to improve the strength-to-weight ratio of their parts in order to improve their performance (Gopsill, Shindler and Hicks, 2018). Hence, the infill pattern will be one of their favourite tuning parameters.

Different infill pattern will affect the material consumption and printing time as well.

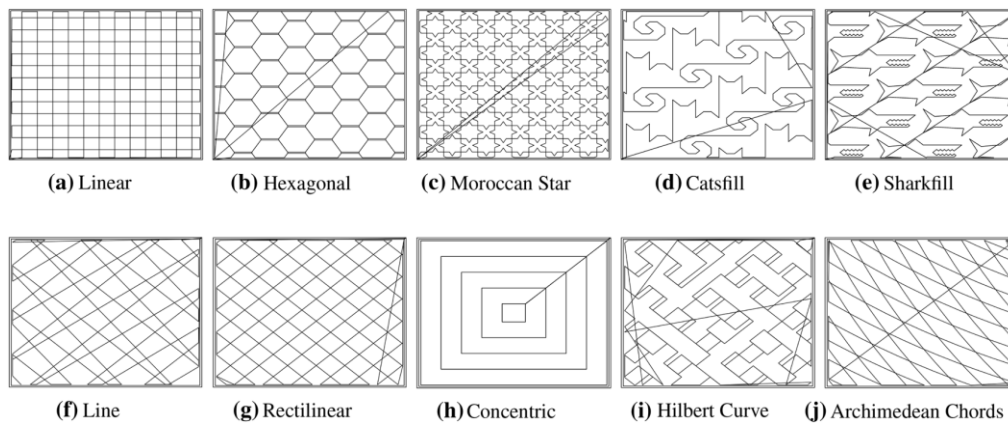


Figure 2.13: Infill pattern from Makerware (Gopsill, Shindler and Hicks, 2018).

## 2.8 3D Graphics Engine

In research of Chen (2010, p. 319), OpenGL is used as the 3D graphics engine for simulating the process of Computer Numerical Control (CNC) system. OpenGL engine provides visualization of tool-cutting for CNC simulation. Thus, OpenGL engine will be utilized for 3D visualization of 3D printer's gantry and nozzle movement and rendering of the layered 3D object. For producing computer-generated pictures, OpenGL is the most extensively used cross-platform API. Its ease of use as a programming framework enables even the most inexperienced OpenGL app developer to easily create programmes capable of generating sophisticated graphics with lighting effects, texture mapping, atmospheric effects, and anti-aliasing, among other features (Shreiner, 2001). The OpenGL rendering pipeline is the mechanism that takes the application's geometric and image primitives and rasterizes them to the framebuffer. The transformation phase and the rasterization phase are the two main components of the pipeline (Shreiner, 2001). Vertex transformations, lighting, and the creation of OpenGL fragments for the rasterization step are all performed by the transformation phase. Rasterization is in charge of colouring the framebuffer's suitable pixels. This phase comprises of depth testing, texture mapping, and alpha blending.

Every major operating system supports OpenGL, and it works with every major windowing system. It may also be called from most programming languages. It is completely unaffected by network protocols or topologies. Regardless of operating system or windowing system, all OpenGL programmes offer uniform visual display results on any OpenGL API-compliant hardware (Chen, 2010).

### 2.8.1 Rendering Line and Triangle

Only a few fundamental shapes, such as points, lines, and triangles, are supported by OpenGL. Curves and curved surfaces have no built-in support; they must be approximated by simpler shapes. Primitive forms are the most fundamental shapes. The vertices of an OpenGL primitive define it. A vertex is a point in three dimensions defined by its x, y, and z coordinates. OpenGL supports three type of line segments drawing which are *GL\_LINES*, *GL\_LINE\_STRIP*, and *GL\_LINE\_LOOP*. The primitive line drawing – *GL\_LINES* draws a line between two vertices. *GL\_LINE\_STRIP* connects all the vertices that stored sequentially in an array. Likewise, *GL\_LINE\_LOOP* connects all the vertices but the first and last vertex is connected unlike the *GL\_LINE\_STRIP*. Figure 2.14 shows the visualization of the primitives for line segment rendering.

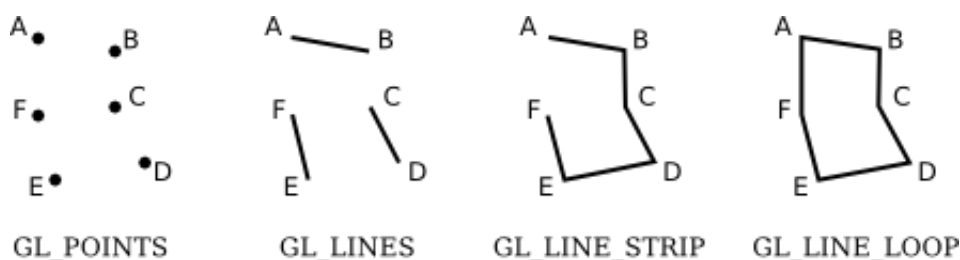


Figure 2.14 : Primitives for rendering lines.

Apart from that, there are three primitives for drawing triangle in OpenGL which are *GL\_TRIANGLES*, *GL\_TRIANGLE\_STRIP*, and *GL\_TRIANGLE\_FAN*. *GL\_TRIANGLES* draws a triangle with three vertices. *GL\_TRIANGLE\_STRIP* uses the first stated vertices to draw the first triangle. After that, using the next vertex and another two vertices from previous triangle to form an additional triangle that connect with the previous triangle.

While `GL_TRIANGLE_FAN` draws the first triangle using the first three vertices. After that, the next triangle will be drawn by connecting a new vertex with a vertex from pervious triangle and the first vertex. These primitives do not draw the points and lines shown in Figure 2.15, instead they fill in the interior of the triangle (green interiors shown in Figure 2.15).

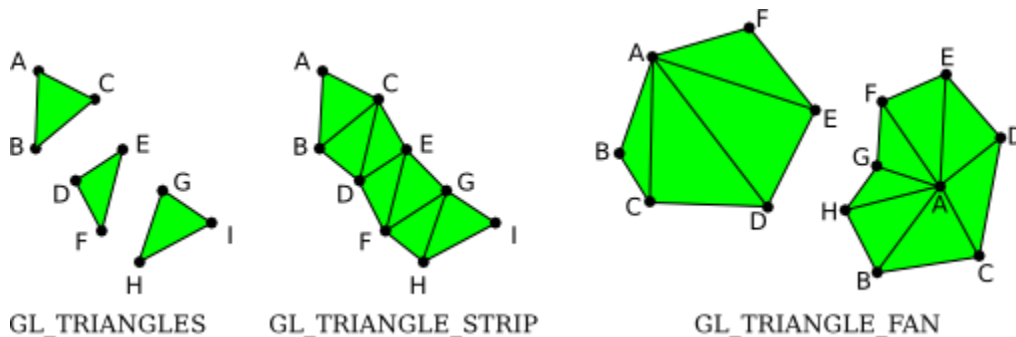


Figure 2.15: Primitives for drawing triangles.

### 2.8.2 Rendering Pipe

A cylinder is a 3D closed surface with two parallel circular bases at the ends and a curved surface connecting them (side). Similar to this, a prism is a 3D closed surface that is joined by flat surfaces from two parallel polygonal bases. Figure 2.16, Figure 2.17, Figure 2.18, Figure 2.19 illustrate cylinder-like geometry can be formed by splitting the base into more sectors. It is impossible to design a perfect circular base and curved side of the cylinder (slices). Therefore, by joining these sampled points together, it is conceptually building a prism. The geometry becomes more cylinder-like as the sample count rises (Ahn, 2019).

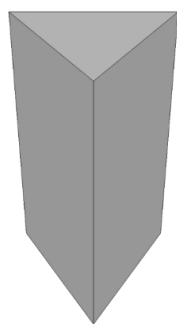


Figure 2.16: Triangular Prism

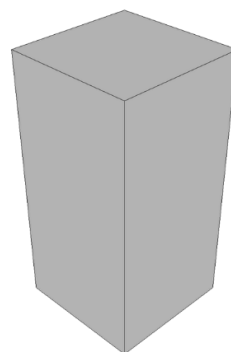


Figure 2.17: Rectangular Prism

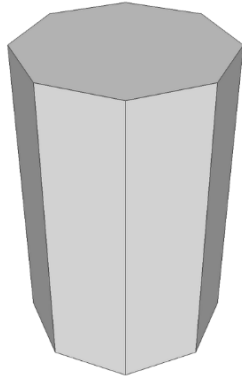
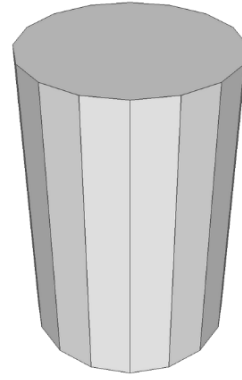


Figure 2.18: Octagonal Prism

Figure 2.19: Hex decagonal  
Prism

Assume that a cylinder has an origin-centred shape, a radius of  $r$ , and a height of  $h$ . The equation of a circle with the matching sector angle,  $\theta$  can be used to get the coordinates of any point  $(x, y, \text{or } z)$  on the cylinder.

$$x = r \times \cos\theta$$

$$y = r \times \sin\theta$$

$$z = \frac{h}{2} \text{ or } -\frac{h}{2}$$

Sector angles can range from 0 to 360 degrees. The following formulas can be used to get the sector angle for each step:

$$\theta = 2\pi \times \frac{\text{sectorStep}}{\text{sectorCount}}$$

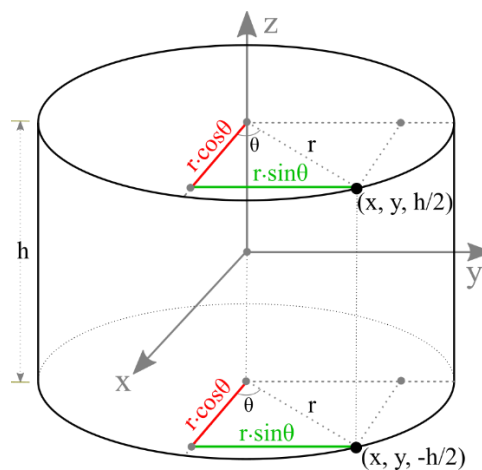


Figure 2.20: A vertex coordinate on a cylinder (Ahn, 2019).

Based on the study from Ahn (2019), in order to render cylinder in OpenGL, there are eight points required to be generated at each of the vertex that obtained from the slicer and infill generator to form a contour of a circle as shown in Figure 2.21. Multiple contours are required to compute in order to generate a cylinder-shaped path to illustrate filament. According to Ahn (2019), to obtain the next contour point,  $P_1'$ , the  $P_1$  is projected to the normal plane,  $\vec{n}$ , of  $Q_2$  where  $Q_1$  to  $Q_2$  and  $Q_2$  to  $Q_3$  intercepted as shown in Figure 2.22. The line equation is  $P_1 + tv_1$ , that passes through  $P_1$  while using the direction vector,  $v_1 = Q_2 - Q_1$ . Additionally, the point on the plane  $Q_2$  ( $x_2, y_2, z_2$ ) and the normal vector,  $\vec{n}$ , can be used to get the plane equation.

$$\vec{n} \cdot (x - x_2, y - y_2, z - z_2) = 0$$

Further, the normal vector is calculated by adding  $v_1$  and  $v_2$ :

$$v_1 = Q_2 - Q_1$$

$$v_2 = Q_3 - Q_2$$

$$\vec{n} = \vec{v}_1 + \vec{v}_2$$

Solving the linear system of the plane and line involves finding the intersection point  $P_1'$ .

$$\begin{cases} \text{Plane:} & \vec{n} \cdot (x - x_2, y - y_2, z - z_2) = 0 \\ \text{Line:} & P_1 + t\vec{v} \end{cases}$$

This step is repeated until all the contour point are generated. Once all the contour points are generated, the pipe can be rendered with the primitive drawing of `GL_TRIANGLE_STRIP`. Every contour will be connected with the drawing mode of `GL_TRIANGLE_STRIP` and allow the contour to present in solid body for graphical visualization. The expected result of the virtual cylinder-shaped filament is rendered to illustrate a 3D printing object as shown in Figure 2.23.

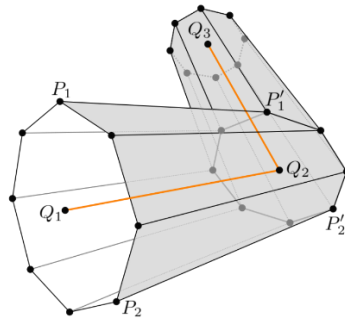


Figure 2.21: Extruding a pipe along a path  $Q_1$ - $Q_2$ - $Q_3$  (Ahn, 2019)

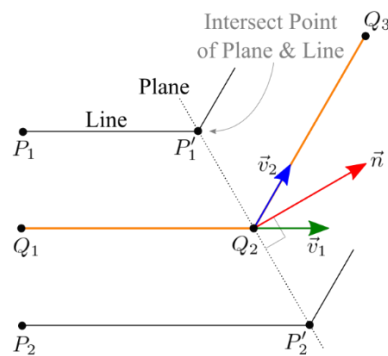


Figure 2.22: Cross-section view of extruding a pipe (Ahn, 2019)

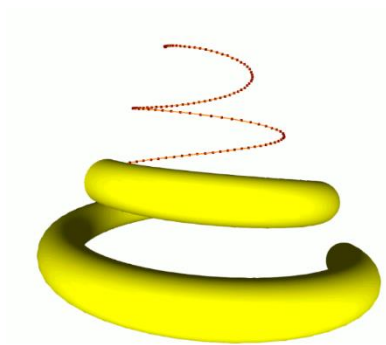


Figure 2.23: Pipe

## 2.9 Summary

Rapid prototyping (RP) process is divided into three phases. 3D CAD modelling and STL conversion are included in the first stage. The second stage is the aforementioned process planning, and the final stage is the physical part creation, which is fully dependent on the RP machine (Topçu, Taşcıoğlu and Ünver, 2011). Second phase of the 3D printing process is the main concern in this study. This study focuses on extracting the 3D model's



data from STL file and layering the data with slicing algorithm. The sliced data will be rendered using OpenGL for visualization on the pre-processing phase. The movement of the 3D printer will also be analysed and simulate in this simulator.

## CHAPTER 3

### METHODOLOGY AND WORK PLAN

#### 3.1 Introduction

Based on the literature studies in previous chapter, the approach and work strategy for this project will be detailed in this chapter. Additionally, those details and discussions will be used to guide the design standards and creation of the 3D printing simulator.

#### 3.2 Project Planning and Milestone

In FYP Part 1, the project schedules are mainly focusing on the study of the related research, journals, books and online resources. All the studies are carried out after the FYP title has been chosen and confirmed. Study on the background of the title is to determine and identify the scope, aims and objectives of the project. After scope of the project is confirmed, the articles of other researchers that related to the project scope are studied and analysed in order to obtain a better understanding of the methods on doing the similar project. To obtain literature reviews that are related to the project, keywords or index terms such as *additive manufacturing*, *simulator*, *simulation*, *rapid prototyping*, *slicing method*, *3D printing*, *visual simulation*, *STL format*, and *OpenGL* are used in the progress of searching. Moreover, other than researching on the ideas, the code that are related to the project have been tested and studied as well. The Gantt chart of FYP Part 1 is shown in Table 3.1.

In FYP Part 2, the project schedules are focusing on the design and development of the 3D printing simulator program based on the literature review that have studied in FYP Part 1. First of all, the schedule is to design and develop the STL file reader to extract the vertex and normal from a STL file in ASCII format. Next, a slicer which includes the intersection point detection and contour creation is planned to start once vertices are extracted successfully. After that, the design and development of infill generator that will perform either linear or line pattern will be started. The rendering function is planned to develop throughout the program design and development phase.

This is due to the reason that rendering function can be used to verify the feasibility of the STL file reader, slicer and infill generator. Afterwards, nozzle movement will be developed once infill generator is completed. Lastly, program testing will be carried out with different shapes of 3D model to evaluate the reliability of the 3D printing simulator program. The project schedule of FYP Part 2 is presented in a Gantt chart as shown in Table 3.2.

Table 3.1: Gantt Chart of FYP Part 1.

Gantt Chart

No.	Project Activities	Planned Completion Date	Gantt Chart														
			W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
1.	Project title selection	2022-01-24	█														
2.	Study On the Project Objective and Set the Project Scope	2022-02-19			█	█	█										
3.	Study On OpenGL	2022-03-18			█	█	█	█	█	█	█						
4.	Literature Review	2022-04-18						█	█	█	█	█	█	█	█		
5.	Studying on code	2022-04-15									█	█	█	█	█		
6.	Report Writing	2022-04-20										█	█	█	█		
7.	FYP Report Submission and FYP Presentation	2022-04-24														█	█

Table 3.2: Gantt Chart of FYP Part 2.

No.	Project Activities	Planned Completion Date	Gantt Chart														
			W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	
1.	Develop the slicing algorithm.	2022-07-05	█	█	█	█											
2.	Develop the infill generator	2022-07-30				█	█	█	█								
3.	Develop the movement of nozzle and filament	2022-08-21				█	█	█	█	█	█						
4.	Prepare the report and presentation	2022-09-04				█	█	█	█	█	█	█	█				

### 3.3 Program Flow of 3D Printer Simulator

The simulated type of 3D printer will be an FDM 3D printer with a static platform and nozzle movement along the X, Y, and Z axes, according to the literature review. This selection was made in order to streamline the software and lighten the burden on the GPU and programming. Moving the platform will require additional functions, which means the code will have more lines,

which will take longer to debug. FDM printers are also the most popular 3D printers used for 3D model prototyping.

According to the literature review, there are several features that a 3D printer simulator must possess. First of all, the starting feature is STL file reader. In this project, the STL file reader will get a STL file in ASCII format from user and extract all the facet's vertices and normal vector. They are stored into their respective array. The vertices will be centred in the origin (0, 0, 0) before passing to slicer program. Next, the extracted triangular facet's vertices will be further processed in a slicer program. The slicer program functions to cut the triangular facet into layer by layer. The increment of a layer to the next slicing layer, so called layer gap, is decided by the user. After that, a function of contour creation will be triggered with the completion of slicing all the triangular facets. Contour creation is to arrange all the sliced vertices accordingly to form a continuous chain of the sliced vertex. This arrangement of sliced vertex is essential as the vertices that are generated from the slicer are irregularly dispensed. This chaotic vertex structure may lead to an inefficient printing path. Afterward, the ordered vertices will then be utilized in the infill creation. The patterns of the infill that available in this project are linear and line pattern. The infill vertex will be stored in a separate array than the sliced vertex array to reduce the size required for the array. Heretofore, sliced vertex and the infill of the 3D printing object are generated.

After all the interested vertices from contour creation algorithm and infill generator algorithm are stored in their respective array, the next steps will be the presentation of the vertices that obtained. First of all, a window will be created with the GLUT library for the OpenGL graphical rendering. The window will be created with the size of 1200 pixels x 800 pixels and the window is named "3D Printing Simulator". After a window is created, an initialized 3D space will be generated with the default setting such as the type of projection, the position of the camera, the position and colour of the lighting. Initially, the appearance of the imported object will be rendered after the space is formed. After this, the keyboard feedback will be read to execute their respective tasks. The simulation of printing process can be started by pressing 'p' while to switch back to solid mode by pressing 'o'. When the printing

process is started, the nozzle will move from the home position to the printing position. The object will be printed layer by layer once the nozzle has reached the printing position. A whole view of 3D-layer object will be displayed once the printing is accomplished. The program flow diagram is illustrated in Figure 3.1,

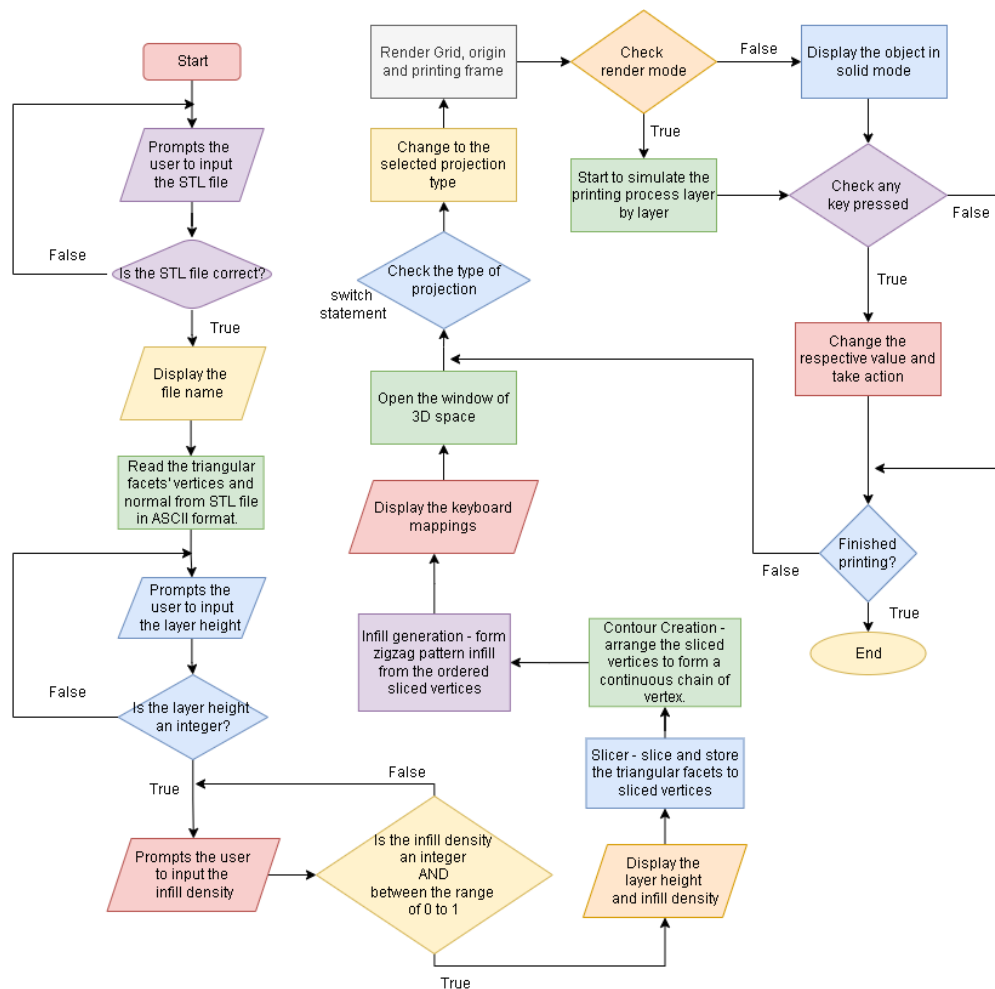
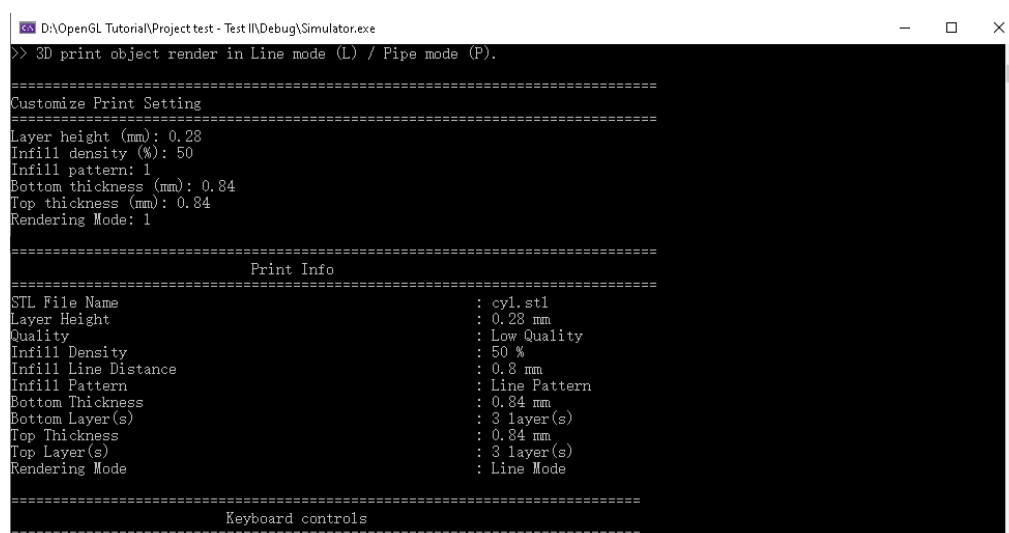


Figure 3.1: Flowchart of 3D printing simulation program.

### 3.4 User Input

In this 3D printing simulator program, a command window will prompt user to input a STL file in ASCII format. Once users enter a STL file name that has been saved in the solution folder and is opened successfully, it will prompt from user the layer height for the slicing algorithm to slice the model. The input must be an integer, otherwise, the simulator will display an error message and ask for another input if users enter a non-integer input. Next, user

is requested to enter the desire infill percentage which range from 0 to 100 and follow up with infill pattern. An error validation for input will be applied. Afterwards, users will be informed to input the desire top and bottom thickness of the object. An integer answer is required for the top and bottom thickness. Once all the input is valid, the entered value will be displayed on the window for user to check. A sample of the window prompt is shown in Figure 3.2.



```

D:\OpenGL Tutorial\Project test - Test II\Debug\Simulator.exe
>> 3D print object render in Line mode (L) / Pipe mode (P).
=====
Customize Print Setting
=====
Layer height (mm): 0.28
Infill density (%): 50
Infill pattern: 1
Bottom thickness (mm): 0.84
Top thickness (mm): 0.84
Rendering Mode: 1
=====
Print Info
=====
STL File Name      : cyl.stl
Layer Height       : 0.28 mm
Quality            : Low Quality
Infill Density     : 50 %
Infill Line Distance : 0.8 mm
Infill Pattern     : Line Pattern
Bottom Thickness   : 0.84 mm
Bottom Layer(s)    : 3 layer(s)
Top Thickness      : 0.84 mm
Top Layer(s)       : 3 layer(s)
Rendering Mode     : Line Mode
=====
Keyboard controls
=====

```

Figure 3.2: A window prompt that gets user input and shows the printing info.

### 3.5 Workspace

A printing volume of a width of 220 mm, length of 220 mm, and height of 250 mm is designed to simulate the printing volume of a CREALITY FDM 3D printer – Ender 3. The designed printing area are built with grid for better visualization of the model and the printing process. The grid is made of 23 lines on each of the X and Y axis with a gap of 10 mm between each line. Apart from that, origin of the nozzle is set at the left front corner of the workspace which is located at the coordinate of (110, 110, 0) in the 3D space. A red colour dot will be rendered to imply the home position of the nozzle. Besides that, X, Y and Z direction from the origin will be rendered in yellow, green, and blue line, respectively. Moreover, a frame will be rendered to illustrate the printing volume of the selected FDM 3D printer. A rectangular frame will be created with 8 vertices which are located at (110, 110, 0), (-110,

110, 0), (110, -110, 0), (-110, -110, 0), (110, 110, 250), (-110, 110, 250), (110, -110, 250), (-110, -110, 250).

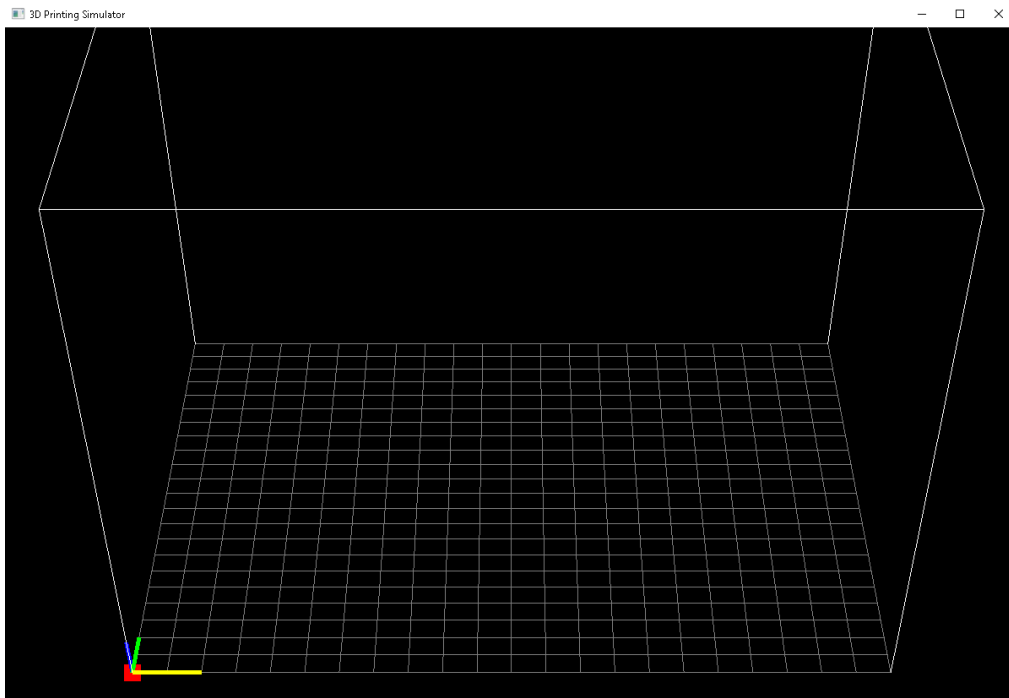


Figure 3.3: A 3D space created for the simulation of 3D printing.

Furthermore, the position of the camera can be set through the function of *gluLookat()*, the default camera position will be at position of (0, 500, 600). There are several options available for different types of projection which are isometric view and orthographic view. The camera position of the isometric view will be located at (500, 400, 600). While the camera position for the orthographic projection of top, right side, left side and front view are (10, 0, 700), (700, 0, 300), (-700, 0, 300) and (0, 750, 400) respectively. The keyboard button to select the respective view of isometric, top, right side, left side and front are '1', '2', '3', '4' and '5'. Last but not least, the lighting of the space is located at (50, 50, 50) and the colour of the lighting is purple.

### 3.6 Vertex Extractor of STL file in ACSII format

The industry standard file type for 3D printing is STL (Standard Triangle Language). The surfaces of a solid model are represented by a succession of triangles. All modern CAD (Computer Aided Design) software supports

exporting native file formats into STL. A list of facet data makes up a STL file. A unit normal (a line perpendicular to the triangle with a length of 1.0) and three vertices uniquely identify each facet (corners). Each facet has a total of 12 numbers because the normal and each vertex are each given by three coordinates. Figure 3.4 shows an example of STL file which is converted from CAD software. The surface of the model has been triangulated which consists of multiple triangular facets and each of them made of three vertices and one facet normal.



Figure 3.4: Graphical visualization of a STL file.

For STL ASCII format, the syntax is as shown in Figure 2.5. In this project, STL format of ASCII will be read, and the vertices data of a 3D object will be extracted into an array to store the triangular facets' vertices. To read the STL file in C++, '*ifstream*' is used and using a *WHILE* loop to do the STL file error validation. After the file is imported, string compare – '*strcmp*' will be used to determine the line that contain the information of interest. The information that we are interested will be the "*vertex*" and "*normal*" of the triangulated facets in a STL file. Hence, facets' vertex and normal will be stored if the import phrase from the file matches with the words – "*vertex*" and "*normal*" by using '*strcmp*'. A *FOR* loop function is constructed to read the STL file line by line and compare with '*vertex*' and '*normal*'. If the '*strcmp*' returns 0 then the data of that line will be stored into their respective array. If the '*strcmp*' returns a non-zero index, then next line will be read and checked until all the data are extracted. By comparing the import phrase from STL file with a word – "*endsolid*", we can examine if a STL file has been gone through.



The flow diagram of the STL reader is shown in Figure 3.5. With the STL file reader, all the vertices of the triangular facets can be extracted as shown in Figure 3.6. The example illustrated in Figure 3.6 is a cylinder and the yellow points are the vertices of the triangular facets that extracted from the STL file.

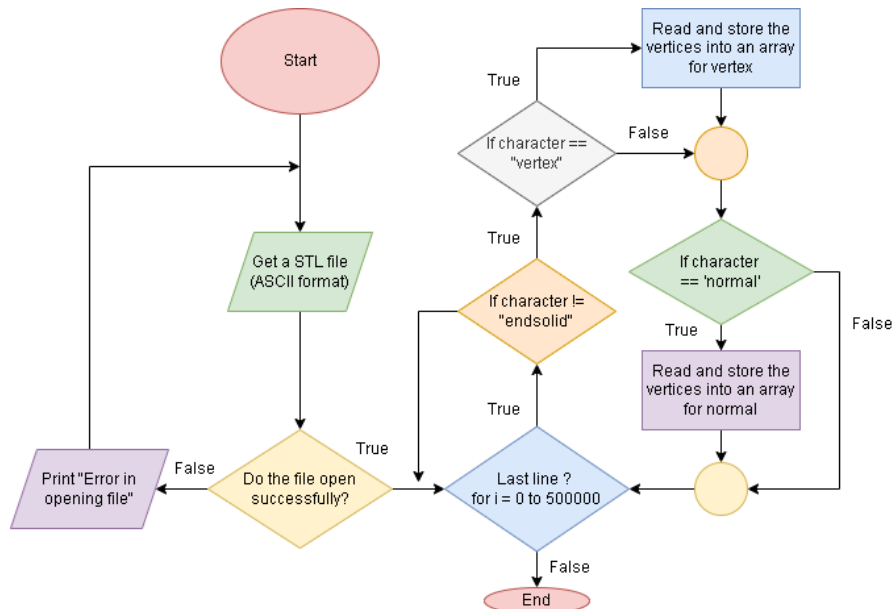


Figure 3.5: Flowchart of STL file reader.

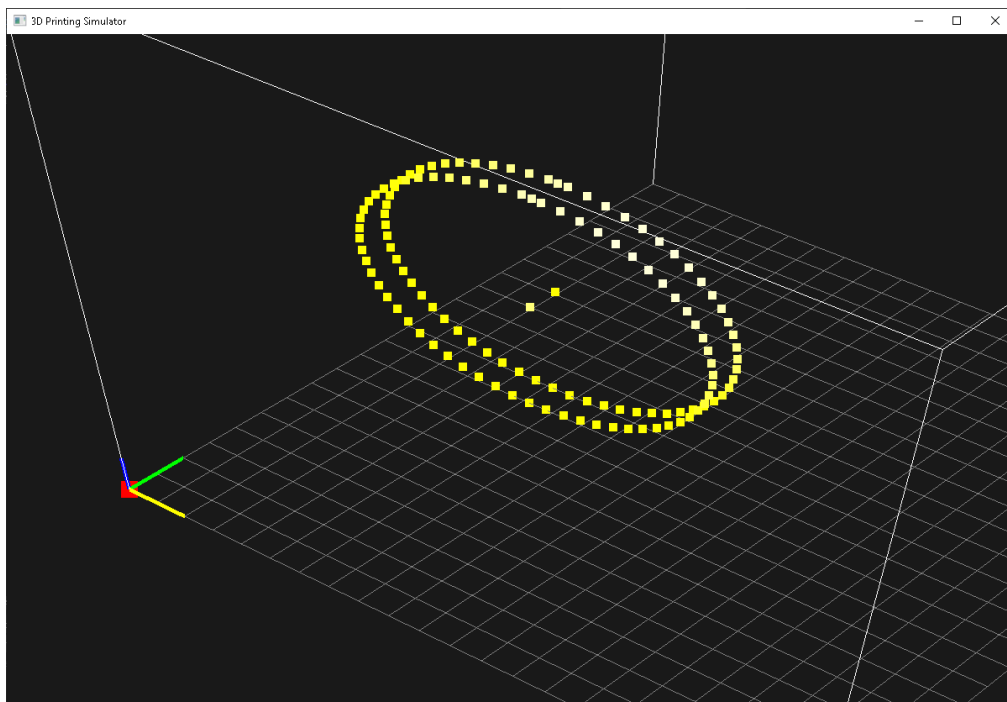


Figure 3.6: Presentation of imported vertices.

Before passing the vertices obtained to the slicer, all the vertices are processed to centre the vertices to the origin. This is to move the imported 3D object to the origin of the 3D space. To centre the vertices, first is to obtain the largest and smallest  $X$  and  $Y$  in the model vertices. After that, the summation of largest  $X$  and smallest  $X$  will be divided by 2 to compute the  $X$  coordinate of the middle point of model. This is applied to compute  $Y$  coordinate of the middle point as well.

$$\left. \begin{aligned} \text{middlepoint}(X) &= (\text{largest}X - \text{smallest}X)/2 \\ \text{middlepoint}(Y) &= (\text{largest}Y - \text{smallest}Y)/2 \end{aligned} \right\} \quad (3.1)$$

Next, the offset of the middle point of the model to the origin will be calculated through equation (3.2) and add to every vertex to translate them to the origin as shown in equation (3.3).

$$\left. \begin{aligned} \text{offset}(X) &= \text{origin } X - \text{middle point}(X) \\ \text{offset}(Y) &= \text{origin } Y - \text{middle point}(Y) \end{aligned} \right\} \quad (3.2)$$

$$\text{vertex} = \langle X + \text{offset}(X), \quad Y + \text{offset}(Y) \rangle \quad (3.3)$$

The flowchart of the centring 3D model algorithm is shown in Figure 3.7. The position of the 3D model before and after centring is shown in Figure 3.8 and Figure 3.9.

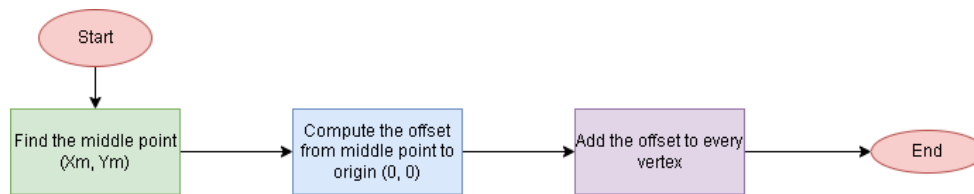


Figure 3.7: Flowchart of centring a 3D model in 3D space.

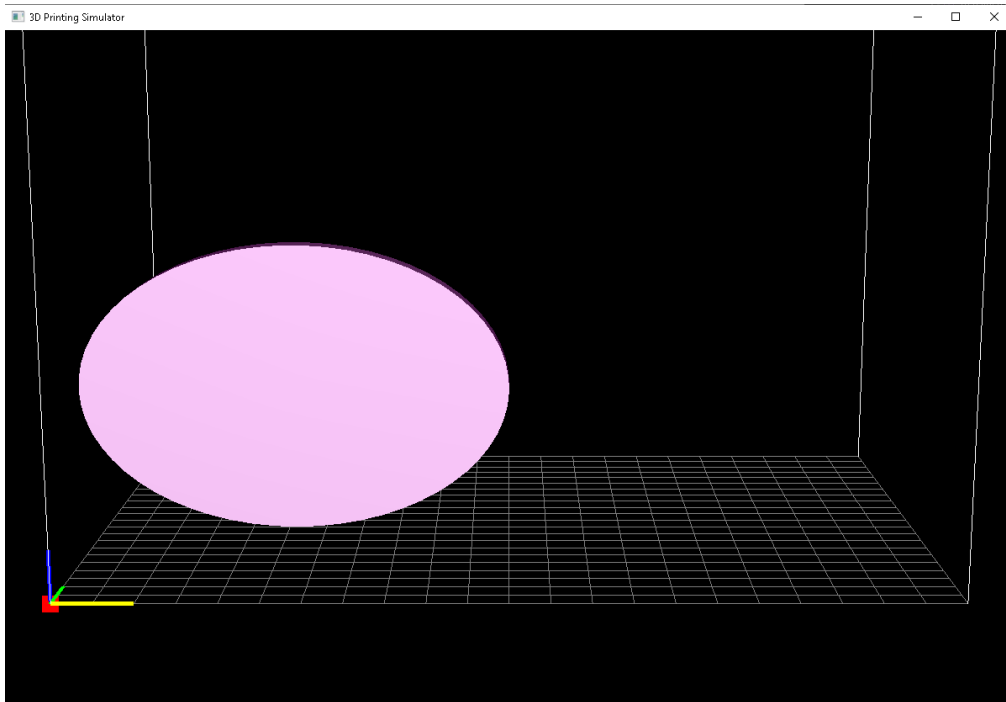


Figure 3.8: Position of 3D model before centring.

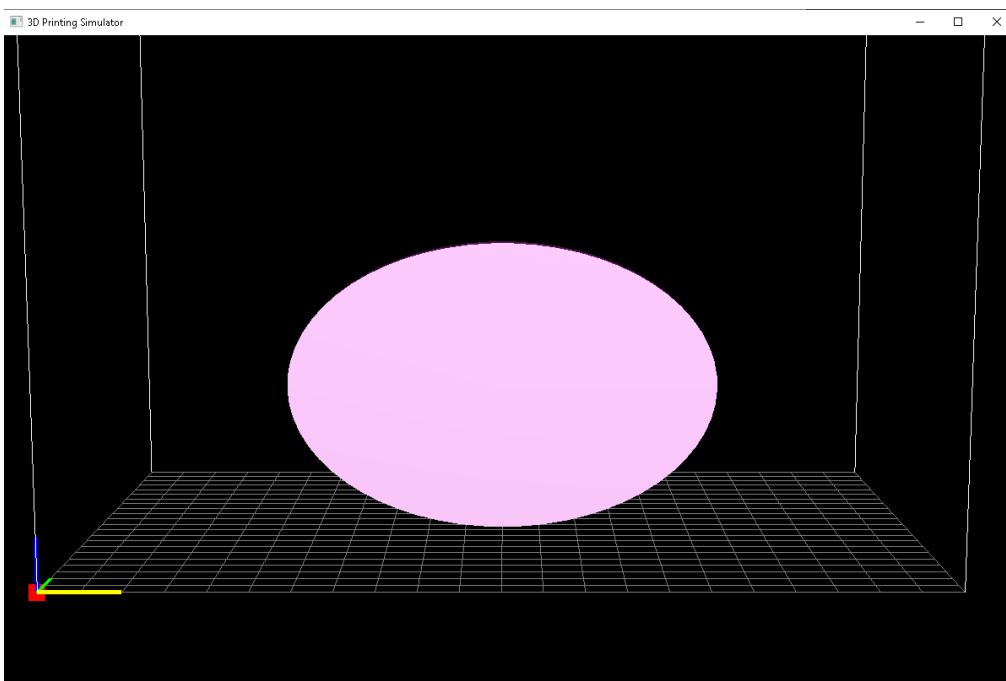


Figure 3.9: Position of 3D model after centring.

### 3.7 Slicer

Based on the literature review, the slicing method used in this project is basic slicing method. A variable,  $z$  represents a slicing plane which will be used for intersection detection. The  $z$  value will compare with each of the three vertices

in each triangle to determine the coordinate of intersecting points. There are five ways for slicing plane to intersect with a triangle as shown in Figure 3.10. These five cases will be applied to slice an 3D object into layer as shown in Figure 3.11.

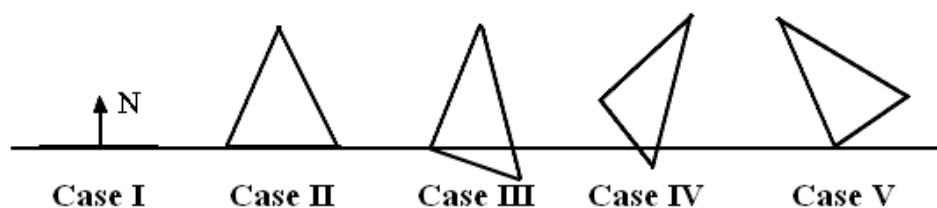


Figure 3.10: Possible intersection cases (Topçu, Taşcıoğlu and Ünver, 2011).

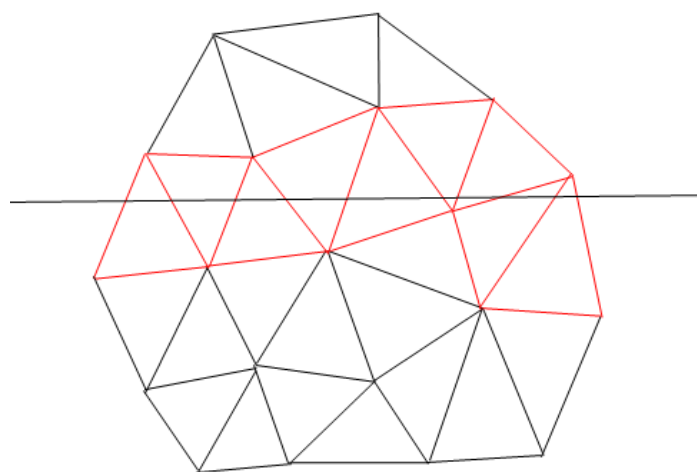


Figure 3.11: An illustration of slicing triangular facets.

After all the facet vertices are extracted from the STL file. The triangular facets will be categorized into their respective case type which as shown in Figure 3.10. The process will be repeated for each slicing plane. According to Figure 3.10, a triangular will be considered as Case I if all its vertices intercept with the slicing plane. Triangular facet that intercepts any two vertices with the slicing plane is categorized under Case II. Next, if there is only one vertex intercepts with the slicing plane, it will be considered as Case III. While the triangle will be deemed as Case IV if any vertex is opposite side with the other two vertices. For instance, if first vertex is above and the other two vertices is below the slicing plane then this triangle will be categorized under Case IV. Lastly, triangle that is not categorized under any Case from I to IV will be considered as Case V. The flowchart for this sorting process is shown in Figure 3.12.

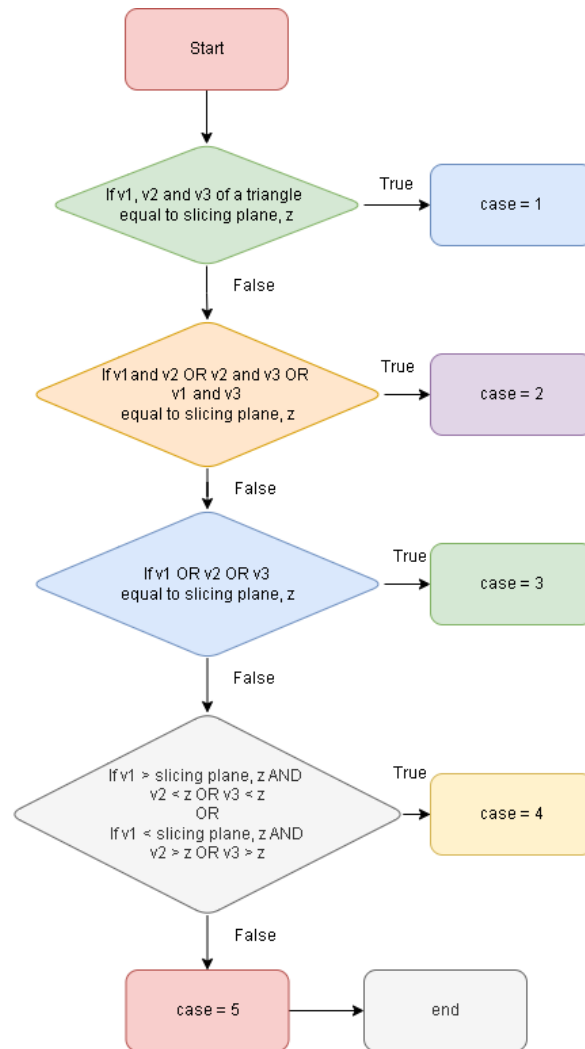


Figure 3.12: Flowchart of slicing algorithm.

### 3.7.1 Case I

Case I is the case when a triangle parallel to the slicing plane and all the facet's vertices of that particular triangle intercept with the slicing plane. According to the literature review, the three vertices with their x, y and z coordinates will be stored in an array as sliced vertex. The flowchart of the implementation is shown in Figure 3.13. However, the vertices that are categorized under Case I will not be stored as shell data in this project. This is because vertices under Case I are considered as redundant point when encounter circle facet. The redundant point will be the middle vertex for a circle. An example of middle points of a circle that are considered as redundant are circled in Figure 3.14. Henceforth, Case I is excluded from storing any vertices to avoid the redundant vertices.

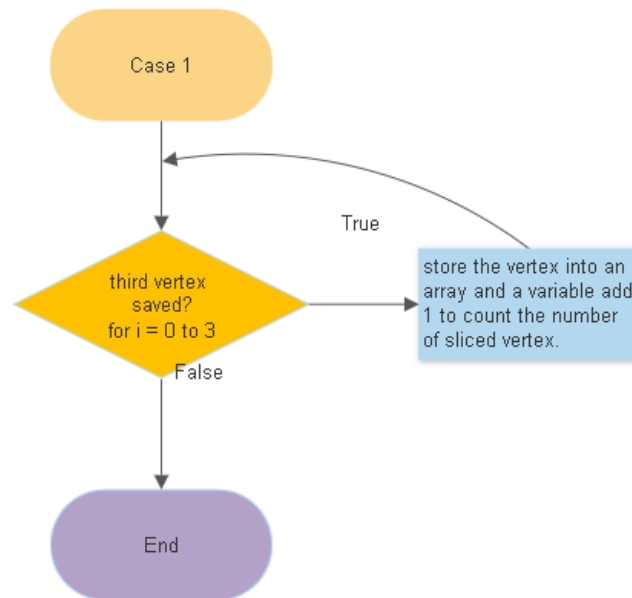


Figure 3.13: Flowchart of Case I.

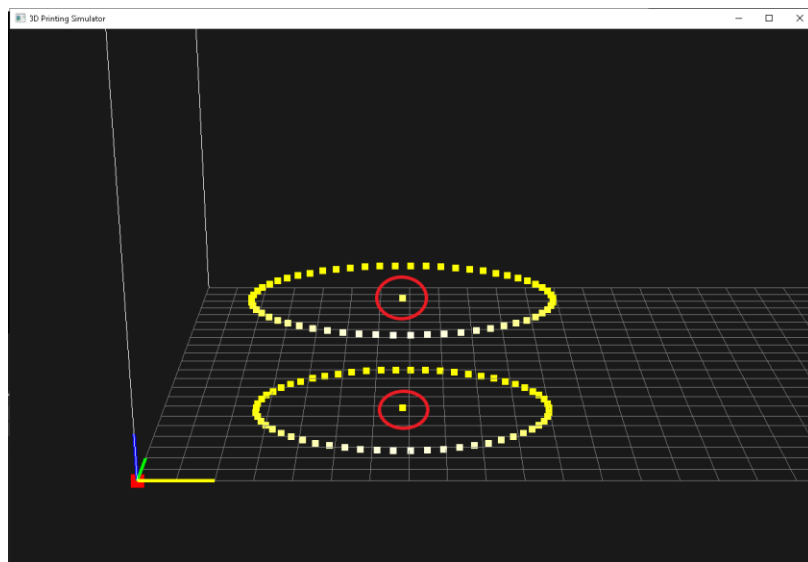


Figure 3.14: Redundant points in a circle

### 3.7.2 Case II

For Case II, two vertices that are intersecting with the slicing plane will be stored as sliced vertices. Any vertex fulfils a condition of its z-coordinate has the similar value with layer height will be stored. Each vertex of a triangular facet is checked by using a *FOR* loop. X, Y and Z coordinates of a qualified vertex will be stored into an array.

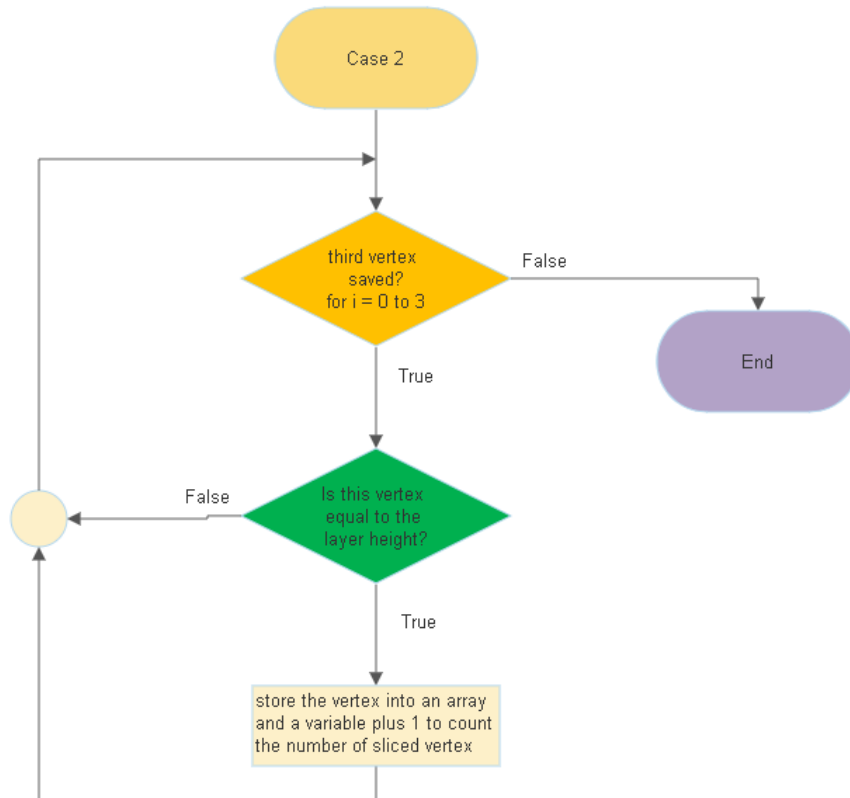


Figure 3.15: Case II.

### 3.7.3 Case III

For Case III, a vertex of the triangle is touching with the plane and other intersection point will be computed using another two vertices. The intersection point can be computed using a vector equation (3.4) where  $r_0$  is an initial vertex,  $r$  is a terminal vertex,  $v$  is a direction vector and  $t$  is a scalar.

$$r = r_0 + vt \quad (3.4)$$

Let  $r_0 = \langle x_0, y_0, z_0 \rangle$ ,  $r = \langle x, y, z \rangle$  and  $v = \langle x - x_0, y - y_0, z - z_0 \rangle$ . With these values, parametric equation of the line can be formed to find the interception points,  $\langle x_i, y_i, z_i \rangle$ . Next, the scalar,  $t = \frac{z_i - z_0}{z - z_0}$  with the given  $z_i$  value as the  $z_i$  value will be the layer height that obtained from the user. After the scalar,  $t$  has been computed, the intersection point can be determined by using the parametric equation which shown in equation (3.5).

$$x_i = x_0 + (x - x_0)t; y_i = y_0 + (y - y_0)t; z_i = \text{layer height} \quad (3.5)$$

An example of case III is shown in Figure 3.16. In the example, point B is the vertex that intercept with the slicing plane ( $z = 1$ ) while point A and point C are the vertices that will be used to determine the intersection point D. Then, both point B and point D will be stored in the array as sliced vertices and the variable used to count the number of sliced points in the array will plus 2 as two vertices are added. Besides the case III shown in Figure 3.10, there are some other possible cases III which are illustrated in Figure 3.17. These types of case III are not desirable by the slicer in this project because they consist of an intersection point with the slicing plane only. To exclude unwanted case III, the two vertices (denoted as  $v1$  and  $v2$ ) that are not intercept with the slicing plane will be used to compare with the slicing plane. If  $v1$  is above the slicing plane and  $v2$  is below the slicing plane or vice versa, indicates that the slicing plane is within the line formed by  $v1$  and  $v2$ . Thus, both of the unwanted case III shown can be excluded from the algorithm by an *IF* statement. The flowchart of the Case III is shown in Figure 3.18.

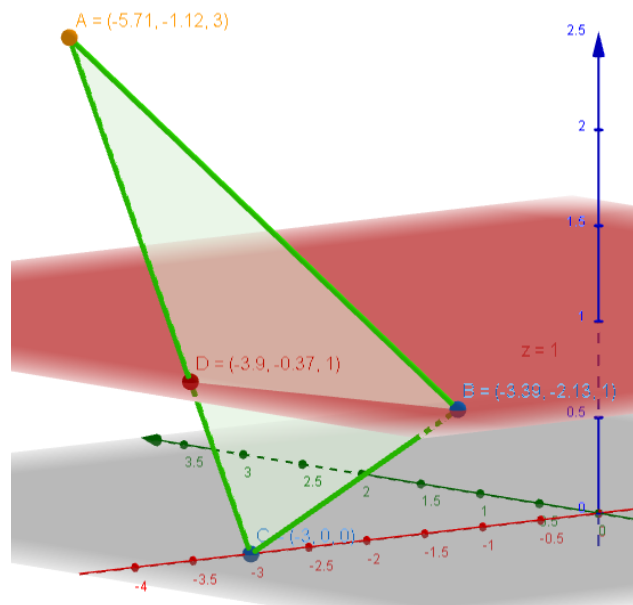


Figure 3.16: An illustration of slicing a triangle under Case III.



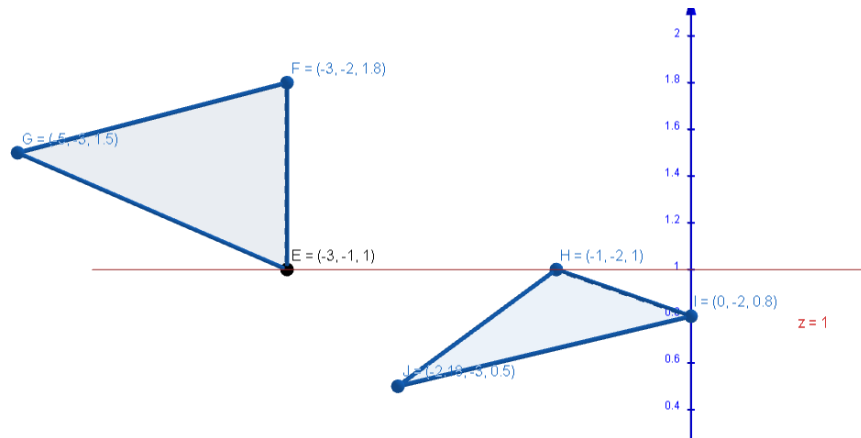


Figure 3.17: Unwanted type of triangles.

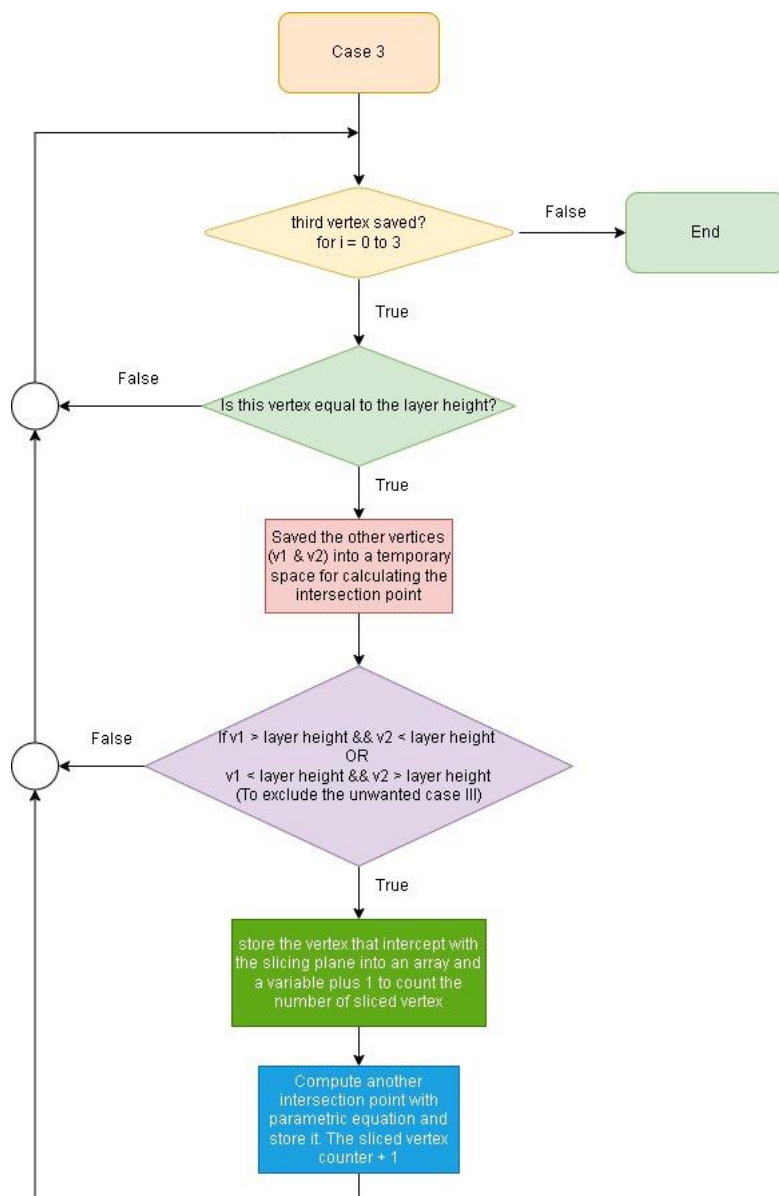


Figure 3.18: Flowchart of Case III.

### 3.7.4 Case IV

Case IV is similar with case III which required to use the equation (3.4) and (3.5) to compute intersection point. Instead, in case IV, both intersection points need to be calculated. Let Figure 3.19 as an example, the vertex located below the slicing plane is denoted as unique vertex, A. Then take A as a centre point to form two straight lines which are line AC and line AB. The intersection points will be calculated with the known value of  $z$  of the slicing plane and the coordinates of the two vertices (A&C and A&B) by using the equation (3.4) and (3.5). The flowchart of case IV is shown in Figure 3.20.

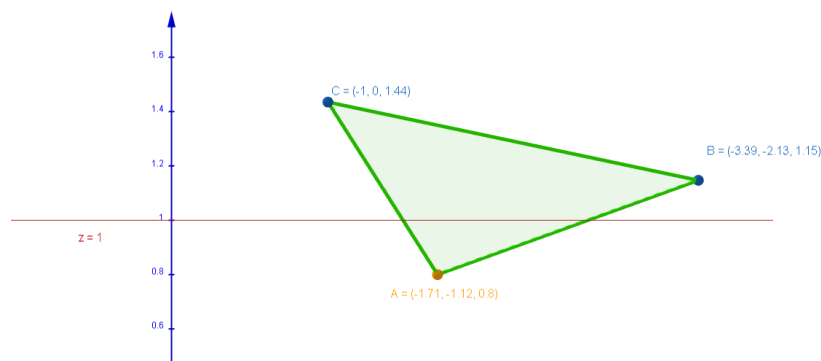


Figure 3.19: An illustration of triangle under Case IV.

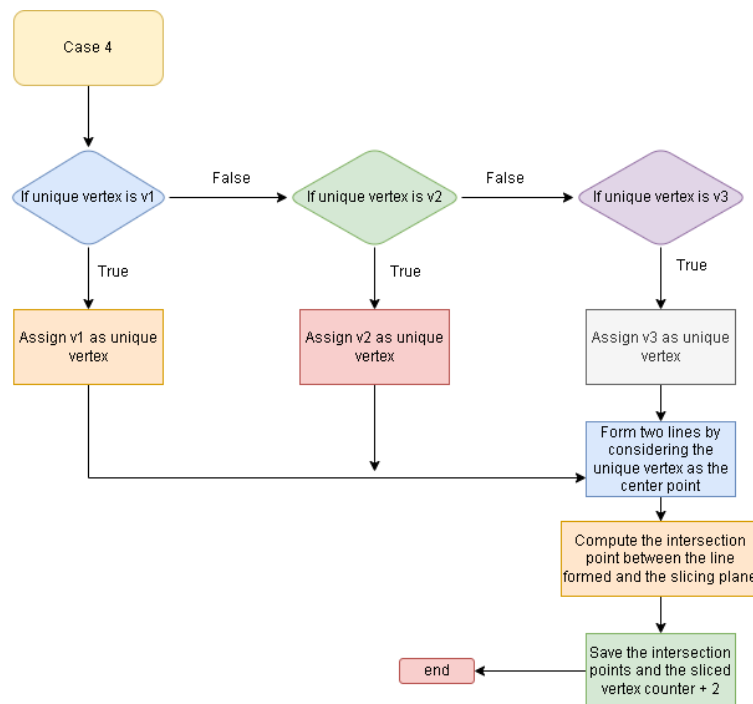


Figure 3.20: Flowchart of Case IV.

### 3.8 Contour Creation

The output of the slicer algorithm in previous section will be the input of the contour creation that is used to make slices of data in lines with the supplied  $z$  values (layer height). The output of the slicer algorithm is a bunch of disordered sliced data which is undesired for the printing process as it may lead to inefficient printing path. Thus, the sliced data is required to be processed to generate a list of linked intersection points to form contour.

First and foremost, the contour creation has to be able to sort the list of sliced data obtained into a set of starting point and end point. Since the slicer will get two intersection points from each case, there will be even number of sliced data output from the slicer to the contour creation function. With this information, the sliced data that stored as even number (eg. 0, 2, 4, ...) in the sliced data array will be deemed as the starting point of a line while odd number (eg. 1, 3, 5, ...) will be considered as the end point of a line. Each starting point and end point will be stored in their respective array for the usage of contour detection. The number of starting point and end point will be distributed evenly because the total number of sliced data is even number.

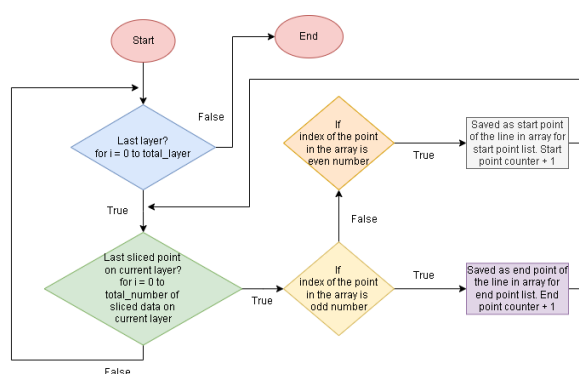


Figure 3.21: Flowchart of assigning start and end point.

After all the sliced data has been obtained from the slicing algorithm, they are arranged to form contour loop. The contour creation is referred to the intersecting points' tracking method which has been discussed in the literature review.

First of all, the first point of the sliced data on each layer is assigned as the starting point to construct an ordered sliced data list. The end point that

lies on the similar line with this starting point will be used to search for the next starting point which coincident with it. The starting point and end point of the line will be stored and the “next” starting point that has been found will be treated as the new starting point and its end point will be used to search for the next starting point.

In case that there are no starting point matches with the end point, the end point will search through the end point list to find any end point coincident with it. If there is an end point matches with the end point, then the end point that has been found will switch with its starting point. For instance,  $E_1$  is the end point of the searcher, if  $E_2$  matches with  $E_1$ , the position of  $E_2$ 's starting point –  $S_2$  will interchange with  $E_2$ . Hence,  $E_2$  will be  $S_2$  and  $S_2$  will be  $E_2$ . The starting point and end point will then be stored in the ordered list. The flag of the visited starting and end point will be marked as “1” to indicate that the point has been visited.

The process is repeated until no point is matched to the end point or all the sliced data on the particular layer has been visited (all flag is “1”). In the case of no point is matched to the end point, the first point that has yet to be visited in the sliced data will be assigned as the new starting point for the new contour forming process. The process will repeat in every layer until all layers are examined.

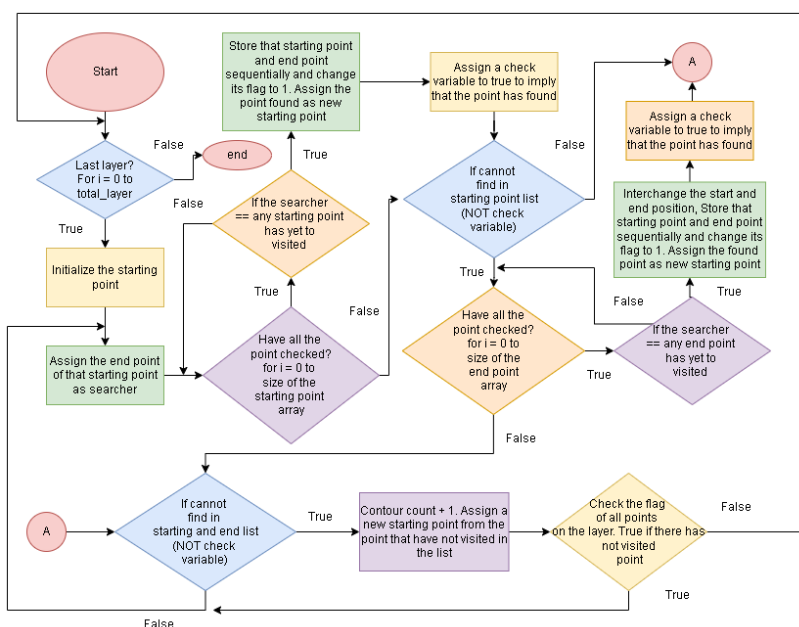


Figure 3.22: Flowchart of Contour Creation program.

### 3.9 Infill

A 3D print's interior is referred to as the infill, while its exterior is referred to as the shell. The infill can be printed in a range of distinct geometries and densities ranging from 0% (hollow) to 100% (solid). The interior geometry and density of a 3D print affects a part's strength, weight, structure, buoyancy, and other properties greatly. Henceforth, these parameters are implemented in this 3D printing simulator.

#### 3.9.1 Infill Density

Infill density is a parameter to adjust the percentage of distance between each infill lines. The maximum distance between each infill lines is set to be 40.0 mm. The gap between each infill lines is calculated by the default setting of maximum distance between each infill lines and the infill density. The distance between the printed infill lines is computed by the equation (3.6). The maximum distance between lines will be a constant value – 40 mm whereas infill density is a user input parameter which range from 0% to 100%. The illustration of the equation is shown in Table 3.3.

$$\text{Infill gap between lines, } d = \frac{\text{Maximum distance between lines}}{\text{Infill Density (\%)}} \quad (3.6)$$

Table 3.3: The infill gap, d with respective infill percentage.

<b>Infill Percentage (%)</b>	<b>Infill gap between lines, d</b>
<b>10</b>	$(40 \text{ mm})/10 = 4 \text{ mm}$
<b>20</b>	$(40 \text{ mm})/20 = 2 \text{ mm}$
<b>30</b>	$(40 \text{ mm})/30 = 1.33 \text{ mm}$
<b>40</b>	$(40 \text{ mm})/40 = 1 \text{ mm}$
<b>50</b>	$(40 \text{ mm})/50 = 0.8 \text{ mm}$
<b>60</b>	$(40 \text{ mm})/60 = 0.67 \text{ mm}$
<b>70</b>	$(40 \text{ mm})/70 = 0.57 \text{ mm}$
<b>80</b>	$(40 \text{ mm})/80 = 0.50 \text{ mm}$
<b>90</b>	$(40 \text{ mm})/90 = 0.44 \text{ mm}$
<b>100</b>	$(40 \text{ mm})/100 = 0.40 \text{ mm}$

### 3.9.2 Infill Pattern

In Chapter 2.7, the Figure 2.13 has shown vast pattern of interior geometric for additive manufacturing. However, this project cannot cover the simulation of all the interior geometric. Henceforth, the project will narrow the variations of the interior geometric becomes two types of patterns which are (a) linear pattern and (f) line pattern that shown in Figure 2.13. These infill patterns can be implemented by controlling the sequence of storing the infill points into the infill array. The behaviour of infill vertices assignment into the array will be discussed in Chapter 3.9.2.2 and Chapter 3.9.2.3. The line and linear patterns are designed to swap direction on alternate layers. The purpose of swapping direction on alternate layers is to evenly fill the interior of a 3D print body. This will provide a more equal distributions of strength over X-axis and Y-axis direction. Prior to sequencing the infill points to respective pattern, intersection points detection for infill vertex are constructed to generate infill vertex.

#### 3.9.2.1 Infill Vertex Detection

The infill vertex is the intersection point between the infill line and the ordered shell data that stated in Infill Density and Contour Creation respectively. Figure 3.23 illustrates an example with a circle that is formed by 12 sides polygon and the wall vertices are ordered in anti-clockwise direction. In the ordered list, the polygon's vertices are listed from the index of 0 to 11 of the arrays. For illustration, the polygon's vertices – {1, 2, 3, ..., 12} that shown in Figure 3.23 will be  $array[i] = \langle x_0, y_0, z_0 \rangle$ ,  $array[i+1] = \langle x_2, y_2, z_2 \rangle$ ,  $array[i+2] = \langle x_3, y_3, z_3 \rangle$ , ...,  $array[i] = \langle x_{n-1}, y_{n-1}, z_{n-1} \rangle$  in the ordered list where  $i = \{0, 1, 2, \dots, n - 1\}$  and  $n$  is the number of vertices. These vertices will be used to form a continuous line such that  $array[0]$  and  $array[1]$  will form a line,  $array[1]$  will form a line with  $array[2]$  and so on. The intersection between these lines and the infill line will be the infill vertex. The infill line will be incremented from the smallest  $x$  value to the largest  $x$  value in the ordered shell list for every iteration in a *FOR* loop. The increment of the infill line will be the infill gap calculated in equation (3.6).

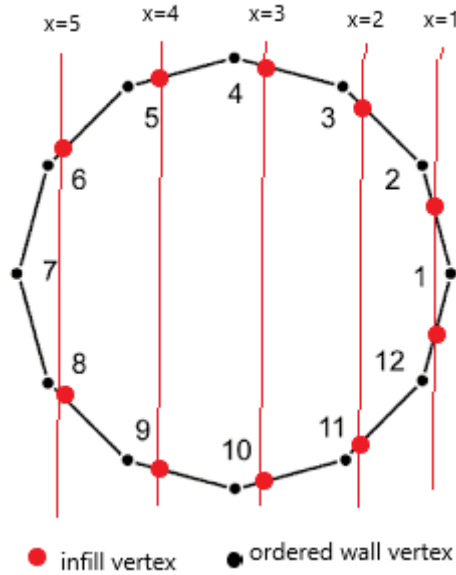


Figure 3.23: Vertical infill.

The intersection line can be computed by using equation (3.4) and (3.7). Let  $r_0 = \langle x_k, y_k, z_k \rangle$ ,  $r = \langle x_{k+1}, y_{k+1}, z_{k+1} \rangle$  and  $v = \langle x_k - x_{k+1}, y_k - y_{k+1}, z_k - z_{k+1} \rangle$ . With these values, parametric equation of the line can be formed to find the intersection points,  $\langle x_{infill}, y_{infill}, z_{infill} \rangle$ . Next, the scalar,  $t = \frac{x_{infill} - x_k}{x_{k+1} - x_k}$  with the given  $x_{infill}$  value as the  $x_{infill}$  value will be the infill line that obtained from equation (3.6). After the scalar,  $t$  has been computed, the intersection point can be determined by using the parametric equation which shown in equation (3.7). The variable,  $k$  is an expression of the point increment from 0 to  $n$  where  $n$  is the number of vertices.

$$\begin{aligned} x_{infill} &= x_k + (x_{k+1} - x_k)t; & y_{infill} &= y_k + (y_{k+1} - y_k)t; & (3.7) \\ z_{infill} &= \text{layer height} \end{aligned}$$

All the ordered vertices on the same layer will be examined to compute the intersection points. Each ordered vertex will form a line with the later vertex in the array to determine the intersection point and store it if there is any. For instance, if  $k = 0$  then vertex 1 and vertex 2 in Figure 3.23 will connect with each other. If infill line is  $x = 1$ , the infill line intercepts with the line 12. Thus, there is an infill vertex being stored to an infill array. Next,  $k$  is

incremented by 1 ( $k = 1$ ) and vertex 2 and vertex 3 will form a line (line 23). The infill line remains as  $x = 1$  but there is no intersection between infill line and line 23. Thus, no infill vertex is found and stored. In this case, the  $k$  will be incremented until  $k = 11$  to form a last line between vertex 12 and vertex 1. There will be an intersection point between the line and the infill line ( $x = 1$ ). Thus, there will be two infill vertices for  $x = 1$ .

After all ordered wall vertices have been examined for infill line,  $x = 1$ , all the ordered vertices will be examined for infill,  $x = 2$  with same procedure that  $k = 0$  until  $k = n$ . Once all vertices have been checked for a particular infill line, the infill line will be incremented by the infill line gap,  $d$ . The example assumes the infill line gap,  $d$  to be 1 mm which makes infill line,  $x = \{1, 2, 3, 4 \text{ and } 5\}$ . The infill line,  $x$  will be range from smallest  $x$  value to largest  $x$  value with increment of infill line gap. Arrival of the largest  $x$  indicates that all the infill vertices on particular layer has been determined. Henceforth, the algorithm will repeat in the next layer until the last layer.

For every alternate layer, the infill pattern will swap the direction from vertical infill to horizontal infill as shown in Figure 3.24. The theory for horizontal infill generation will be similar with the vertical infill detection. Instead of the X-axis variable in vertical infill algorithm, they will be substituted with Y-axis variable for horizontal infill detection. In this program, horizontal infill will be applied to odd layer while vertical infill will be adopted by even layer. The implementation of the algorithm in C++ programming is shown in Figure 3.25.

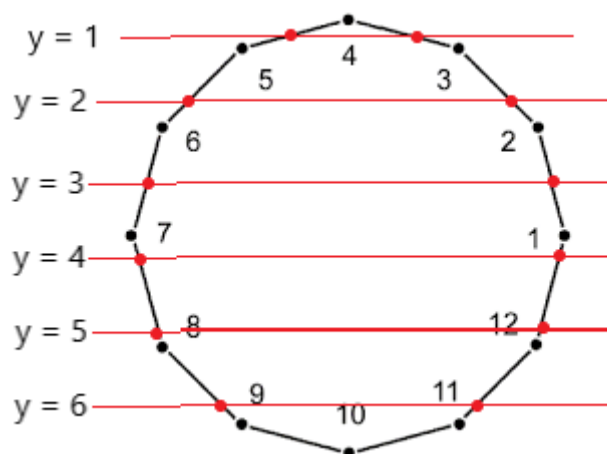


Figure 3.24: Horizontal infill



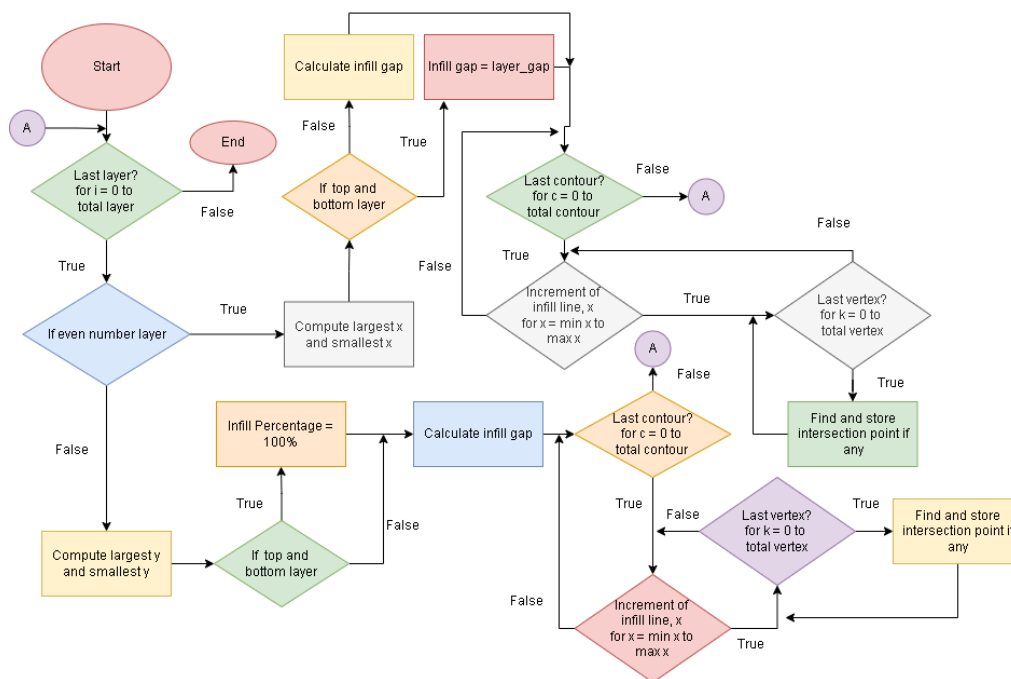


Figure 3.25: Flowchart of infill vertex generator

### 3.9.2.2 Line Infill Pattern

The infill vertex that obtained in the mentioned chapter - Infill Vertex Detection, are detected from the smallest  $x$  to largest  $x$ . The infill vertex will be stored from negative  $X$ -axis to positive  $X$ -axis. The lines that formed by the ordered wall vertices examined the intersection point in anticlockwise direction. Therefore, the first infill vertex for each infill line will always locate at the negative  $Y$ -axis while the later vertex will be located at the positive  $Y$ -axis. This will lead to a creation of crossing line when connecting the last vertex to the next infill line first vertex. For instance, in Figure 3.26, the last infill vertex, (2), of the infill line,  $x = 1$  will travel across the  $X$ -axis to reach the first infill vertex, (3) of the infill line,  $x = 2$ . The infill pattern that is formed due to the storing manner is shown in purple arrows of the Figure 3.26. The manner of storing the infill vertex will be adopted to the horizontal infill as well. Hence, line infill pattern can be formed by the behaviour of detecting and storing the infill vertex that have mentioned in Infill Vertex Detection in section 3.9.2.1.

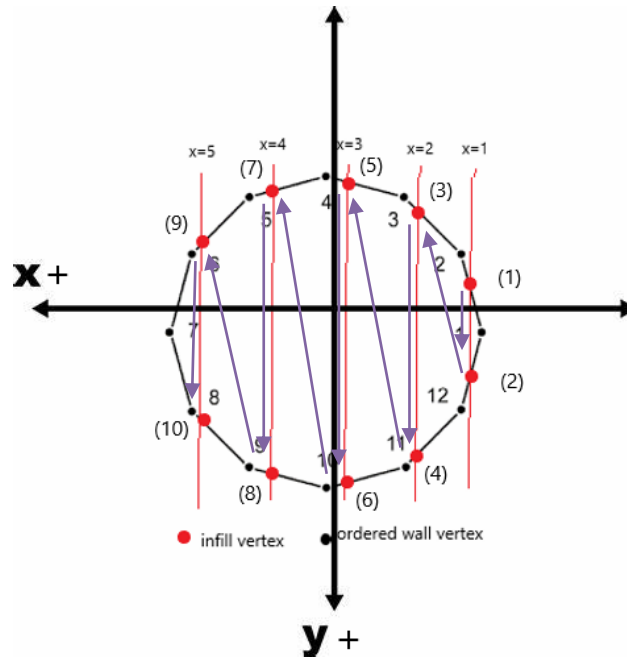


Figure 3.26: Line infill pattern's illustration.

### 3.9.2.3 Linear Infill Pattern

Line infill patterns are created due to the default storing manner of the infill vertex detection. Linear infill patterns are the extension of the line infill pattern. It can be implemented by interchanging the elements of the infill vertex array. Each infill line will only detect two vertices. Thereinafter, two vertices for an infill line will be expressed as a set of vertices. If there are ten infill vertices, there will be five sets of vertices. In Figure 3.27, there are five sets of infill vertices as follows:

- 1) set 1: [infill vertex (1), infill vertex (2)]
- 2) set 2: [infill vertex (3), infill vertex (4)]
- 3) set 3: [infill vertex (5), infill vertex (6)]
- 4) set 4: [infill vertex (7), infill vertex (8)]
- 5) set 5: [infill vertex (9), infill vertex (10)]

The infill vertices in the even set of infill vertices (set 2 and set 4) will encounter an interchange of the infill vertex. As an illustration, the vertex (3) and vertex (4) of the line infill pattern in Figure 3.26 will be vertex (4) and vertex (3) of the linear infill pattern in Figure 3.27 respectively. The infill line formation will be altered by interchanging the vertices in the even set. After

the last *vertex* (2) of the infill line,  $x = 1$ , the vertex will form a line with the *vertex* (3) instead of forming line with *vertex* (4). The implementation of this interchanging infill vertices in even vertex set in C++ programming is illustrated in Figure 3.28.

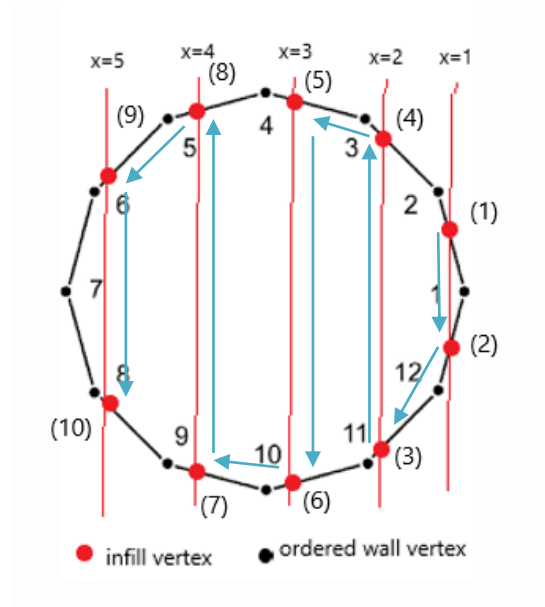


Figure 3.27: Linear interior geometric generation.

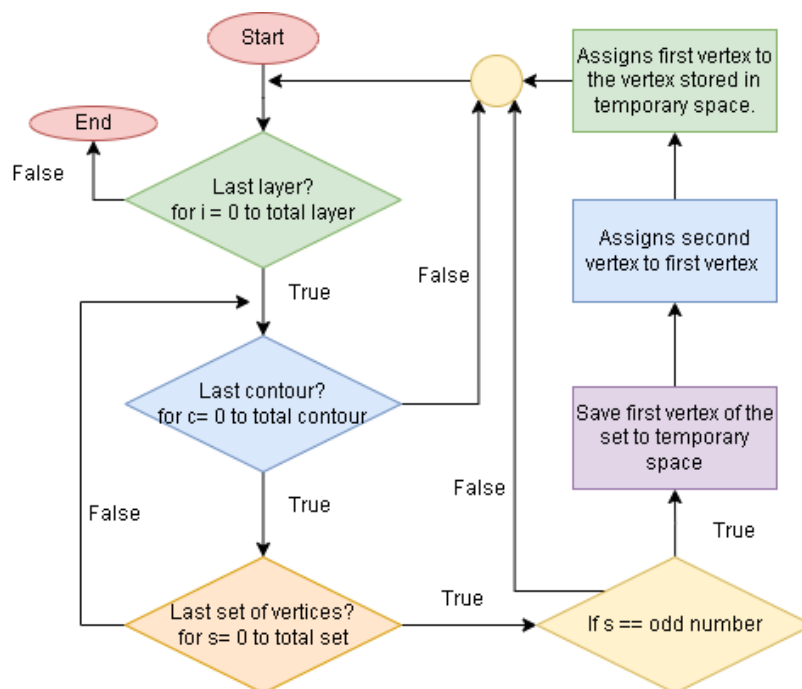


Figure 3.28: Flowchart of interchanging vertices for linear infill pattern.

### **3.10 Top and Bottom layer**

Based on the literature review, the walls extend upward along the Z-axis and surround a print's horizontal perimeter. The top and bottom layers completely enclose the horizontal space inside the boundaries set by the walls. The shell and top and bottom layer are alike but distinct in the direction of printing. The layer height affects the line gap of the shell in Z-axis. However, the top and bottom layer will be printed like infill line, but the infill gap will be assigned to the value of layer height.

The number of top and bottom layers are controlled by the top and bottom thickness. The number of top and bottom layers is determined by dividing the top and bottom thickness by the layer height. The number of top and bottom layers is rounded to a whole number. Users may control the thickness of the print's top and bottom layers by adjusting the top/bottom thickness. A higher value guarantees that all openings on the top and bottom layers are entirely sealed. However, this can potentially lengthen print times and use more filament. The implementation of top and bottom layers is illustrated in Figure 3.25. The line gap for top and bottom will be set to 0.4 mm.

### **3.11 Rendering Process**

Based on the literature review under Chapter 2.8, there are several types of primitive drawing which draw lines and triangles in the 3D space of OpenGL. The structure of the imported object, sliced data, infill data and the preview form of the sliced object will be presented through the primitive drawings provided by OpenGL engine. The rendering process for the simulation of printing process will be a sequential process in this 3D printing simulator program.

First of all, the process includes the rendering of solid body of the imported object before the simulation of printing process. This render mode is programmed for the user to verify the correctness of the imported object and visualize the object before the printing starts. The user may check if the imported body matches with its modelled body. The technique of rendering a solid body will be discussed in Chapter 3.11.1. To exit the preview of the

imported file, a small letter 'p' can be pressed on the keyboard and the program will proceed to the next printing process.

The next render part will be the slice-form of the imported object. The slice-form of the 3D print object will be rendered progressively along the printing path. Filament will be simulated along the printing path through the line drawing, *GL\_LINES* in the OpenGL as discussed in Chapter 3.11.2.2. Besides than presenting in line form, the filament can be simulated more realistically in the pipe form. Multiple cylinder-shaped bodies are joined with one another to display a printing filament in pipe form as mentioned in Chapter 3.11.2.1. The sequence of the printing will be prior to the shell of the sliced model on every layer. The printing of the infill part of the sliced model will be started with the completion of the shell printing. The printed shell and the infill part will retain in the 3D space. This rendering procedure will be discussed in Chapter 3.11.2.1. The slice-form model will be rendered layer by layer until all the layers are printed. Once the printing is completed, a full form of the slice-form model will be displayed.

Apart from that, the preview of the slice-form model can be presented before the completion of the printing process by pressing 'f'. The user may skip the printing process to view the full form of the sliced model. This preview of the sliced model is limited to the version of the pipe form render mode.

### **3.11.1 Solid Body**

The graphical visualization of the solid body will be rendered with the primitive drawing of *GL\_TRIANGLES*. This primitive drawing will render triangles using the triangular facets' vertices and normal that have been extracted with the method as discussed in Chapter 2.3. Figure 3.29 shows an example of a cube that is formed by 8 vertices from STL file and 12 triangular facets. Each of the triangular facet are formed by three vertices. Henceforth, there are twelve combinations of the vertices to form twelve facets. The twelve combinations of the faces made out of three vertices are shown in Table 3.4. In the simulator program, each of the vertex comprises of three axes coordinates which are X, Y and Z axes. Thus, there are nine numbers for each

vertex and 3 number for the facet's normal to be included for graphical visualization of a solid body. With this concept, any model that is imported can be rendered and visualized in solid body form.

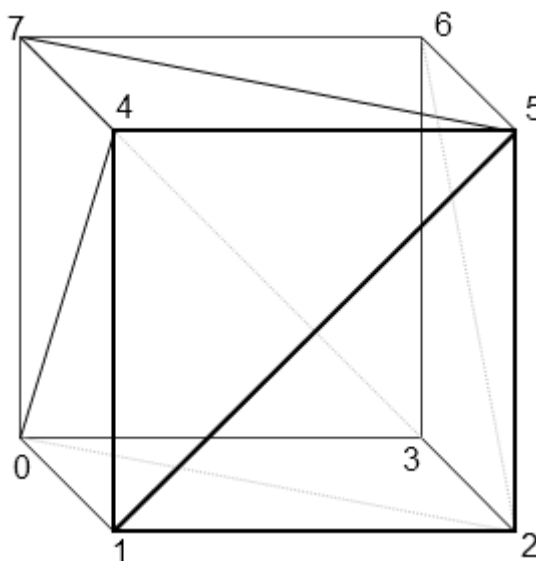


Figure 3.29: A cube that formed by GL\_TRIANGLES.

Table 3.4: The combinations of 3 vertices to form 12 facets for the cube.

Facet	Combination of 3 vertices
1 <sup>st</sup> Facet	0, 1, 4
2 <sup>nd</sup> Facet	1, 4, 5
3 <sup>rd</sup> Facet	1, 2, 5
4 <sup>th</sup> Facet	2, 5, 6
5 <sup>th</sup> Facet	2, 3, 6
6 <sup>th</sup> Facet	3, 6, 7
7 <sup>th</sup> Facet	0, 3, 7
8 <sup>th</sup> Facet	0, 4, 7
9 <sup>th</sup> Facet	0, 1, 2
10 <sup>th</sup> Facet	0, 2, 3
11 <sup>th</sup> Facet	4, 5, 7
12 <sup>th</sup> Facet	5, 6, 7

### 3.11.2 Filament Rendering

OpenGL engine provides primitive types drawing, comprising points, lines, quadrilaterals, and triangles. These primitive drawing required the geography information of a sequence of points in 3D space for rendering. In this project, a sequence of vertices that stored layer by layer after the basic slicing method that implemented above will be the input points for rendering a layered body. There will be two render mode to demonstrate the printing filament in this project. One of the draw modes will render the filament in simple line form while the other will render the filament in form of cylinder-shaped. The sliced data and infill data will be connected using multiple lines and continuous cylinders in each draw mode to demonstrate the object's filament in layers.

#### 3.11.2.1 Filament in Pipe form

The filament rendering in pipe form will be applied to present the shell and the infill of a slice-form model. For 3D printing process, the heated 3D printing filament will be extruded from a nozzle to form the 3D print object. The nozzle will move slowly along the printing path and extrude the melted filament when the nozzle moves from a vertex to the next vertex. The melted filament will form and solidify on the hot end according to the movement of the nozzle. Therefore, to simulate the printing process, a pipe will be extruding from a vertex to the next vertex along a straight path. Extensive numbers of circles will be formed along the moving path to create a cylinder/ pipe to represent the extruding filament. Each circle is formed by 48 vertices as explained in Chapter 2.8.2. An example of a circle that is formed by a polygon with 48 vertices is shown in Figure 3.30.

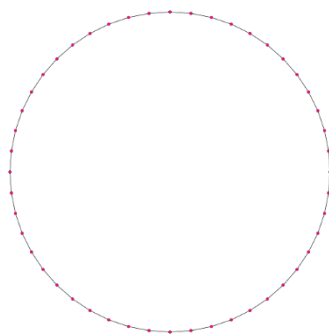


Figure 3.30: A tetracontaogon

Apart from that, the normal vector of a circle that formed for pipe formation is subject to the vector of the line that is desired to render. Figure 3.31 shows a sample of a pipe that is formed by eight circles. The normal vector of the circles is aligned with the vector of A and B. Let A and B to be the start vertex and an end vertex. On the other hand, Figure 3.32 shows an error formation of pipe due to the misalignment of line vector and normal vector of the circles. The red arrow in Figure 3.32 shows the normal vector of a circle while green arrow shows the vector of line AB. Cylinder in yellow colour indicates the pipe that is desired to generate. The normal vector of the circle is perpendicular to the vector of line AB which leads to failure of pipe formation.

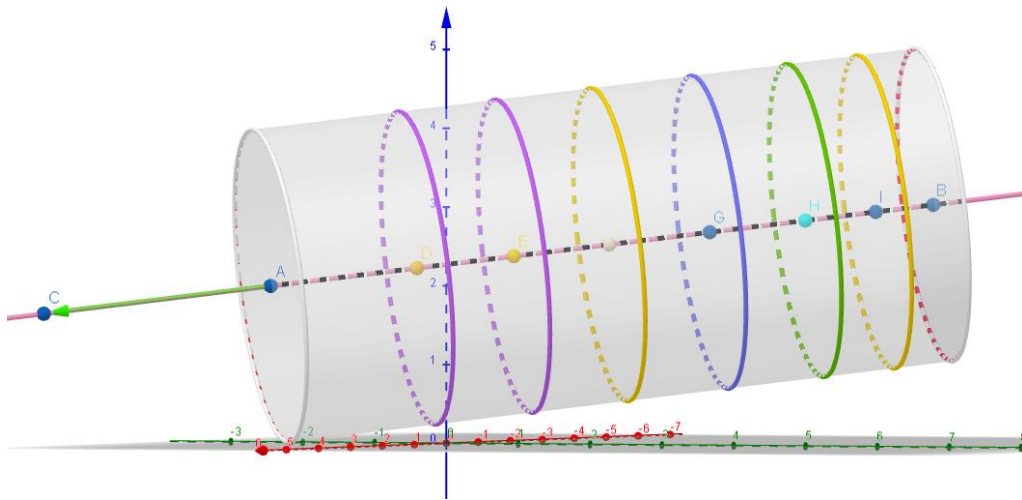


Figure 3.31: Correct formation of pipe

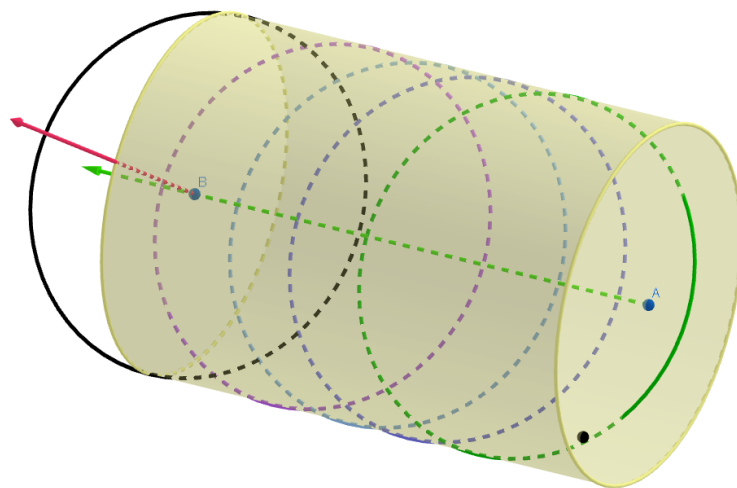


Figure 3.32: Failure of pipe formation



In this 3D printing simulator program, the normal vector of a circle will parallel with either X-axis or Y-axis depends on the angle between a line and the X-axis. There are four quadrants needed to be considered as shown in Figure 3.33. The angle between a line and the X-axis is determined by the equation (3.8). If the magnitude of the angle between a line and the X-axis is smaller than  $45^\circ$ , the normal vector of the circle will parallel to X-axis. Otherwise, the normal vector of the circle will parallel to Y-axis.

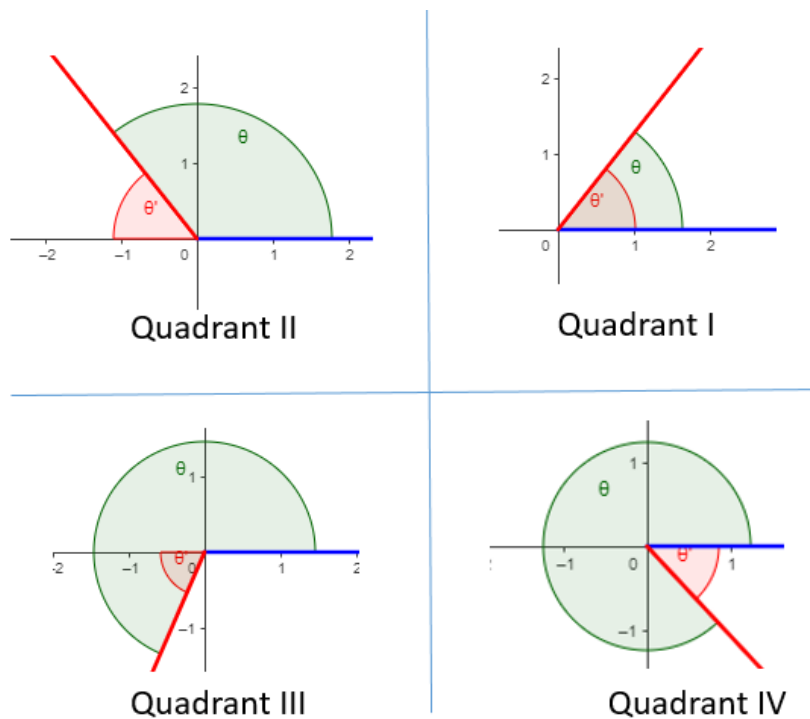


Figure 3.33: Possible cases for the angle between a line and the X-axis.

$$\theta' = \left| \tan^{-1} \frac{y_2 - y_1}{x_2 - x_1} \right| \quad (3.8)$$

Once the direction of the circles' normal vector has been determined, the pipe will be generated. The generated pipe will retain in the 3D space to represent the solidified filament.

There are seven functions defined and implemented to render the printing process for a 3D print object. The usage of the seven functions is described in Table 3.5. In each function, a set of vertices that consists of one starting point and an end point are used to form a pipe. They will be used to

build and compute points in between for building a pipe shaped filament in a defined C++ function - *buildPath*. This function generates a set of points in a straight path with the input parameters of a start vertex and an end vertex. The path generation required the distance on X and Y axes and Euclidean distance between the start point and end point to compute the points' coordinates ( $x$ ,  $y$  and  $z$ ) along a straight line. The distance between X coordinate of start point and end point,  $x_2 - x_1$  and Y coordinate's absolute distance,  $y_2 - y_1$  are calculated for point increment from start point to end point. While Euclidean distance,  $d$ , is computed to the number of steps required for both X and Y coordinate of starting point to reach the end point.

In Figure 3.34, let  $(x_1, y_1)$  equals to  $(1, 2)$  and  $(x_2, y_2)$  is  $(3, 5)$ . The absolute distance for X coordinates will be 2 mm and Y coordinates will be 3 mm while the Euclidean distance is  $\sqrt{5}$  mm. The gap computed between each point are set to be 0.0125 mm which equivalent to a moving step of a NEMA 17 stepper motor of the Ender 3 3D printer. Hence, the Euclidean distance of the points,  $\sqrt{5}$  will divide by a step of stepper motor which is 0.0125 mm to get the number of steps,  $i^{th}$  steps – 178 steps. Thus, the X coordinate of starting point takes  $2/178$  mm per step to move from coordinate 1 to 3 whereas Y coordinate take  $3/178$  mm per step to move from coordinate 2 to 5. The equation (3.9) is to generate the points in between the start and end point.

$$x_i = x_1 + (x_2 - x_1) \times \frac{i}{d/0.0125mm} \quad (3.9)$$

where,  $x_i$  is X coordinate of a point lied on the straight line between start and end point,  $x_1$  as the X coordinate of start point and  $x_2$  as the X coordinate of end point,  $d$  is the Euclidean distance and  $i$  is an integer from  $\{0, 1, 2 \dots i^{th} \text{ steps}\}$ . This equation is applied for generating Y coordinate of point lied on the straight line as well. A *FOR* loop is used to do the  $i^{th}$  increment to generate all the points lies on the path.

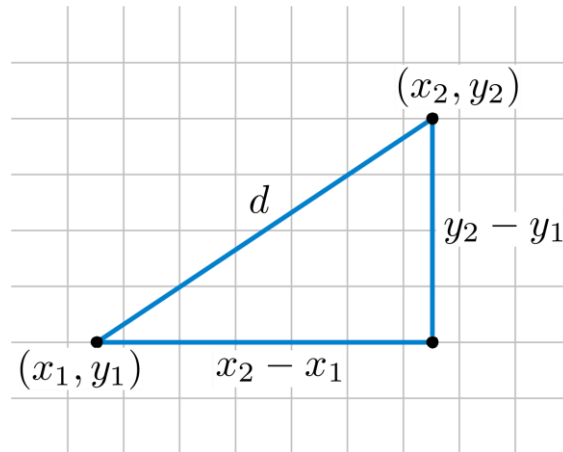


Figure 3.34: A triangle.

Equation (3.9) is applicable if the distance between  $x_2$  and  $x_1$  (or  $y_2$  and  $y_1$ ) is greater than the stepper motor's smallest step which is 0.0125 mm. The number of steps,  $i^{th}$ , will be zero which leads to an error for this rendering method as no points are generated. The solution for this issue is to assign 0.0001 to the distance between the points to generate a point that is 0.0001 mm away from the starting point and the number of steps takes will be 1 step. After the reassignment of the distance, the sign of the distance is required to change as well as the sign is subject to the position of the start and end point. If the starting X coordinate is greater than end X coordinate, the start point will minus the distance to reach the end point (same goes to Y axis). Therefore, the equation is generalized by replace distance to absolute distance and the sign is subject to the points' position. The general equation is shown in (3.10).

$$x_i = x_1 \mp |x_2 - x_1| \times \frac{i}{d/0.0125mm} \quad (3.10)$$

Table 3.5: The functions defined and usage of the functions for object rendering in pipe form.

No.	Functions name	Description
1.	<i>renderPrinting()</i>	To render the extruding filament that printing the wall of an object.
2.	<i>renderInfill()</i>	To render the extruding filament that printing the infill of an object.

3.	<i>renderPrinted()</i>	To render the solidified filament of the wall of an object on previous layers.
4.	<i>CurrPrinted()</i>	To render the solidified filament of the wall of an object on current layer. It renders the vertex set that have rendered in the <i>renderPrinting()</i> .
5.	<i>CurrInfill()</i>	To render the solidified filament of the infill of an object on current layer. It renders the vertex set that have rendered in the <i>renderInfill()</i> .
6.	<i>PrintedInfill()</i>	To render the solidified filament of the wall of an object on the layer before the printing layer.
7.	<i>CurrLayer()</i>	To render the solidified filament of the wall of an object on current layer. It is significant to retain the printed wall on current layer during the printing process of infill on the same layer.

The functions that are illustrated in Table 3.5 is presented in Figure 3.35 and Figure 3.36 with different colours to different functions. The filament that is coloured with matcha green in Figure 3.35 illustrates the solidified filament of the wall on previous layers and the respective function is *renderPrinted()*. Apart from that, the purple colour pipe illustrates the result of the function – *CurrPrinted()* whereas the white pipe body that is rendered below the nozzle is the outcome of the function – *renderPrinting()*. The cyan colour filament represents the printed infill of an object on the previous layer which is the function of *PrintedInfill()*. In Figure 3.36, the pink filament shows the pipe rendered by the *renderInfill()* while the pickle green shows the pipe rendered by the *CurrInfill()*. Last but not least, the red pipe is the result of the function – *CurrLayer()*. Furthermore, the preview of the slice-form model will be shown with a keyboard press of ‘f’. Nested FOR loop is implemented to render all the ordered list of slice data contour by contour and layer by layer. An example for preview of a sliced model is displayed in Figure 3.37.

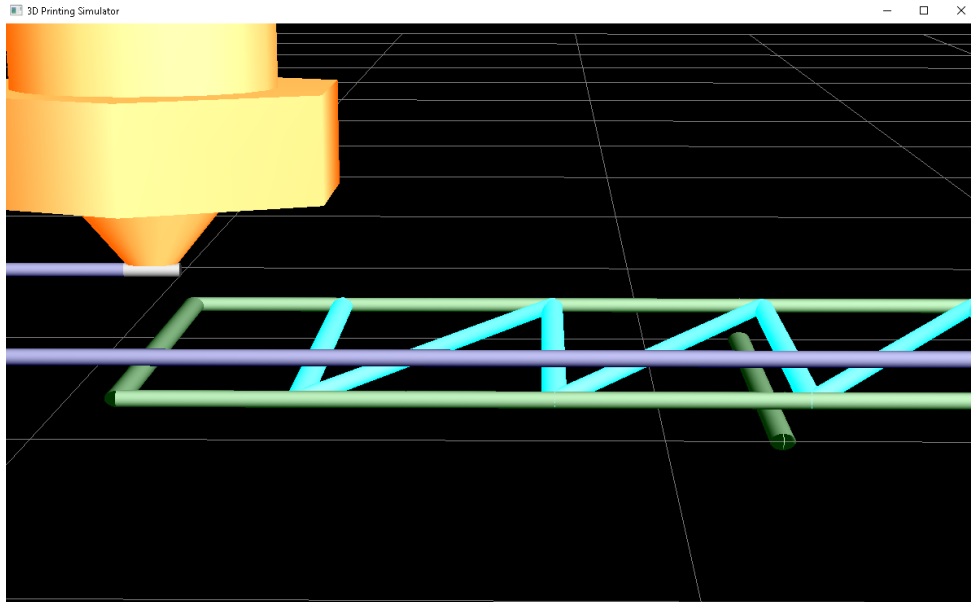


Figure 3.35: renderPrinted (matcha green), CurrPrinted (purple),renderPrinting (white) and PrintedInfill (cyan).

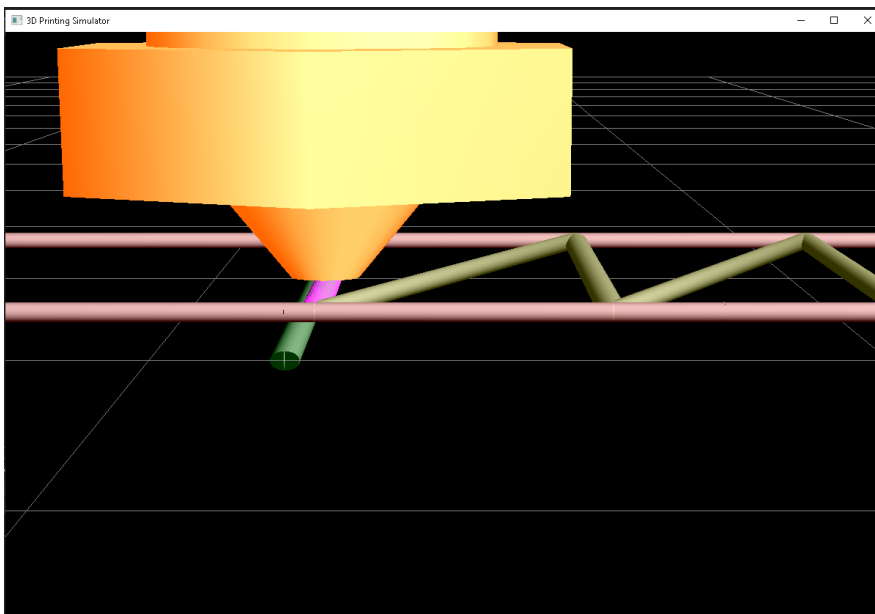


Figure 3.36: renderInfill (pink), CurrInfill (pickle green) and CurrLayer (red).

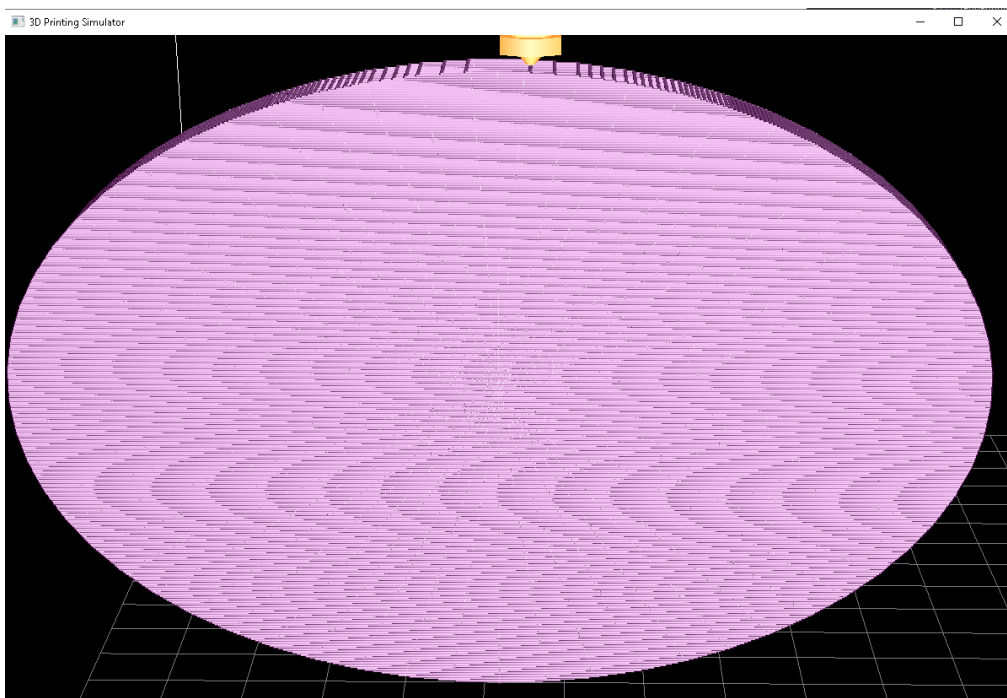


Figure 3.37: Preview of a sliced model in pipe form.

### 3.11.2.2 Filament in Line form

The vertices that obtained from slicer and infill generator can be rendered through one of the primitive drawings in OpenGL which is *GL\_LINES*. *GL\_LINES* is used to render a line between a vertex to the next vertex to form a connection between each vertex. Therefore, the vertices of the sliced data and infill data can be displayed in line form for visualization in the OpenGL 3D space via *GL\_LINES*.

Implementation of *GL\_LINES* to every vertex will be executed inside a FOR loop. This will render all the sliced data and infill data accordingly for the visualization of the layered object. There are five parts for line form rendering which are similar with the functions in the pipe form rendering. The functions are not included in line form rendering are *CurrPrinted()* and *CurrInfill()*. This is because line form rendering mode excludes the consideration of rendering points between each set of vertices, instead this rendering mode just draw line between each set of vertices. It is developed to generate quick view for user as it is comparatively computational inexpensive than pipe form rendering. This is due to the reason that pipe form rendering required to draw extensive number of circles between the vertices to create a

cylinder. Each contour of a circle is created by 48 vertices. Let assume that a pipe between a set of vertices is formed by ten circles. There will be 480 vertices required to compute for a formation of a pipe. Otherwise, line form rendering uses a set of vertices to render a line to join the vertices. There will be two vertices needed for a formation of line. Therefore, line form rendering is much computational inexpensive than pipe form rendering. The visualization of line form rendering is shown in Figure 3.38.

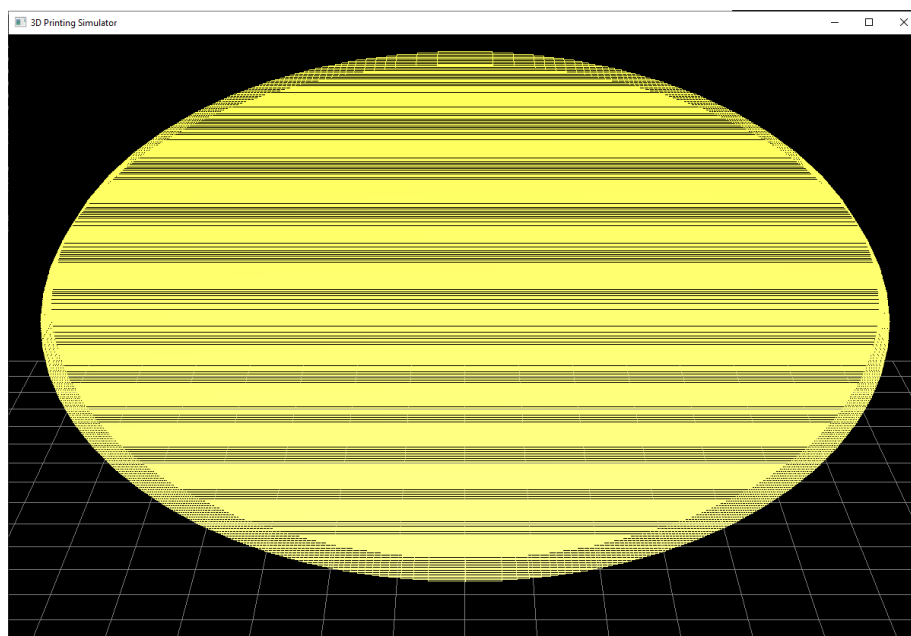


Figure 3.38: Sliced data rendered with GL\_LINES.

### 3.12 Gantries movement

Simulation of the gantries for a 3D printer can be done by rending three cuboids. The cuboids can be rendered in OpenGL by forming two triangles form a face and total will be 12 faces to form a cuboid. The movement of the cuboids which represent the movement of gantries depends on the location of current rendering line. The robot geometry of the 3D printer is a cartesian coordinates which consists of three prismatic or sliding joints. The distance,  $d_i$  of each prismatic joint can be computed by inverse kinematic. The location of the tool tip will be the coordinate of the vertex that is rendering at that moment. Figure 3.39 shows a schematic diagram of a cartesian robot which can represent a 3D printer

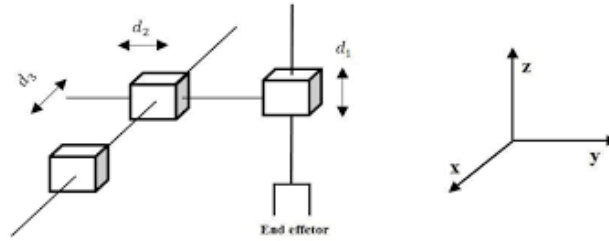


Figure 3.39: Cartesian Robot Schematic Diagram

The Denavit-Hartenberg Matrix is a transformation matrix from one coordinate frame to the next. After computing the transformation matrices, the distance,  $d_i$  can be determined using inverse kinematics with known coordinates of rendering vertices,  $x$ ,  $y$  and  $z$ .

Table 3.6: Debavit-Hartenberg Table.

$i$	$\alpha_{i-1}$	$a_{i-1}$	$d_i$	$\theta_i$
1	0	0	$d_1$	0
2	90	0	$d_2$	90
3	90	0	$d_3$	0
E	0	0	$L_1$	0

where

- $a_i$  is the distance from  $Z_i$  to  $Z_{i+1}$  measured along  $X_i$ .
- $\alpha_i$  is the angle from  $Z_i$  to  $Z_{i+1}$  measured about  $X_i$ .
- $d_i$  is the distance from  $X_{i-1}$  to  $X_i$  measured along  $Z_i$ .
- $\theta_i$  is the angle from  $X_{i-1}$  to  $X_i$  measured about  $Z_i$ .

Transformation matrices:

$${}^0_1T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^1_2T = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & d_2 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^2_3T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & d_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^3_E T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^0_2T = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & d_2 \\ 1 & 0 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^0_3T = \begin{bmatrix} 0 & 0 & 1 & -d_3 \\ 0 & -1 & 0 & d_2 \\ 1 & 0 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$${}^0T_E = \begin{bmatrix} 0 & 0 & 1 & L_1 - d_3 \\ 0 & -1 & -1 & d_2 \\ 1 & 0 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \equiv \begin{bmatrix} r & r & r & x \\ r & r & r & y \\ r & r & r & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

- $x$  is the x-axis coordinate of the rendering vertex.
- $y$  is the y-axis coordinate of the rendering vertex.
- $z$  is the z-axis coordinate of the rendering vertex.

Inverse Kinematics:

$$L_1 - d_3 = x$$

$$L_1 - x = d_3 - (1)$$

$$d_2 = y - (2)$$

$$d_1 = z - (3)$$

### 3.13 Experiments

The program is tested with five 3D models which are a disk, a cylinder, a cube, a bracket and a spoon. Shell and infill of each sample are presented to analyse the capability of the simulator. The infill is presented in line form while the shell is shown in pipe form. Apart from that, the estimation time to print the 3D models are counted and compared with the actual printing time used. The model of the 3D printer used is Ender 3 and the selected G-code generator is ULTIMAKER CURA. The error percentage of estimated time and actual printing time is calculated with equation (3.11). Besides that, the line gap for infill, top and bottom layer is studied to measure the feasibility of the simulator to generate actual interior geometric.

$$Error\ percent = \frac{|Estimated\ time - Actual\ time|}{Actual\ time} \times 100\% \quad (3.11)$$

Disk model is tested with different sets of layer height which are 0.12 mm, 0.16 mm, 0.20 mm and 0.24 mm to examine the feasibility of this simulator upon the layer height. Next, cylinder model is used to test the feasibility of infill generation. The line gap of each infill line is measured to determine the feasibility of the simulator to generate infill that can match with real-world print. After that, a cube model is tested with different values of top and bottom thickness – 0.20 mm, 0.40 mm, 0.60 mm and 0.80 mm. Lastly,

two 3D models that obtained randomly are tested to examine the feasibility of simulator. The test specimens chosen are a bracket and a spoon.

### **3.14 Summary**

An ASCII format STL (stereolithography) file is imported into the program. Triangular facets' vertices are extracted from the imported STL file. Extraction of triangular facets' vertices is used to form the shell of a 3D print model. Slicer generates layers' vertices from the triangular facets' vertices based on the user-defined layer height. Layers' vertices are then well-organized through contour creation module to form shell contour of a 3D print model. Creation of exterior part of a 3D print object are completed once shell contours are formed. Formation of infill based on the ordered shell's vertices on each layer with user-defined density. Infill changes from horizontal to vertical infill on alternate layer to fill interior part of a 3D print model more evenly. Top and bottom layers of a model is skin of a 3D print model. Top and bottom layers have a similar interior geometric. The line gap of top and bottom layers is fixed at 0.4 mm unlike infill density defines the infill line gap. Movement of nozzle depends on the vertices of shell, skin and infill. Smallest step of the nozzle is based on the smallest step of NEMA 17 stepper motor which is 0.0125 mm. Print material of FDM 3D printing is simulated by forming continuous cylinder between each vertex. Line form rendering is a low computational cost method to simulate the printing trajectory. Several experiments are conducted to fine tune the print parameter in this simulator to match with reality.

## CHAPTER 4

### RESULTS AND DISCUSSION

#### 4.1 Introduction

This chapter discuss the C++ program that are designed and developed in accordance with the methodology to create a 3D printing simulator. A step-by-step user documentation is included in this section. The user guide comprises of the procedure of running the 3D printing simulator to generate a 3D print object. There are four 3D models being tested to examine the competence of the program to function as FDM 3D printing simulator. The selected 3D models are a disk, a cylinder, a bracket and a spoon.

#### 4.2 User Documentation

This section shows the procedure of using this 3D printing simulator and the keyboard press to control the process.

##### 4.2.1 Preparation of STL File in ASCII Format

The 3D printing simulation is started with the preparation of STL file from CAD software. The CAD software used to design the 3D models is SOLIDWORKS. After the design are completed, the 3D models are exported to STL file in ASCII format. The steps of exporting a CAD model to STL file in ASCII format is illustrated in Figure 4.1 and Figure 4.2.

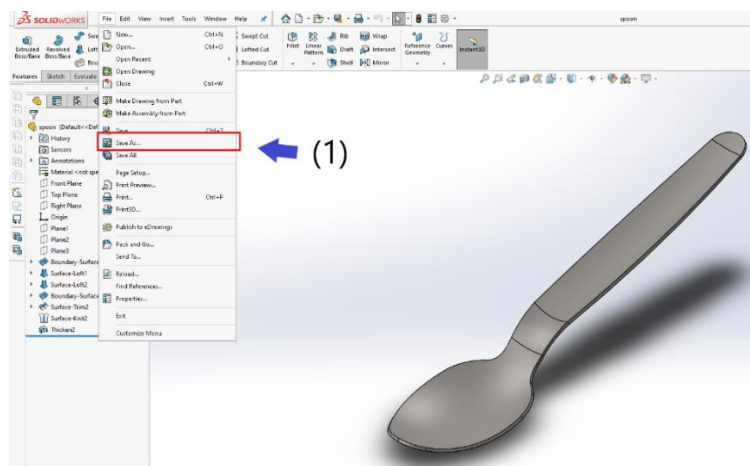


Figure 4.1: SOLIDWORKS interface to save CAD model.

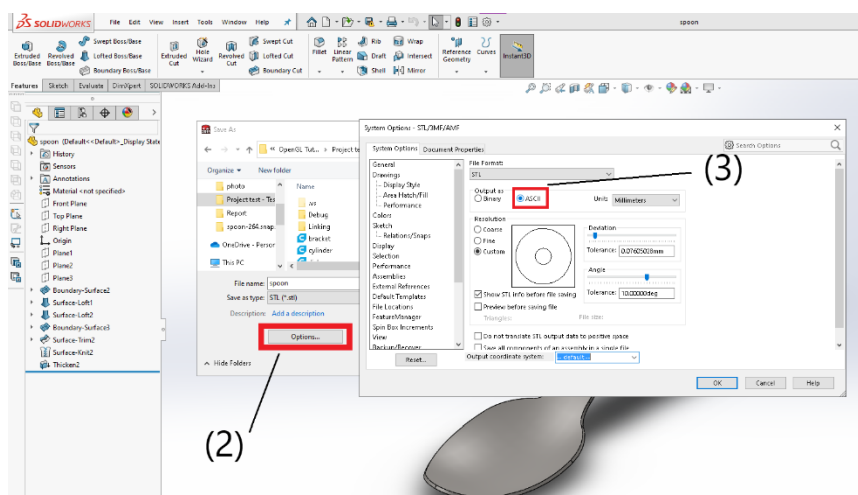


Figure 4.2: Save CAD model as STL file in ASCII format.

#### 4.2.2 Input Print Parameter

The program is started with a series of inputting print parameters. The window prompts users to import a STL file in ASCII format by entering the STL file's name. Once the STL file is successfully imported, the default print setting is displayed in the window. The default print setting of this simulator is shown in Table 4.1. The simulator allow users to customize their own print setting so users may customize the print setting according to their needs by typing 'c' after importing the file.

Table 4.1: Default print setting.

No.	Print Parameters	Value
1.	Layer Height	0.28 mm
2.	Infill Density	50 %
3.	Top Thickness	0.84 mm
4.	Bottom Thickness	0.84 mm
5.	Wall Print Speed	25 mm/s
6.	Infill Print Speed	50 mm/s
7.	Skin Print Speed	25 mm/s
8.	Initial Layer Print Speed	20 mm/s
9.	Infill Pattern	Line Pattern
10.	Rendering Mode	Pipe Mode

Users who would like to customize their own print setting will be prompted to enter their preferred parameters accordingly. The explanation and description of the parameters are provided before the prompt. Thus, users may refer to the description before the selection of value. After users have decided the new print setting, it is presented in the window for users to check. Users may obtain the estimation time for the 3D models to be printed in the window. The estimation printing time is further explained in terms of infill, wall and shell. Users may tune the print parameters to reduce the printing time by referring to the highest time-consuming section. Users may type 's' to confirm the print setting and proceed to the printing simulation. Figure 4.3 shows the user interface after importing the STL file. Figure 4.4 shows the description of print setting to users. Figure 4.5 shows the user interface after users customize the print setting.

```

D:\OpenGL Tutorial\Project test - Test II\Debug\Simulator.exe
3D Printer Simulation
=====
Place STL file into the same folder as this solution file.
Enter name of STL file : disk
=====
Default Print Setting
=====
STL File Name           : disk.stl
Layer Height            : 0.28 mm
Quality                 : Low Quality
Infill Density          : 50 %
Infill Line Distance    : 0.8 mm
Infill Pattern          : Line Pattern
Bottom Thickness        : 0.84 mm
Bottom Layer(s)        : 3 layer(s)
Top Thickness           : 0.84 mm
Top Layer(s)           : 3 layer(s)
Wall Print Speed        : 25.0 mm/s
Infill Print Speed     : 50.0 mm/s
Skin Print Speed        : 25.0 mm/s
Initial Layer Print Speed : 20.0 mm/s
Rendering Mode          : Pipe Mode
=====
Press C to customize the print setting ...
Press ANY KEY to confirm the default print setting ...
>> c

```

Figure 4.3: Import STL file and show the default print setting.

```

=====
Print Setting
=====
Layer Height

>> The height of each layer in mm. Higher values produce faster prints in lower
resolution, lower values produce slower prints in higher resolution.
(Range: 0.12 mm ~ 0.28 mm)
-----
Infill Density

>> Adjusts the density of infill of the print.
-----
Infill Pattern

>> The pattern of the infill material of the print. The linear and line infill
swap direction on alternate layers.
[Line pattern(L) / Linear pattern(Z).]
-----
Bottom Thickness

>> The thickness of bottom layers in the print, this value is divided by the
layer height defines the number of bottom layers.
-----
Top Thickness

>> The thickness of top layers in the print, this value is divided by the
layer height defines the number of top layers.
-----
Bottom Layers

>> The number of bottom layers, this value is rounded to a whole number.
-----
Top Layers

>> The number of top layers, this value is rounded to a whole number.
-----
Rendering Mode

>> 3D print object render in Line mode (L) / Pipe mode (P).

```

Figure 4.4: Description of print parameters.

3D printing simulator is launched once users enter ‘s’ to proceed. In the case of pipe mode rendering, solid body of the imported 3D model is presented. The printing process is started with the keyboard press of ‘p’. Users may pause the printing process by pressing ‘o’. While pausing the process, the solid body of the imported 3D body is presented. Users may press ‘f’ to switch to view the complete 3D printed object and press ‘g’ to return to the printing process. During the printing process, users may control the camera view to find a better view of the printing process through zoom, pan and rotate. Zoom in and zoom out to view the 3D print object close by pressing ‘z’ and ‘x’ respectively. Move the camera up and down via the keys – ‘q’ and ‘e’

respectively. Camera view can be panned through the keys – ‘w’, ‘s’, ‘a’, ‘d’ while rotate the view through the keys – ‘i’, ‘j’, ‘k’, ‘l’.

```

=====
Customize Print Setting
=====
Layer height (mm): 0.28
Infill density (%): 10
Infill pattern: z
Bottom thickness (mm): 0.84
Top thickness (mm): 0.84
Rendering Mode: 1
=====

                                Print Info
=====
STL File Name                   : disk.stl
Layer Height                     : 0.28 mm
Quality                           : Low Quality
Infill Density                   : 10 %
Infill Line Distance             : 4 mm
Infill Pattern                   : Linear Pattern
Bottom Thickness                 : 0.84 mm
Bottom Layer(s)                 : 3 layer(s)
Top Thickness                    : 0.84 mm
Top Layer(s)                    : 3 layer(s)
Wall Print Speed                 : 25.0 mm/s
Infill Print Speed              : 50.0 mm/s
Skin Print Speed                 : 25.0 mm/s
Initial Layer Print Speed       : 20.0 mm/s
Rendering Mode                   : Line Mode
=====

                                Keyboard controls
=====

Camera controls
Key      Function                Key      Function
w        Move camera forward      i        Rotate X axis counter clockwise
s        Move camera backward     k        Rotate X axis clockwise
a        Move camera left         j        Rotate Y axis counter clockwise
d        Move camera right        l        Rotate Y axis clockwise
q        Move camera up           e        Move camera down
z        Zoom in                  x        Zoom out
1        Isometric View           2        Top View
3        Right View               4        Left View
5        Front View
p        Exit solid body mode and starts printing process
=====

3D printing process control
b        Fast forward              n        Normal speed
f        Show entire 3D print body g        Back to printing simulation
=====

Height of the object              : 116.48 mm
Total number of printing layer(s) : 416 layer(s)
=====

                                Time Estimation
=====

Outer wall                        : 1 hours 8 minutes
Infill                            : 0 hours 40 minutes
Skin                              : 0 hours 1 minutes
=====

Total                             : 1 hours 50 minutes
=====

Press S to proceed to the simulation.....

```

Figure 4.5: User Interface after the customization of print setting.

### 4.3 Simulation Results

The program is tested with five 3D models which are a disk, a cylinder, a cube, a bracket and a spoon.

#### 4.3.1 Disk

The diameter of the disk is 50 mm with thickness of 3 mm as shown in Figure 4.6.

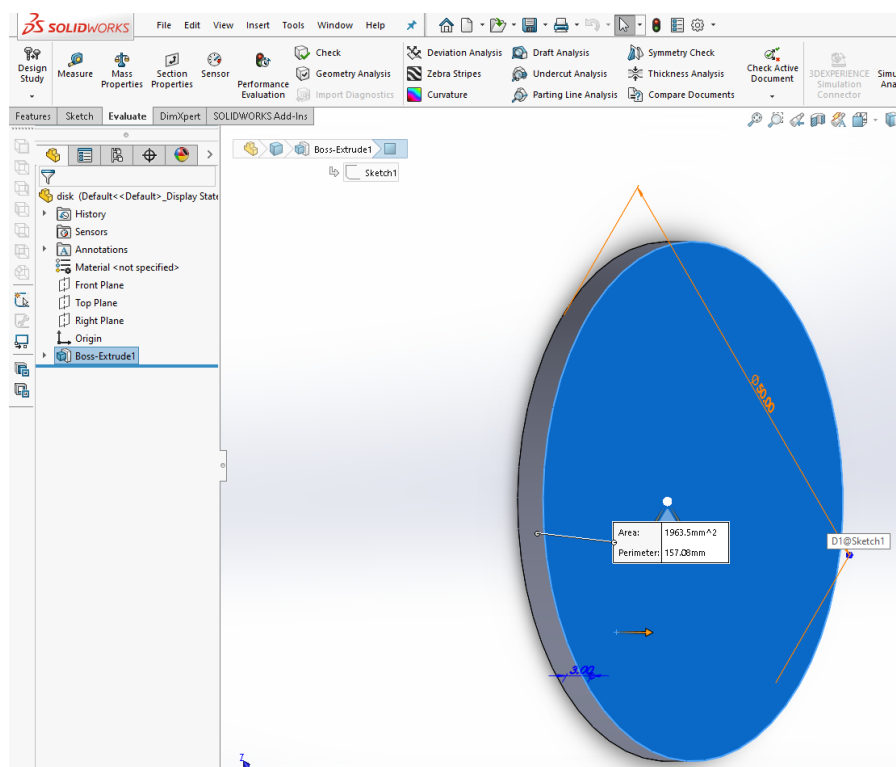


Figure 4.6: Disk was modelled in SOLIDWORKS.

The print parameters for the models are shown in the Table 4.2. The parameters are varied with the layer height – 0.12 mm, 0.16 mm, 0.20 mm and 0.28 mm. Estimation time for the layer height – 0.12 mm, 0.16 mm, 0.20 mm and 0.28 mm is 56 minutes, 42 minutes, 34 minutes and 24 minutes respectively. These disks are printed in a FDM 3D printer. The model of the 3D printer is Ender 3. The method of generating G-code in ULTIMAKER CURA software is shown in Appendix. The actual printing time for the disks is 1 hour and 15 minutes, 1 hour 4 minutes, 51 minutes and 36 minutes



respectively. The error percentage for sample 1, 2, 3 and 4 in terms of printing time is 33.33%, 33.33%, 34.38% and 34.12% respectively.

Table 4.2: Print parameters for four disk.

STL file	Sample	Print Parameters				
		Layer Height	Infill Density	Infill Pattern	Top Thickness	Bottom Thickness
disk	1	0.28 mm	50 %	Linear	0.84 mm	0.84 mm
	2	0.20 mm	50 %	Linear	0.84 mm	0.84 mm
	3	0.16 mm	50 %	Linear	0.84 mm	0.84 mm
	4	0.12 mm	50 %	Linear	0.84 mm	0.84 mm

Table 4.3: Test result of the disks.

Sample	Number of Layers	Estimation Time	Actual Time	Difference	Error percentage
1	179	24 minutes	36 minutes	12 minutes	33.33 %
2	250	34 minutes	51 minutes	17 minutes	33.33 %
3	313	42 minutes	1 hour 4 minutes	22 minutes	34.38 %
4	417	56 minutes	1 hour 25 minutes	29 minutes	34.12 %

The disk in the simulator with layer height of 0.12 mm, 0.16 mm, 0.20 mm and 0.28 mm is illustrated in Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10 respectively. The difference of different layer height of the disk can be viewed from the figures below. Figure 4.7 shows a finer surface of the models which has the layer height of 0.12 mm while surface of the disk is courser for layer height of 0.28 mm which shown in Figure 4.10. Figure 4.11 to Figure 4.14 show the infill part of the disk. The figures show that the infill of the disk increases with the number of layers. Figure 4.15 shows the exterior and interior of the disk in a section view.

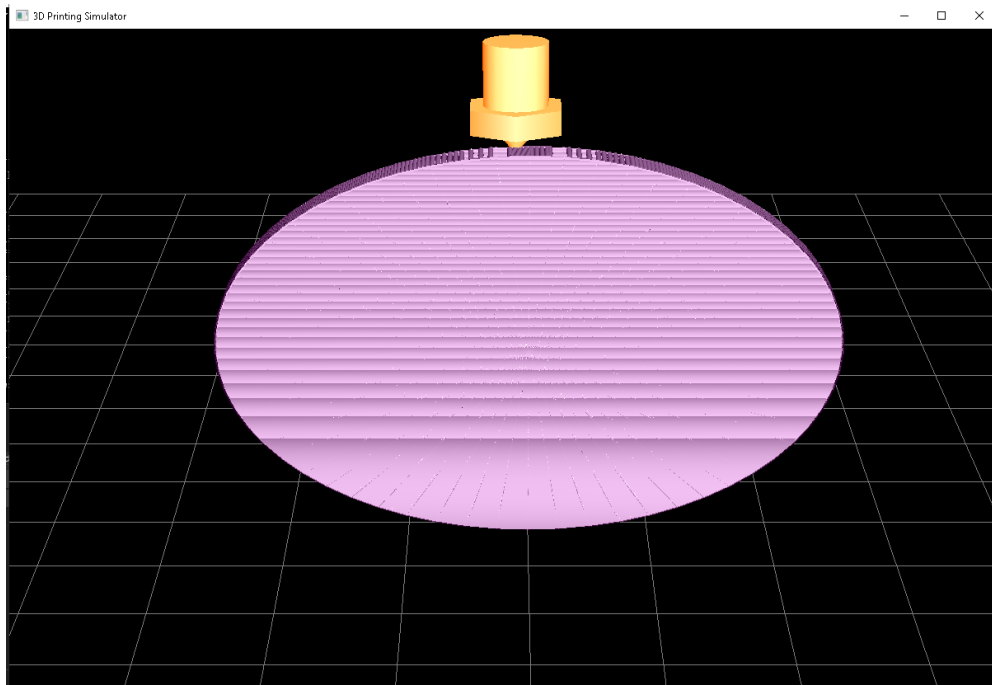


Figure 4.7: Disk is presented in simulator with layer height of 0.12 mm.

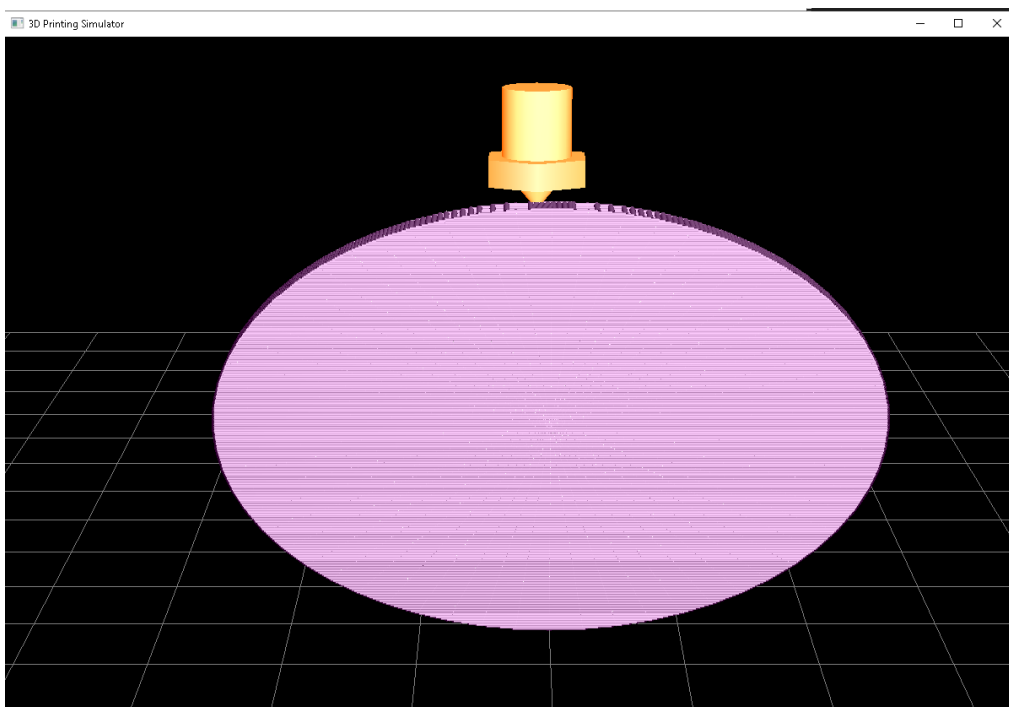


Figure 4.8 Disk is presented in simulator with layer height of 0.16 mm.

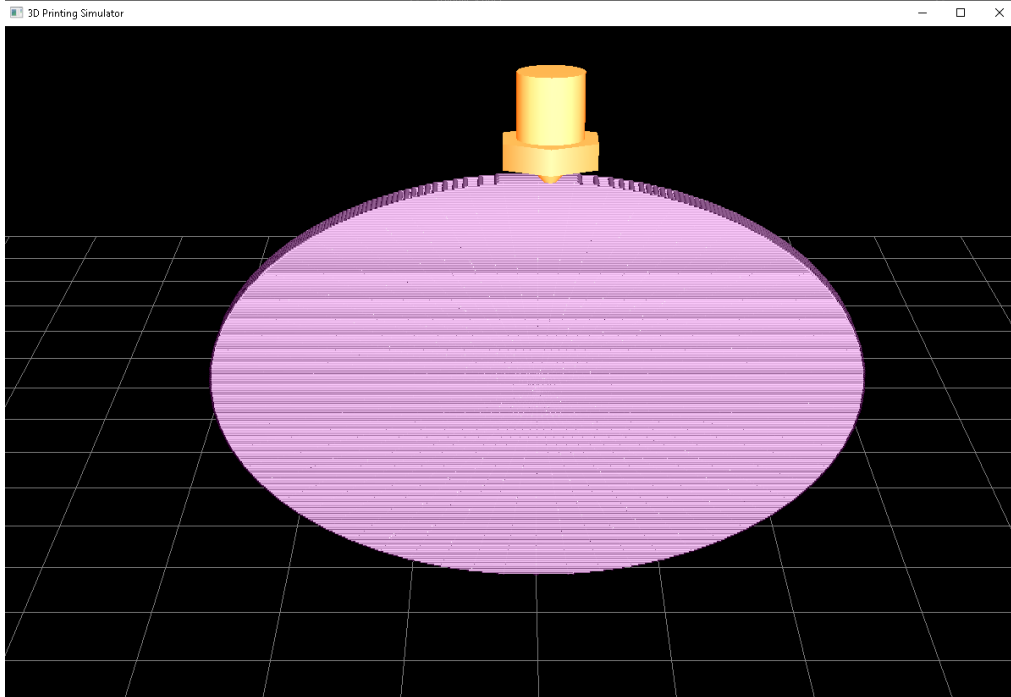


Figure 4.9: Disk is presented in simulator with layer height of 0.20 mm.

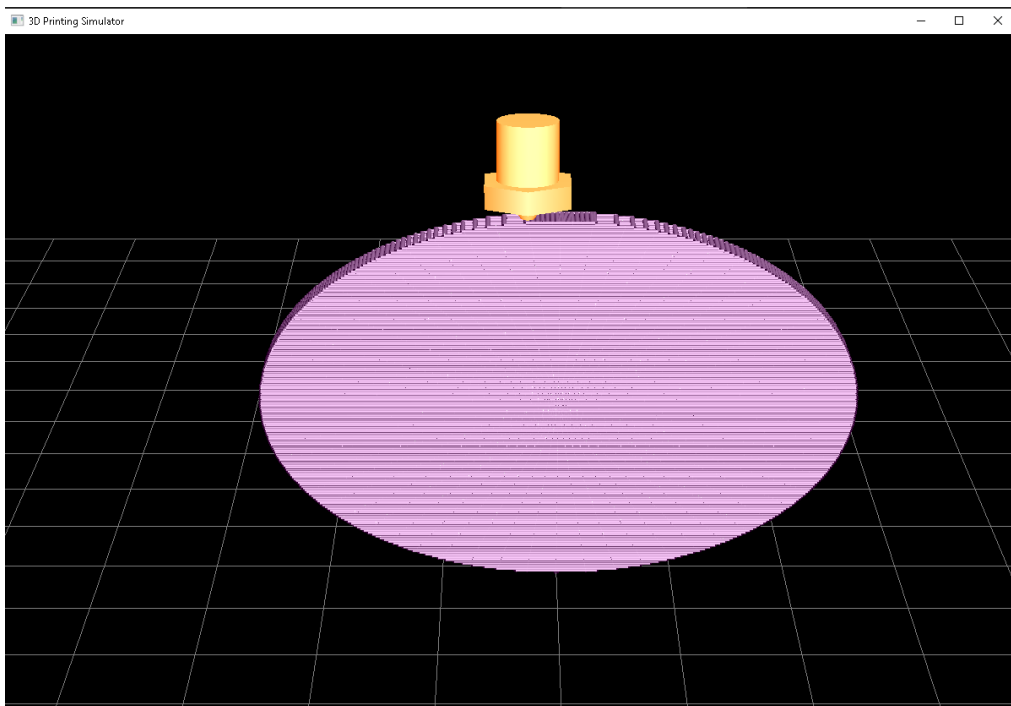


Figure 4.10: Disk is presented in simulator with layer height of 0.28 mm.

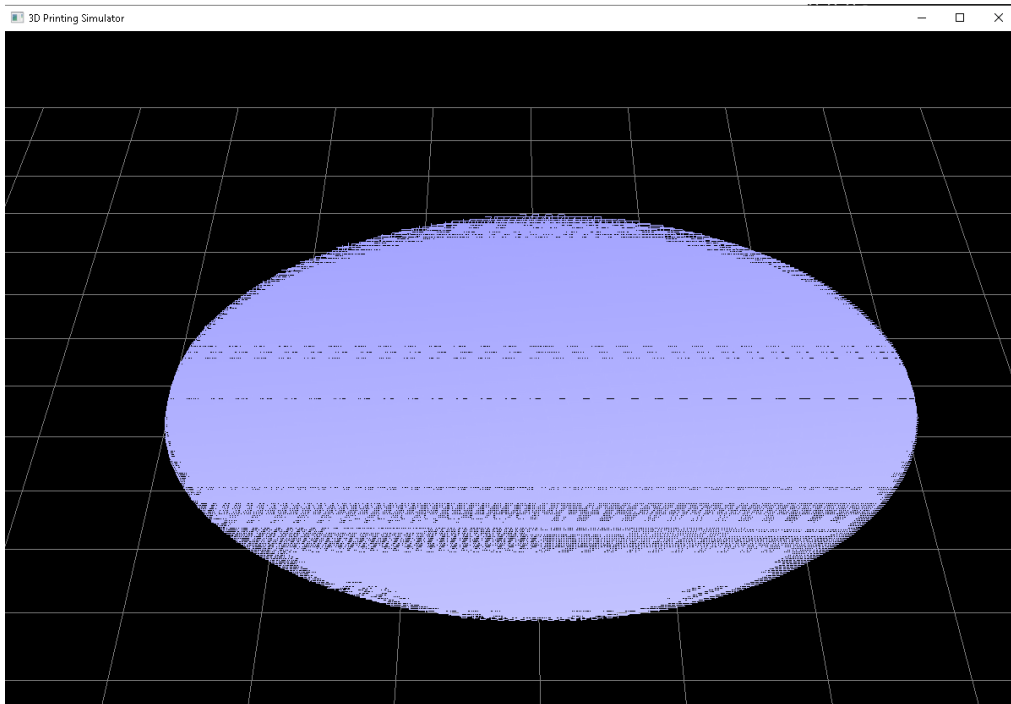


Figure 4.11: Infill of disk with layer height of 0.12 mm.

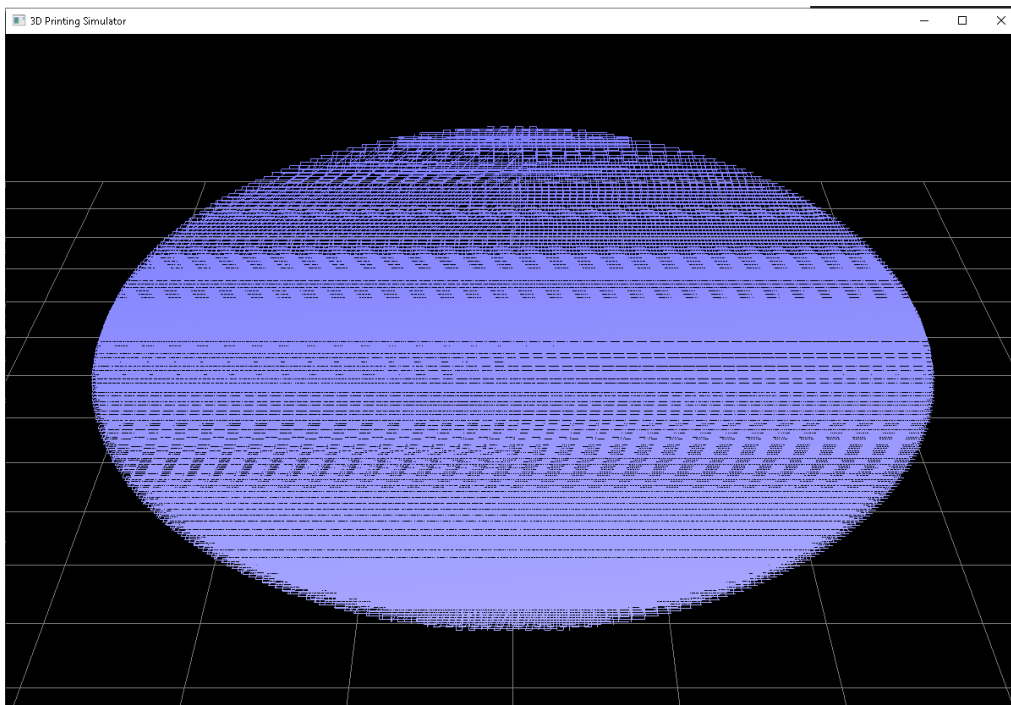


Figure 4.12: Infill of disk with layer height of 0.16 mm.

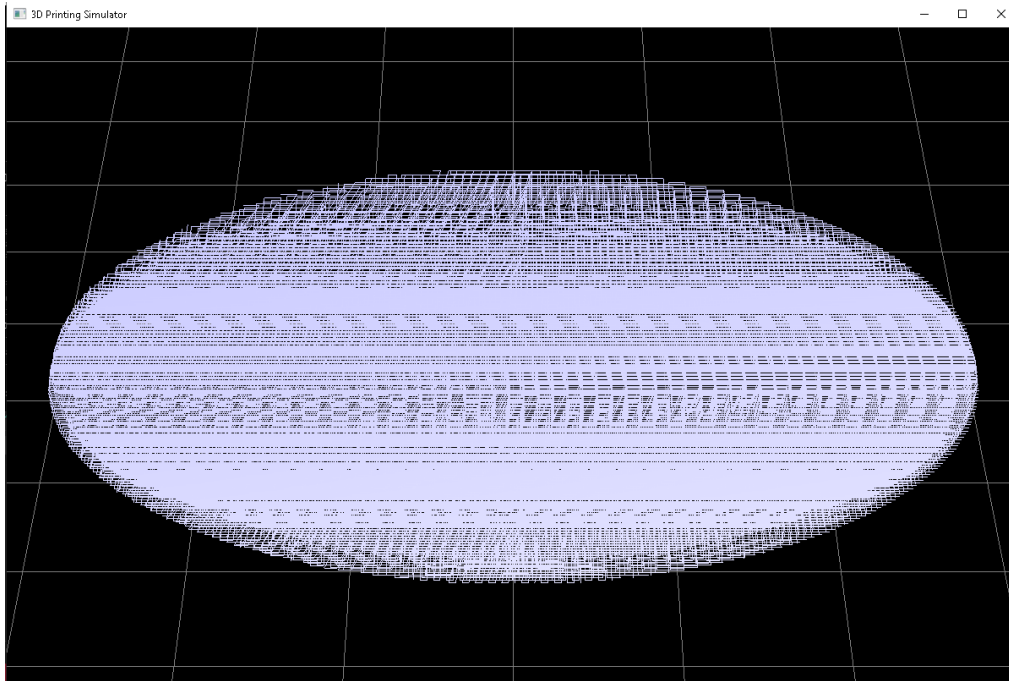


Figure 4.13: Infill of disk with layer height of 0.20 mm.

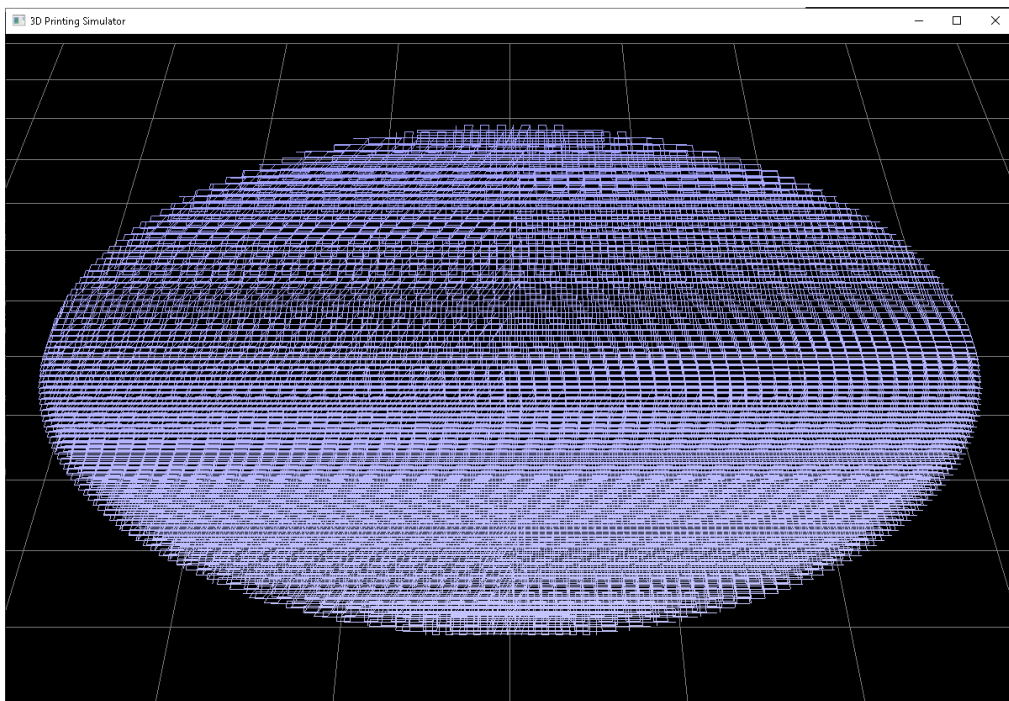


Figure 4.14: Infill of disk with layer height of 0.28 mm.

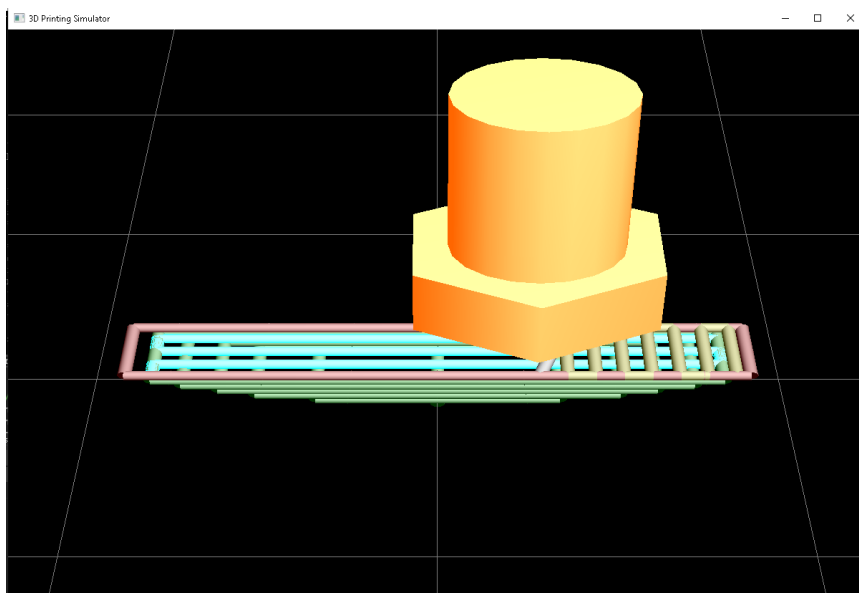


Figure 4.15: Section view of the disk.

### 4.3.2 Cylinder

The cylinder has diameter of 50 mm and thickness of 3 mm. It is similar with the disk as mentioned in previous section, but cylinder is lied on the printing platform instead of standing. The print parameters for cylinder are shown in. There are four cylinders to be printed with 7 different infill densities which are 10 %, 20 %, 30 %, 40 %, 50 %, 70 % and 90 %. Each infill density indicates the infill gap of 4 mm, 2 mm, 1.33 mm, 1 mm, 0.8 mm, 0.57 mm and 0.44 mm respectively.

Table 4.4: Print parameters for four cylinders.

STL file	Sample	Print Parameters				
		Layer Height	Infill Density	Infill Pattern	Top Thickness	Bottom Thickness
cylinder	5	0.28 mm	10 %	Linear	0.84 mm	0.84 mm
	6	0.28 mm	20 %	Linear	0.84 mm	0.84 mm
	7	0.28 mm	30 %	Linear	0.84 mm	0.84 mm
	8	0.28 mm	40 %	Linear	0.84 mm	0.84 mm
	9	0.28 mm	50 %	Linear	0.84 mm	0.84 mm
	10	0.28 mm	70 %	Linear	0.84 mm	0.84 mm
	11	0.28 mm	90 %	Linear	0.84 mm	0.84 mm

Table 4.5: Test result of the cylinders.

Sample	Infill Gap in Simulation	Actual Infill Gap	Number of Layers	Estimation Time	Actual Time	Error percentage
5	4 mm	4 mm	11	23 minutes	23 minutes	0 %
6	2 mm	2 mm	11	24 minutes	24 minutes	0 %
7	1.33 mm	1.33 mm	11	25 minutes	25 minutes	0 %
8	1 mm	1 mm	11	26 minutes	26 minutes	0 %
9	0.8 mm	0.8 mm	11	26 minutes	26 minutes	0 %
10	0.571 mm	0.571 mm	11	28 minutes	28 minutes	0 %
11	0.444 mm	0.444 mm	11	30 minutes	30 minutes	0 %

Figure 4.16 shows the cylinder that is printed with 10 % of infill density in the 3D printing simulation. The distance between each infill lines for 10 % infill density is 4 mm. Figure 4.17 shows the infill gap of 10 % infill density in actual printing. The infill gap of the actual printed cylinder with 10% infill density is  $10.4 \text{ cm} - 10.0 \text{ cm} = 0.4 \text{ cm}$  which is 4 mm. As a result, the actual printing matches with the simulated cylinder with 10 % infill density.

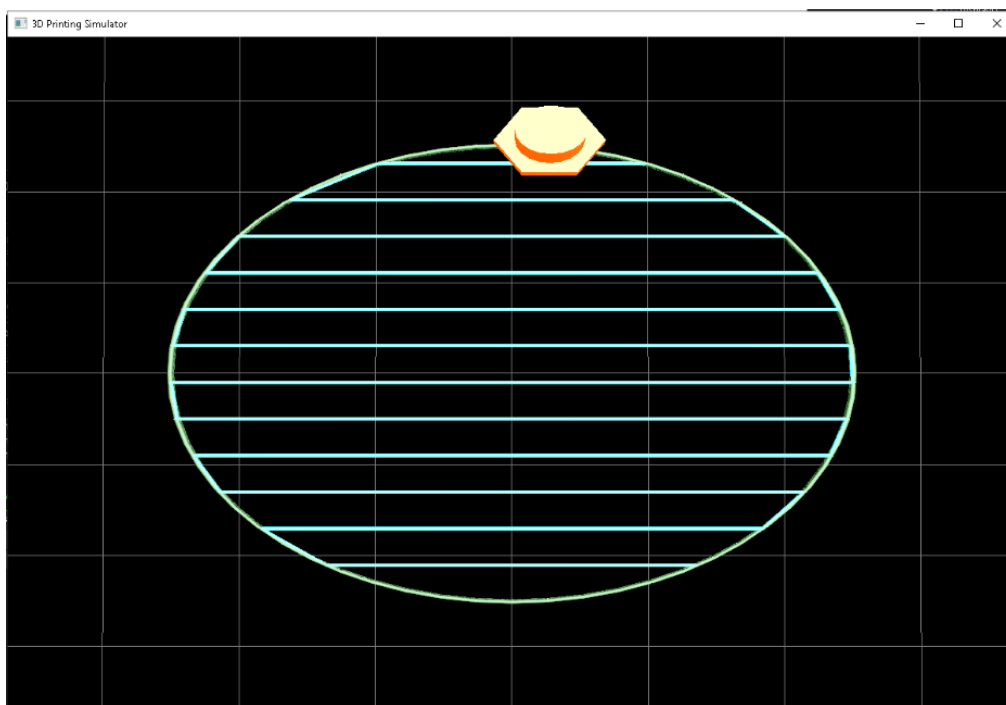


Figure 4.16: Cylinder with infill density of 10 %.

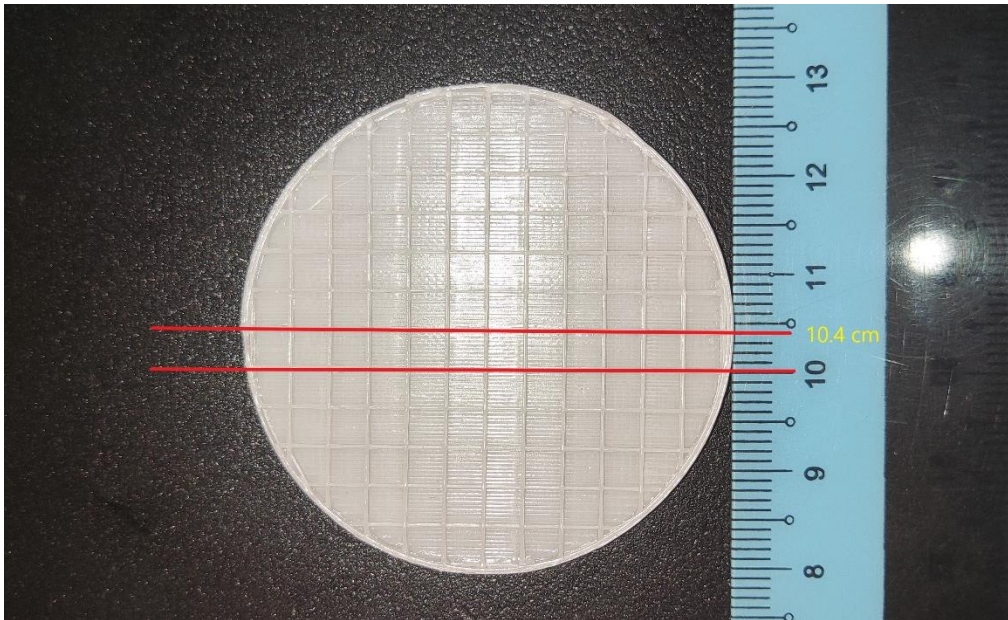


Figure 4.17: Measurement of 4 mm infill gap for 10 % infill density in actual printing.

Figure 4.18 shows the cylinder that is printed with 20 % of infill density in the 3D printing simulation. The distance between each infill lines for 20 % infill density is 2 mm. Figure 4.19 shows the infill gap of 20 % infill density in actual printing. The infill gap of the actual printed cylinder with 20% infill density is  $10.2 \text{ cm} - 10.0 \text{ cm} = 0.2 \text{ cm}$  which is 2 mm. As a result, the actual printing matches with the simulated cylinder with 20 % infill density.

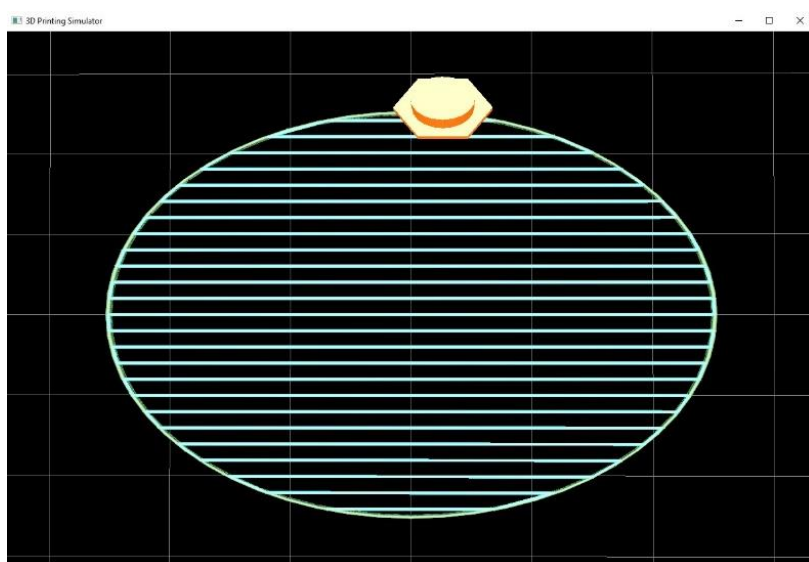


Figure 4.18: Cylinder with infill density of 20 %.



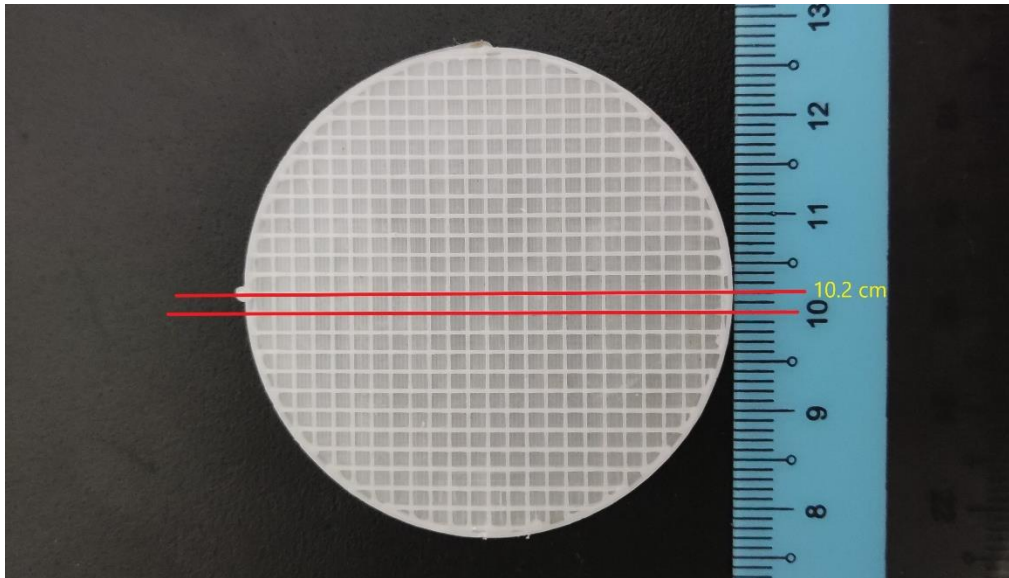


Figure 4.19: Measurement of 2 mm infill gap for 20 % infill density in actual printing.

Figure 4.20 shows the cylinder that is printed with 30 % of infill density in the 3D printing simulation. The distance between each infill lines for 30 % infill density is 1.33 mm. Figure 4.21 shows the infill gap of 30 % infill density in actual printing. The infill gap of the actual printed cylinder with 30% infill density is around 1.3 mm. As a result, the actual printing matches with the simulated cylinder with 30 % infill density.

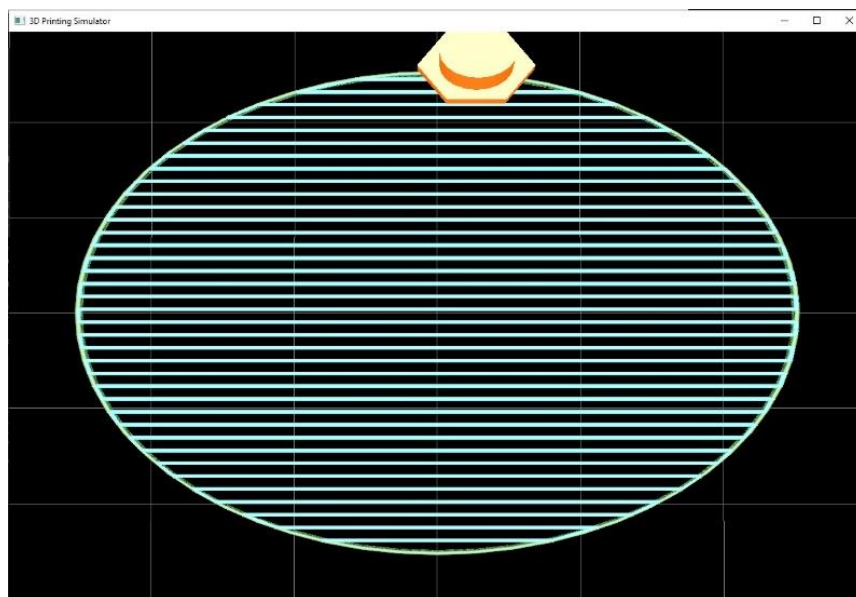


Figure 4.20: Cylinder with infill density of 30 %.

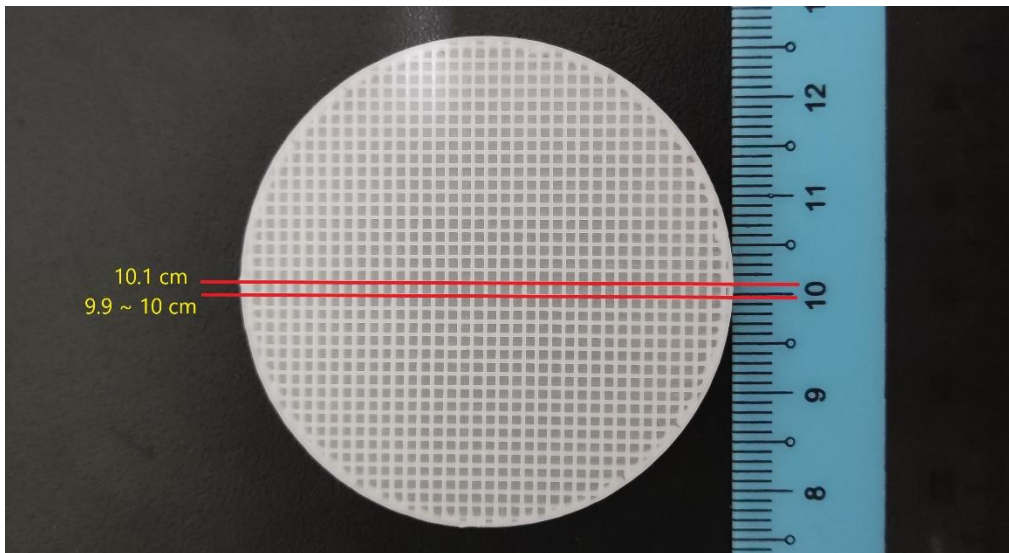


Figure 4.21: Measurement of 1.3 mm infill gap for 30 % infill density in actual printing.

Figure 4.22 shows the cylinder that is printed with 40 % of infill density in the 3D printing simulation. The distance between each infill lines for 40 % infill density is 1.0 mm. Figure 4.23 shows the infill gap of 40 % infill density in actual printing. The infill gap of the actual printed cylinder with 40% infill density is  $10.1 \text{ cm} - 10.0 \text{ cm} = 0.1 \text{ cm}$  which is 1.0 mm. As a result, the actual printing matches with the simulated cylinder with 40 % infill density.

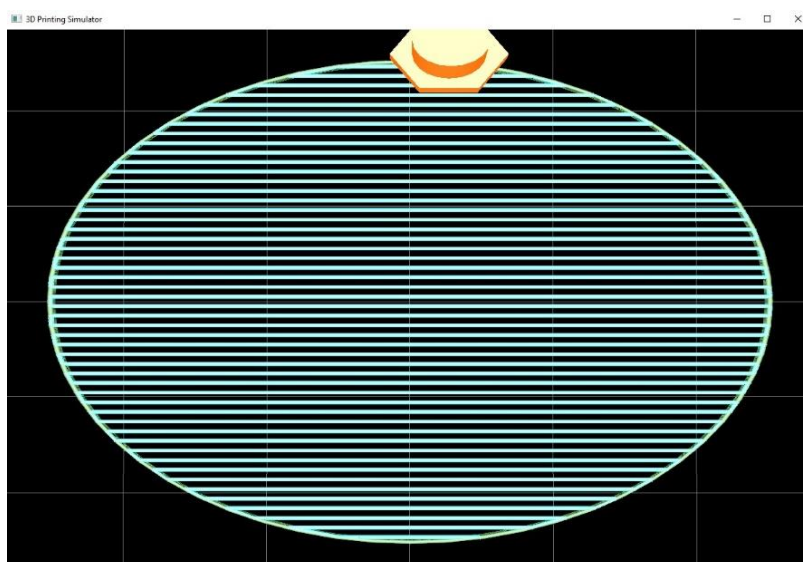


Figure 4.22: Cylinder with infill density of 40 %.

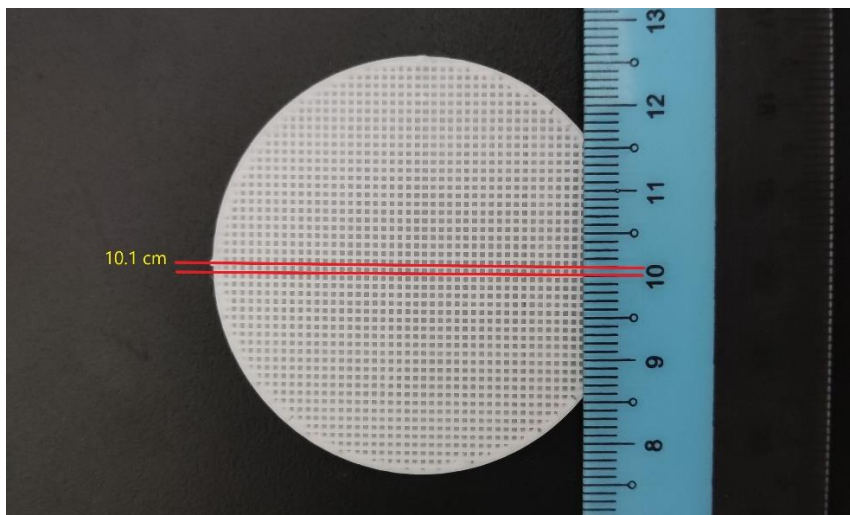


Figure 4.23: Measurement of 1 mm infill gap for 40 % infill density in actual printing.

Cylinder with infill density of 50 % is illustrated in Figure 4.24. The infill gap is 0.8 mm which is smaller than the smallest scale division of a ruler – 1 mm. Thus, due to lacking measuring equipment, G-code that obtained from ULTIMAKER CURA is used to examine the infill gap for subsequent cylinders. A third-party G-code viewer is used to visualize the generated G-code and extract the movement of the printer for infill. The horizontal infill gap of the actual printing is  $Y129.5 - Y128.7 = 0.8$  mm as shown in Figure 4.25. Hence, the infill gap obtained from the simulator with 50 % matches with the actual printing.

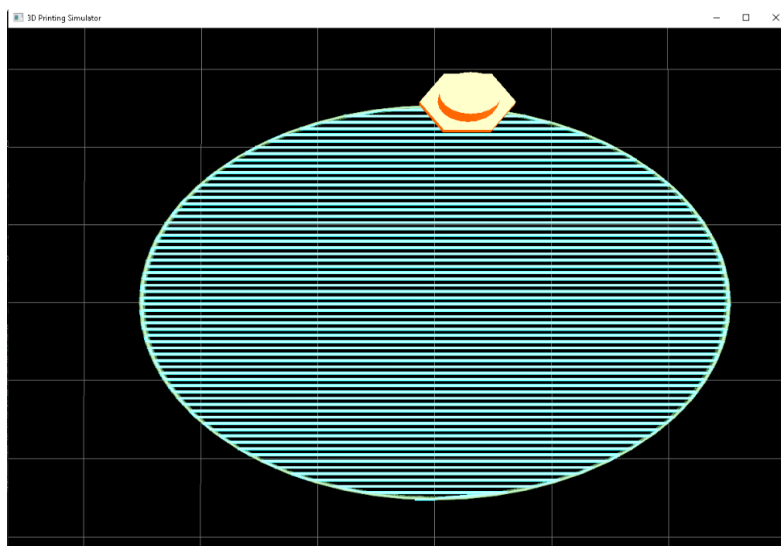


Figure 4.24: Cylinder with infill density of 50 %.

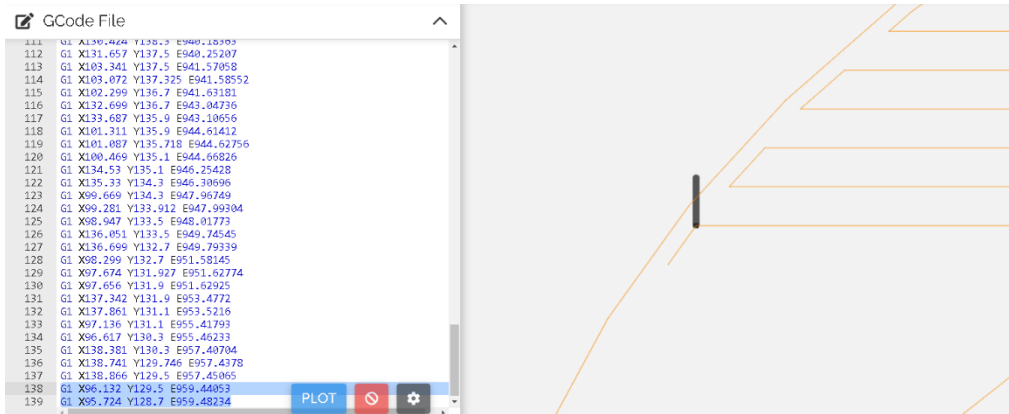


Figure 4.25: G-code viewer to verify actual infill gap for 50 % infill density.

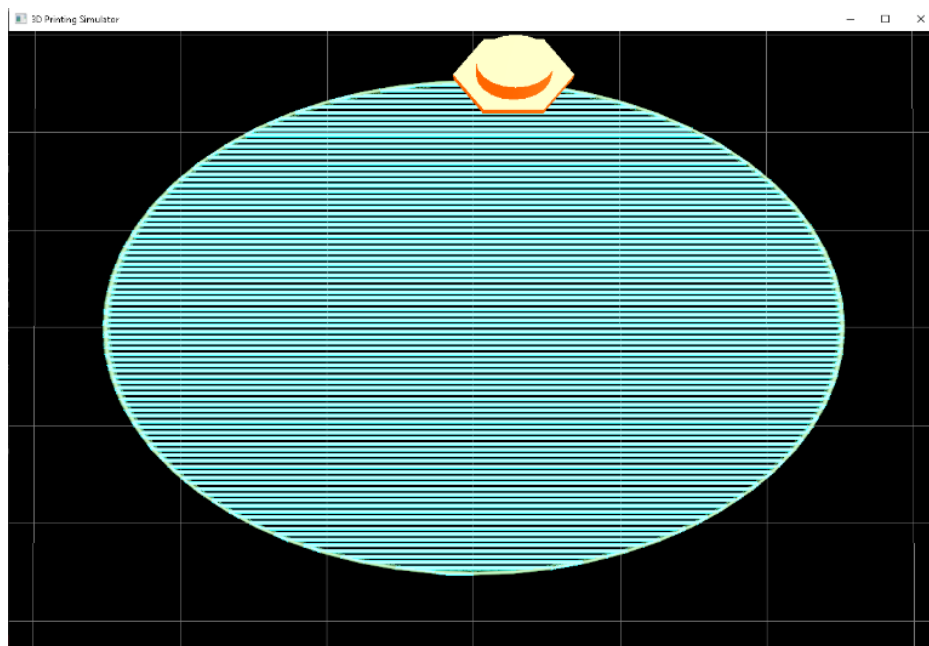


Figure 4.26: Cylinder with infill density of 70 %.

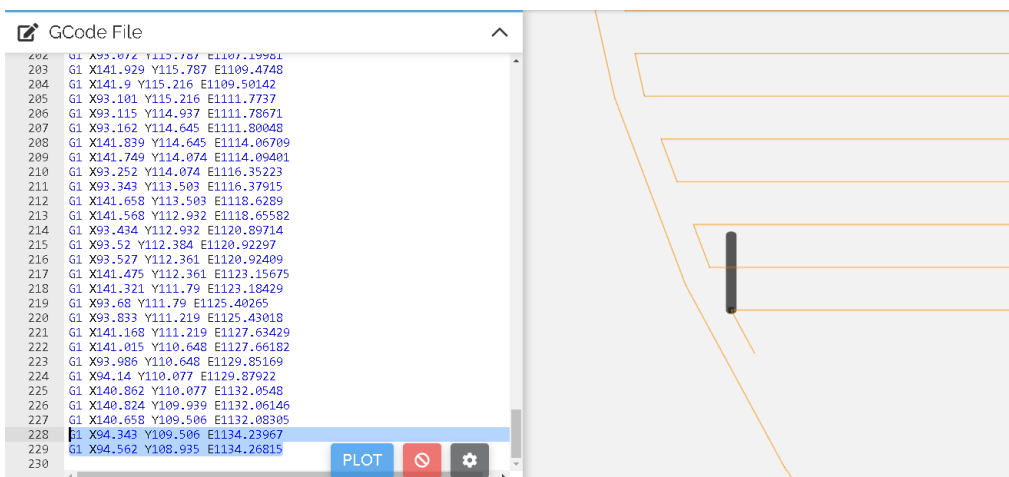


Figure 4.27: G-code viewer to verify actual infill gap for 70 % infill density.

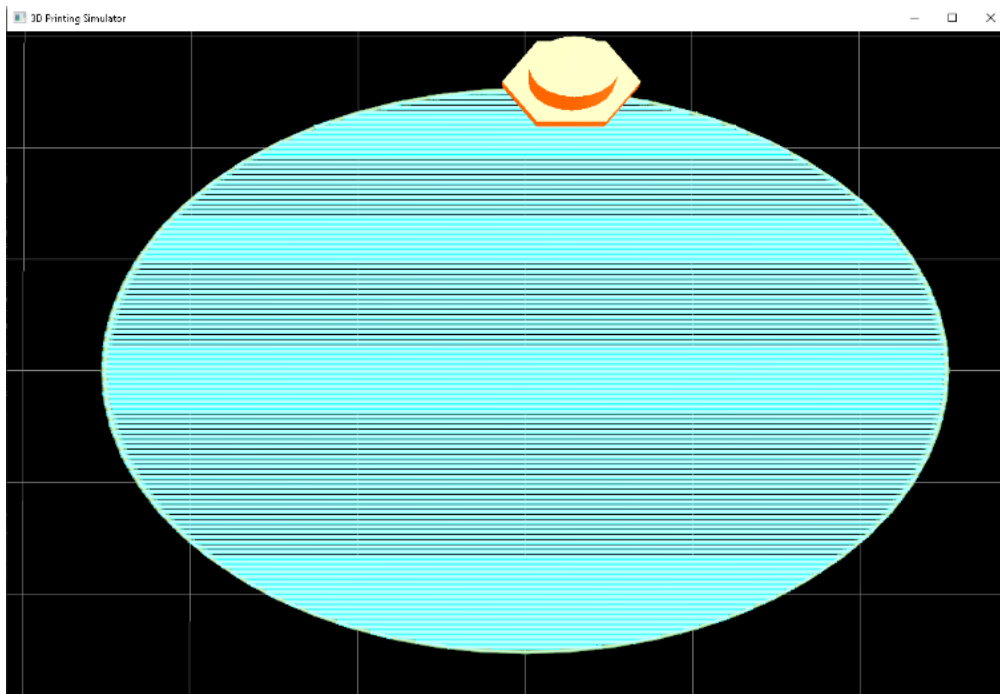


Figure 4.28: Cylinder with infill density of 90 %.

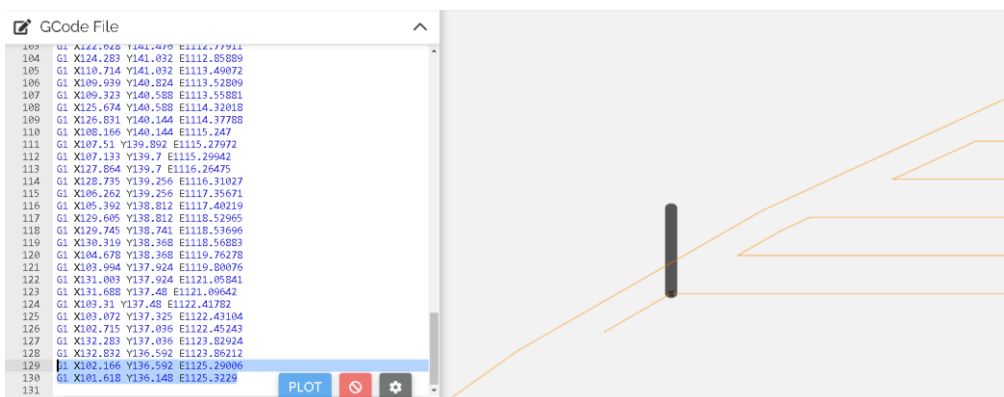


Figure 4.29: G-code viewer to verify actual infill gap for 90 % infill density.

Cylinder with infill density of 70 % is shown in Figure 4.26. The infill gap is 0.57 mm. According to the readings of the Y-axis movement in G-code, the horizontal infill gap of the actual printing is  $Y109.506 - Y108.935 = 0.571$  mm as shown in Figure 4.27. Hence, the infill gap obtained from the simulator with 70 % matches with the actual printing. Next, for infill density of 90% cylinder which infill gap is 0.444 mm. The G-code in Figure 4.29 shows horizontal infill with gap of  $Y136.592 - Y136.148 = 0.444$  mm. As a result, the simulator has capability of generating infill gap that matches with

real printing. Figure 4.30 shows the interior of the cylinder with infill density of 50 % and the infill changes direction on alternate layer.

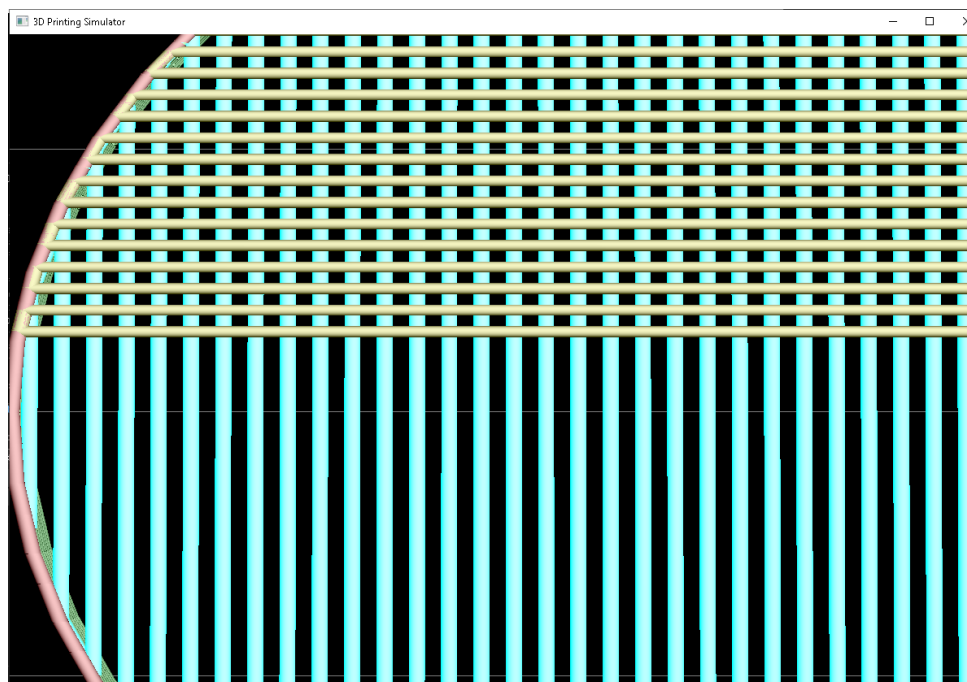


Figure 4.30: Infill changes direction on alternate layer.

#### 4.4 Cube

The dimension of the cubes is 30 mm x 30 mm x 30 mm. Four cubes are being printed to examine feasibility of different top and bottom thickness upon the 3D print object. The feasibility is examined by the difference of actual printing time and estimated printing time. Greater value of top and bottom thickness can lead to longer printing time with constant infill density of 50 % and layer height of 0.20 mm. The print parameters for four cubes are shown in Table 4.6. The result of printing time of Ender 3 and 3D printing simulation is shown in Table 4.7. Number of layers for each cube are 150 layers. The printing time error percentage for top and bottom thickness of 0.2 mm, 0.4 mm, 0.6 mm and 0.8 mm is 13.25 %, 15.12 %, 11.76 % and 11.49 %. The line gap for top and bottom layers is set to be 0.4 mm. G-code in Figure 4.34 shows that the line gap is  $X106.8 - X106.4 = 0.4$  mm. Figure 4.32 and Figure 4.33 shows the top and bottom layers of the cube from the simulator and real print respectively. Figure 4.31 shows the 3D printed cube that is rendered in the simulator.

Table 4.6: Print parameters for four cubes.

STL file	Sample	Print Parameters				
		Layer Height	Infill Density	Infill Pattern	Top Thickness	Bottom Thickness
cube	12	0.20 mm	50 %	Linear	0.20 mm	0.20 mm
	13	0.20 mm	50 %	Linear	0.40 mm	0.40 mm
	14	0.20 mm	50 %	Linear	0.60 mm	0.60 mm
	15	0.20 mm	50 %	Linear	0.80 mm	0.80 mm

Table 4.7: Test result of the cubes.

Sample	Number of Layers	Estimation Time	Actual Time	Error percentage
12	150	1 hour 12 minutes	1 hour 23 minutes	13.25 %
13	150	1 hour 13 minutes	1 hour 26 minutes	15.12 %
14	150	1 hour 15 minutes	1 hour 25 minutes	11.76 %
15	150	1 hour 17 minutes	1 hour 27 minutes	11.49 %

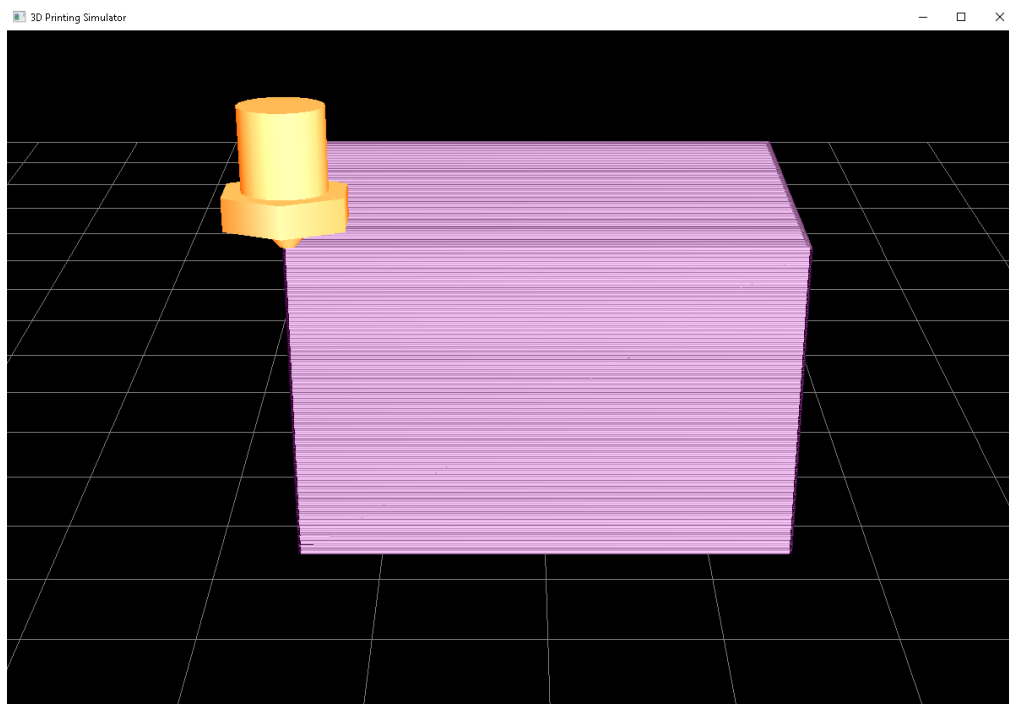


Figure 4.31: Cube is rendered in simulator with layer height of 0.20 mm.

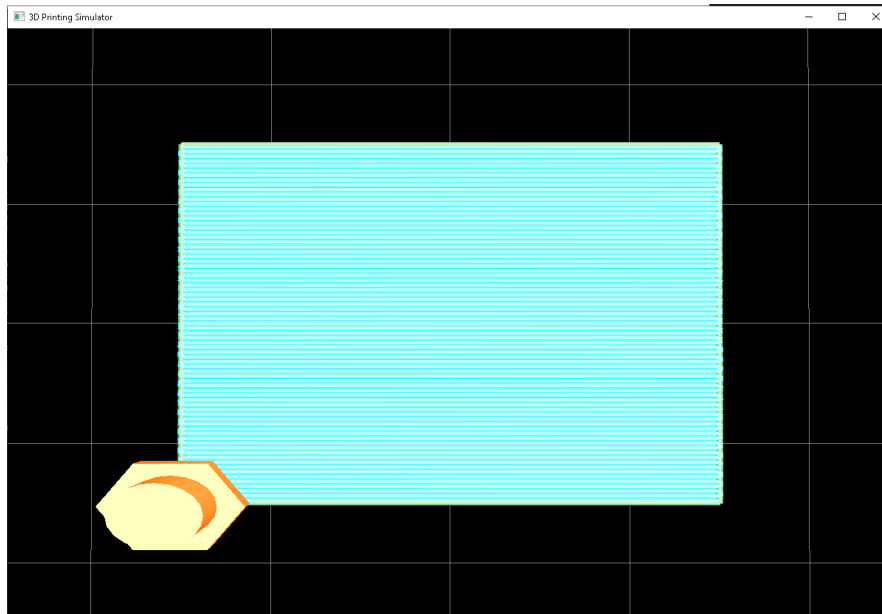


Figure 4.32: Top and bottom layers of the cube in simulator.

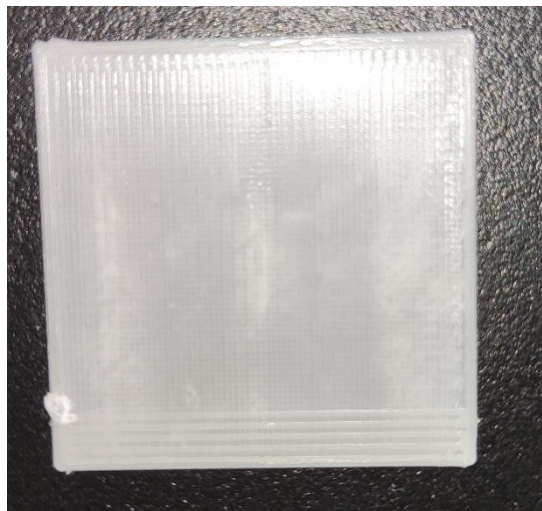


Figure 4.33: Actual top and bottom layers of the cube from 3D print.

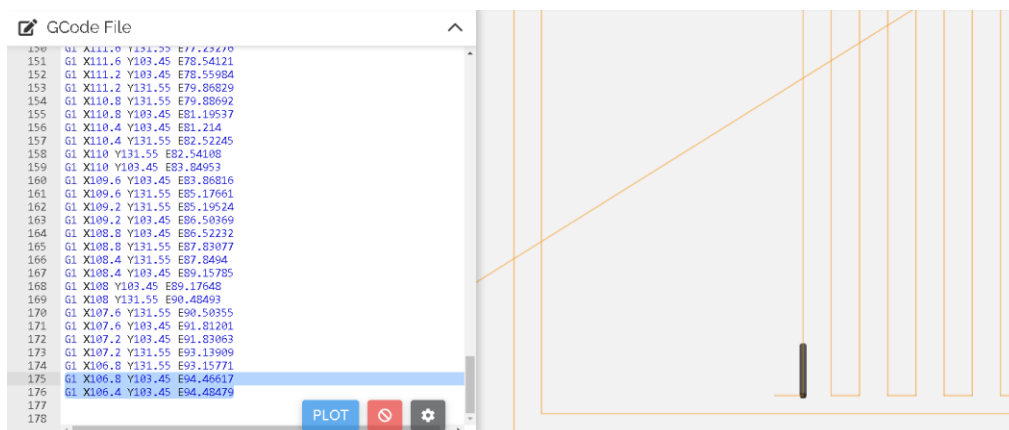


Figure 4.34: G-code viewer to verify the real line gap of top and bottom layer.



#### 4.4.1 Bracket

The simulator sliced the bracket into 39 layers with the layer height of 0.28 mm. The estimated printing time for this print is 10 minutes and the actual printing time is 10 minutes as well. This makes the error percentage of printing time between estimated time and actual time to be 0%. The print parameters are set in accordance with Table 4.8. Figure 4.35 shows front view of the bracket in the simulator. Isometric view of the 3D print bracket's shell in simulator is illustrated in Figure 4.36. Interior geometric of the bracket is displayed in Figure 4.37. The capability of CY simulator to render a lifelike 3D print object can be examined through the comparison between Figure 4.35 and Figure 4.38. The strength of the simulator upon rendering lifelike object is verified as the bracket generated in simulator is similar with the actual printed bracket.

Table 4.8: Print parameters for the bracket.

STL file	Sample	Layer height	Infill density	Infill pattern	Top thickness	Bottom thickness
bracket	16	0.28 mm	50 %	Linear	0.84 mm	0.84 mm

Table 4.9: Test result of the bracket.

Sample	Number of Layers	Estimated Time	Actual time	Error percentage
16	39	10 minutes	10 minutes	0 %

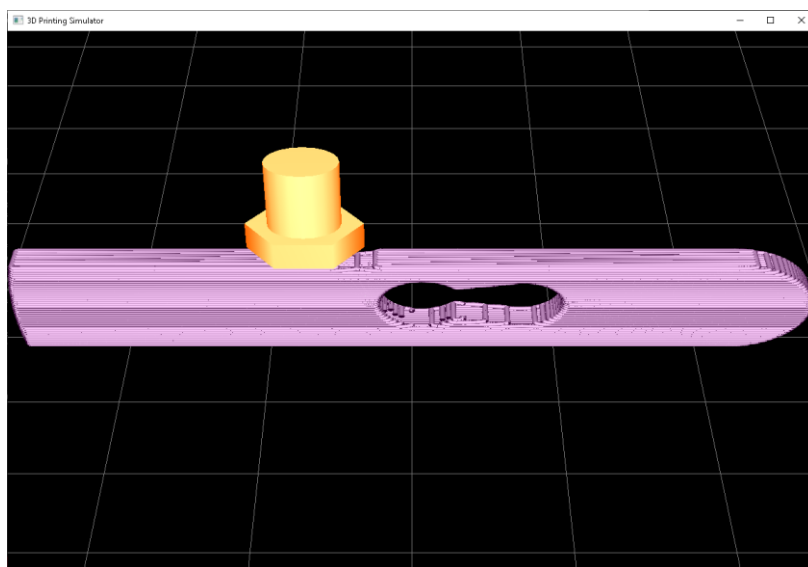


Figure 4.35: View of the bracket in simulator.

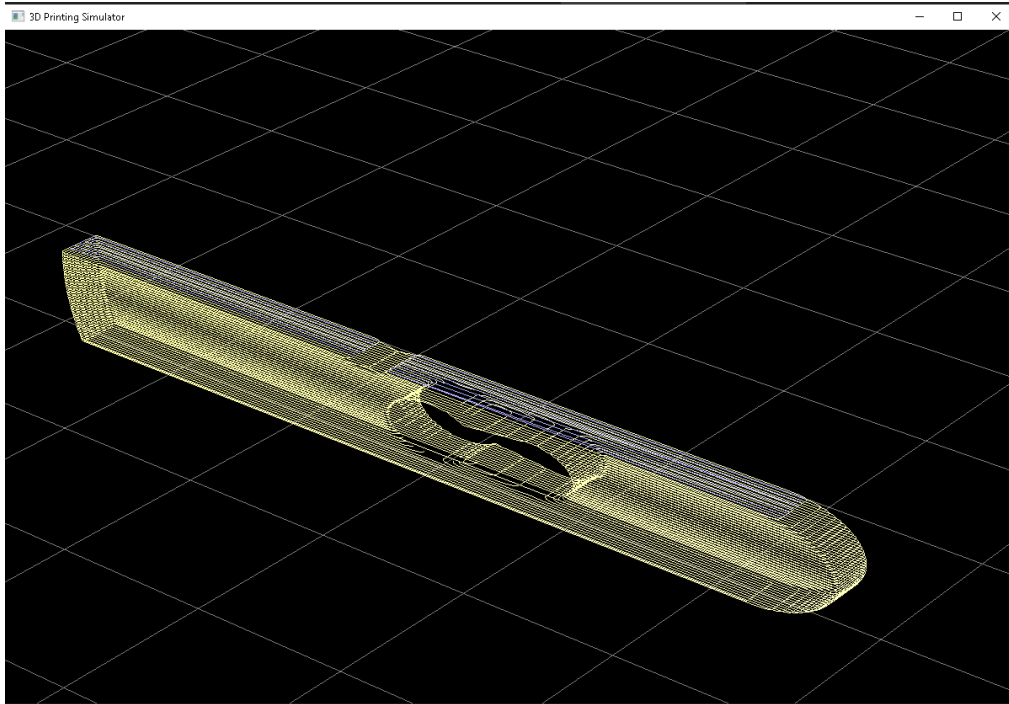


Figure 4.36: Isometric view of the bracket's shell.

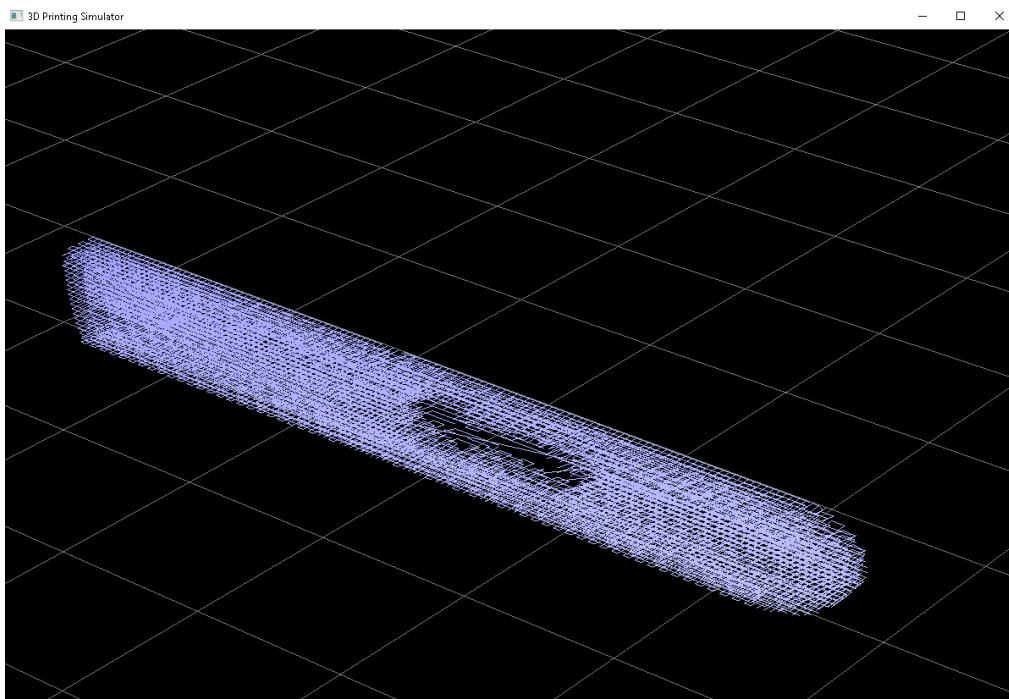


Figure 4.37: Isometric view of the bracket's infill.



Figure 4.38: Actual 3D printed bracket.

#### 4.4.2 Spoon

A spoon is sliced into 289 layers with the layer height of 0.12 mm. According to the estimated printing time in the simulator for this spoon is 1 hour and 14 minutes which has 7 minutes different with the actual printing time – 1 hour and 21 minutes. Error percentage of printing time between estimated time and actual time to be 8.64 % which equivalent to difference of 7 minutes. The print parameters for the spoon are set based on the Table 4.10. Figure 4.39 shows solid body presentation of the imported STL file in the simulator. Presentation of the simulation for 3D printed spoon is stated in Figure 4.40. Shell and infill of the spoon in the simulator is presented in Figure 4.41 and Figure 4.42 respectively. The spoon is printed by using Ender 3 printer with the generated G-code from the software - ULTIMAKER CURA. The spoon is printed according to the print parameters on Table 4.10. Figure 4.43 shows the 3D printed spoon.

Table 4.10: Print parameters for the spoon.

STL file	Sample	Layer height	Infill density	Infill pattern	Top thickness	Bottom thickness
spoon	17	0.12 mm	20 %	Linear	1.12 mm	1.12 mm

Table 4.11: Test result of the spoon.

Sample	Number of Layers	Estimated Time	Actual time	Error percentage
17	289	1 hour 14 minutes	1 hour 21 minutes	8.64 %

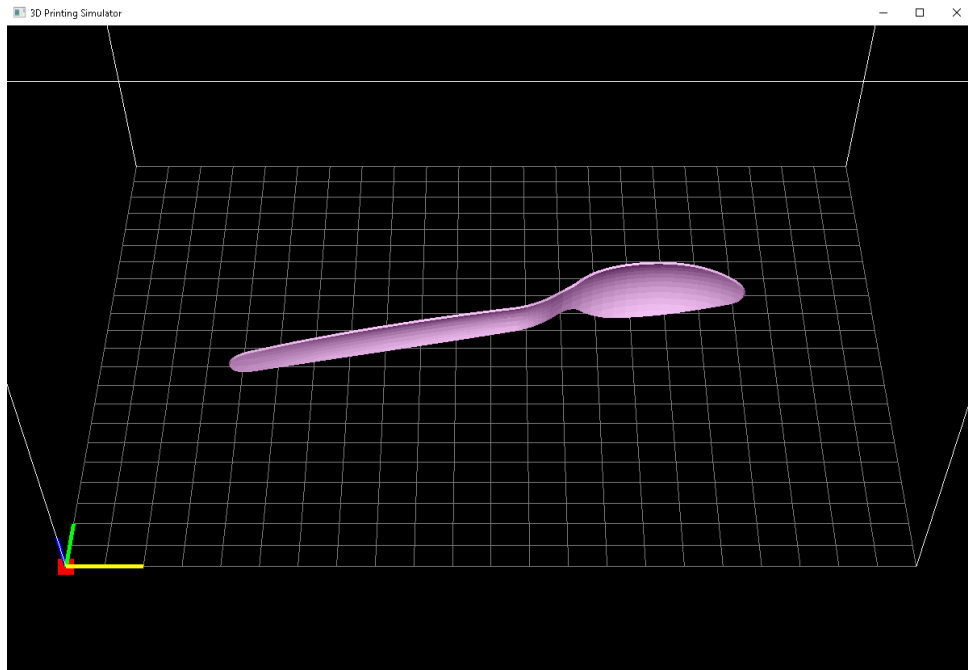


Figure 4.39: Spoon is rendered in solid body mode.

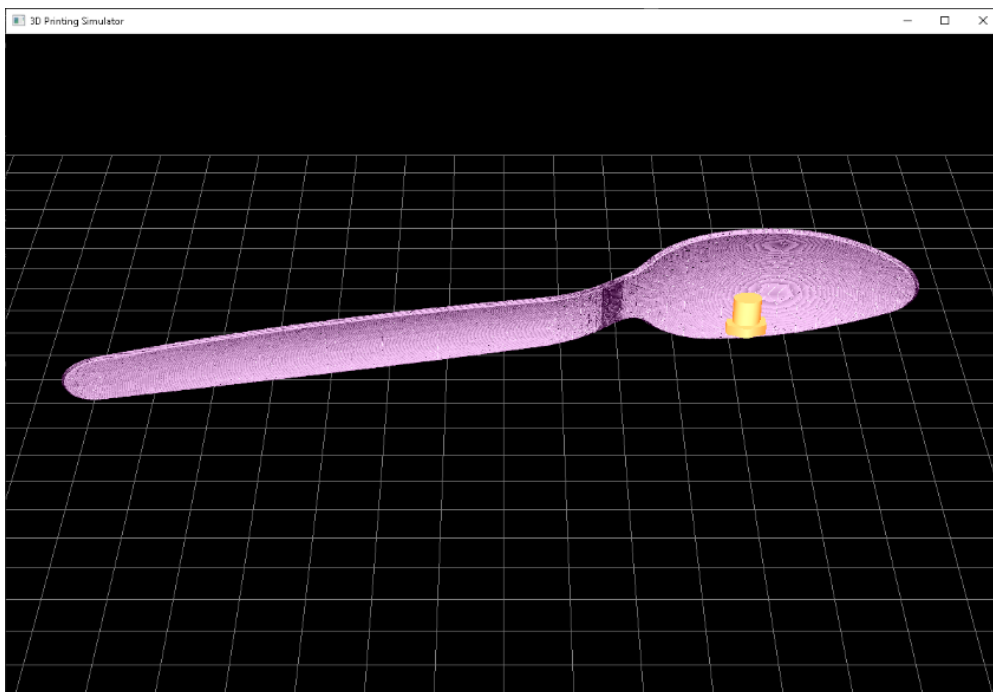


Figure 4.40: Appearance of the 3D printed spoon in the simulator.

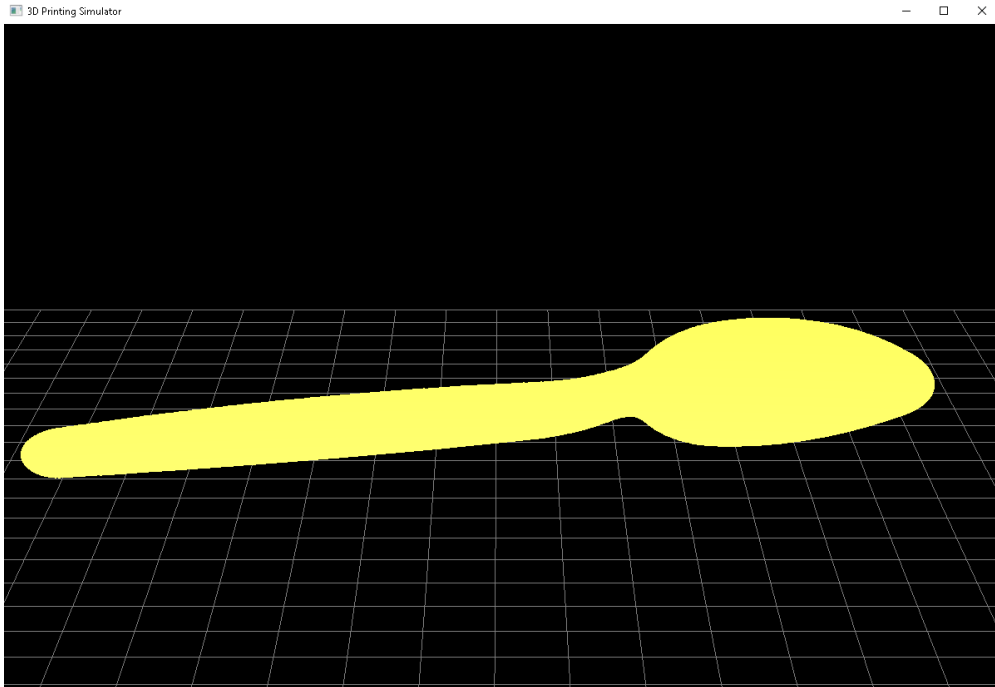


Figure 4.41: Shell of the spoon in the simulator.

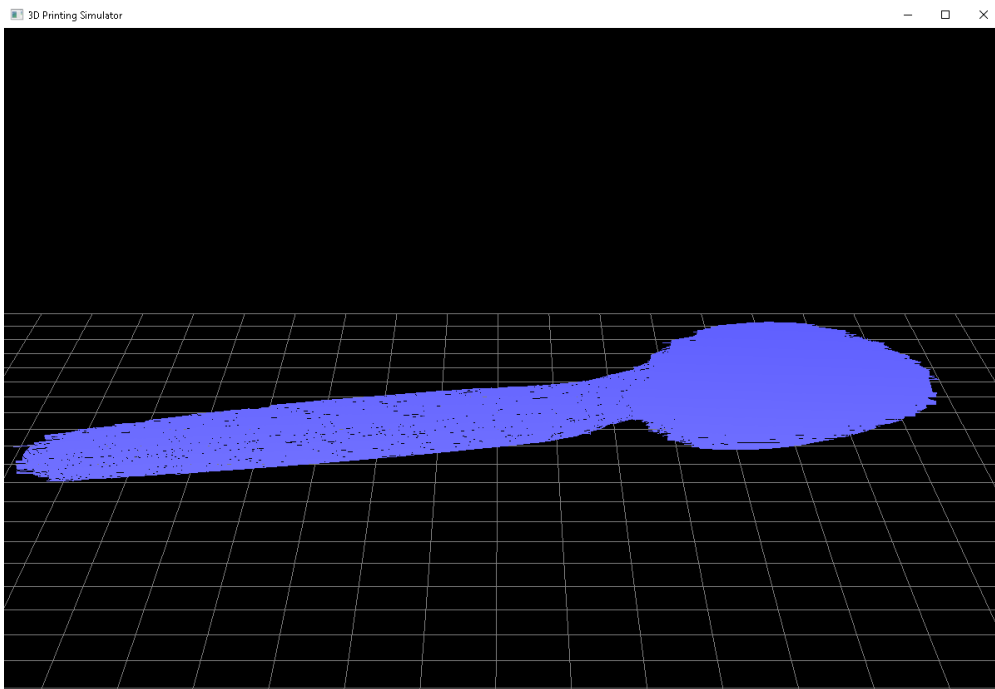


Figure 4.42: Infill of the spoon in the simulator.



Figure 4.43: Actual 3D printed spoon.

#### 4.5 Summary

The framework of the simulator is tested with five different 3-D models which are disk, cylinder, cube, bracket and spoon. The implementation of print parameters in the simulator are compared with the actual printing models. All infill line gap generated by the simulator have 100 % accuracy with the actual printing results. Apart from that, number of layers of a 3D print affects the accuracy of estimating printing time. Accuracy of printing time estimation can be high if number of layers is low. Otherwise, lower accuracy upon the time estimation for 3D printing if number of layers for the 3D print is high. The reason of the inaccuracy is due to the negligence of travelling time for an actual printing from the algorithm. The travelling path that excluded from the algorithm is the trajectory that move from the last vertex of a layer to the first vertex of the next layer. Henceforth, more travelling time is not taken into consideration by the printing time estimation algorithm when encounter 3D print that is sliced into high number of layers.

## CHAPTER 5

### CONCLUSION AND RECOMMENDATIONS

#### 5.1 Conclusions

Development of 3D printing simulator in this project has succeeded with the capability of simulating the process of Fused Deposition Modelling (FDM) 3D printing technology. The framework of this simulation program is capable to read a stereolithography (STL) file in ASCII format. The raw data extracted from the STL file can be processed into layers' vertices for the usage of 3D printing. This generates the shell of a 3D print object with user-defined layer height that range from 0.12 to 0.28 mm. Users are free to tune the value of layer height for either studying the 3D print mechanism or prototyping.

Apart from that, interior of a 3D print can be generated in this 3D printing simulator program. The simulator has succeeded to generate infill line gap that is similar with the actual 3D printed model. Moreover, skin of a 3D model can be generated successfully to enclose the interior of a 3D print model. In this program, users may input percentage of infill density to analyse the 3D print model. They may alter the top and bottom thickness that suitable for their application.

Furthermore, the program has succeeded to simulate the 3D printing process layer by layer in progressive manner. Shell, skin and infill of a 3D print can be simulated in pipe form manner to represent FDM printing filament. On top of that, this simulator provides an option of switching pipe form rendering to line form presentation for the shell, skin and infill. This is implemented to increase the speed of 3D printing simulation due to lesser computational cost. Last but not least, a nozzle is constructed in the simulator to actualize 3D printing process. The simulator has the ability of estimating printing time for users' reference.

#### 5.2 Contributions

One of the contributions for this simulator is implementation of slicing algorithm in C++ programming language. The slicing algorithm is able to slice

any 3D models into layers with the user-defined layer gap. Besides that, a contour creation function is written after the developed slicing algorithm to organize the vertices obtained. Contour creation is essential to ascertain the feasibility of the slice vertices. Without contour creation, vertices obtained from slicing algorithm are mere scattered points that cannot be used to form a 3D print's shell.

Moreover, the method of generating two infill patterns: line infill pattern and linear infill pattern. The idea to generate these two patterns is to control the arrangement in the infill array. Infill generation and slicing algorithm is alike. Slicing algorithm is to slice a 3D print object vertically while infill detection algorithm is to compute infill vertex horizontally. Slicing plane for infill is changed on alternate layers to form vertical and horizontal infill. Apart from that, the idea of generating the skin of a 3D print object is similar with infill generation. The skin is different from infill as the line gap of skin is fixed at 0.4 mm. The skin generation algorithm is built with the infill algorithm but with a condition that if the layers is below the user-defined bottom thickness then the line gap is fixed at 0.4 mm. Otherwise, the line gap is based on the user-defined infill density to generate infill. This is applied to forming of top layers as well.

Furthermore, each vertex obtained from contour creation and infill generator is linked by cylinder. The cylinder is extruded from a vertex to the next vertex by 0.0125 mm to simulate the process of printing. On top of that, dedication in developing another rendering mode – line rendering is to reduce the computational cost. Users may use line mode rendering to have a quick view on the printing process. This line rendering mode also helps me to validate the correctness of the generated vertices from slicing algorithm, contour creation, infill detection and skin formation.

The printing time is estimated by computing the distance between each vertex and defined printing speed. The idea of implementing this feature is for users to have a sense on the printing time of the 3D print object so that users cannot control the printing time instead of mere architecture of a 3D print model. Last but not least, several printings are carried out to ascertain the print parameters in the simulator is matched with the real-world print.



### 5.3 Recommendations for future work

There are several recommendations can be implemented to the simulator to enrich this 3D printing simulator. First and foremost, development of multiple shell walls generator should be executed in future work. This is because wall thickness is considered as one of the crucial print parameters to affect the strength of a 3D print. Hobbyist and users of 3D printing would prefer to adapt 3D printing simulator with the function of varying wall thickness. Hence, it would be valuable to be implemented for future work.

Apart from that, one of the limitations for this 3D printing simulator is that it is limited to non-hollow object. Most of the objects would be a hollow object with at least one hole for the design. Hence, a 3D printing simulator with the feature of generating infill in a hollow object would be favourable. Moreover, a simulator could provide variation of infill pattern such as cubic, triangles, grid and concentric pattern. This is due to the reason that different application will be built with different infill pattern. Line and linear infill pattern that equipped in the simulator of this project is not ample to fulfil the market needs. Hence, multifarious infill patterns would escalate the competitiveness of this simulator.

On top of that, a critical feature of a 3D printing simulator is generating support structure to avoid overhangs. This is because a 3D object is produced with an FDM 3D printer by depositing thermoplastics layer by layer. As a result, every new layer needs to be supported by the layer below it. There is also a considerable risk that the model will drop or even topple if it has an overhang that is not supported by anything below, thus additional 3D printed support structures is needed to guarantee a successful print. A very slight horizontal offset (barely perceptible) is used by 3D printers between successive layers. A layer therefore stacks with a slight offset rather than exactly over the layer underneath it. As a result, overhangs that don't tilt too much from the vertical can be printed by the printer. The preceding layers can tolerate an angle less than 45 degrees. The line of failure is generally accepted to lie at 45 degrees. The previous layer cannot support an angle greater than 45 degrees. Consequently, a support structure will be crucial to reduce overhangs.

Hence, support structures must be included in a 3D printing simulator to consummate a 3D printing process simulation.

## REFERENCES

- Aaryaman Aashind, 2021. *3D Printing Statistics for 2022*. [online] Available at: <https://cybercrew.uk/blog/3d-printing-statistics/> [Accessed 12 April 2022].
- Adnan, F. A., Romlay, F. R. M. and Shafiq, M., 2018. Real-time slicing algorithm for Stereolithography (STL) CAD model applied in additive manufacturing industry. *IOP Conference Series: Materials Science and Engineering*, [e-journal] 342, p. 12016–12016. <http://dx.doi.org/10.1088/1757-899X/342/1/012016>.
- Ahn, S. H., 2019. *OpenGL Cylinder, Prism & Pipe*. [online] Available at: [http://www.songho.ca/opengl/gl\\_cylinder.html#pipe](http://www.songho.ca/opengl/gl_cylinder.html#pipe) [Accessed 24 April 2022].
- Benjamin Vaissier, 2022. *Simulations in 3D printing | Hubs*. [online] Available at: <https://www.hubs.com/knowledge-base/simulations-3d-printing/#intro> [Accessed 14 April 2022].
- Brown, A. C. and Beer, D. de, 092013. Development of a stereolithography (STL) slicing and G-code generation algorithm for an entry level 3-D printer. In: *2013 Africon. AFRICON 2013*. Pointe-Aux-Piments, Mauritius, 9/9/2013 - 12/9/2013: IEEE, pp. 1–5.
- Chang Y., S. R., 1986. *Using SLAM to design the material handling system of a flexible manufacturing system*. *International journal of Production Research*, Vol.24, 15-26.A
- Chen, Z., 102010. Development of OpenGL Based 3D Simulator for Computer Numerical Control. In: *2010 International Conference on Artificial Intelligence and Computational Intelligence. 2010 International Conference on Artificial Intelligence and Computational Intelligence (AICI)*. Sanya, China, 23/10/2010 - 24/10/2010: IEEE, pp. 319–321.
- Choong, Y. Y. C., Tan, H. W., Patel, D. C., Choong, W. T. N., Chen, C.-H., Low, H. Y., Tan, M. J., Patel, C. D. and Chua, C. K., 2020. The global rise of 3D printing during the COVID-19 pandemic. *Nature reviews. Materials*, [e-journal] 5(9), pp. 637–639. <http://dx.doi.org/10.1038/s41578-020-00234-3>.
- Christine Evans, 2021. *6 Industries Being Transformed by 3D Printing | Fictiv*. [online] Available at: <https://www.fictiv.com/articles/6-industries-being-transformed-by-3d-printing> [Accessed 12 April 2022].
- Dave Shreiner, 2001. *Performance OpenGL: Platform Independent Techniques*.
- Douglas, K., 2021. 3D Printer Heated Bed – The Advantages. *All3DP*. [online] 28 Jul. Available at: <https://all3dp.com/2/3d-printer-heated-bed-advantages/> [Accessed 15 April 2022].

ElMaraghy, H. A., 1982. *Simulation and graphical animation of advanced manufacturing systems*. Journal of Manufacturing Systems, 53- 63.

G. Avventuroso, R. Foresti, M. Silvestri, E. Morosini Frazzon, ed., 2017. *"Engineering, technology & innovation management beyond 2020: new challenges, new approaches": 2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC) : conference proceedings*. Piscataway, NJ: IEEE. Available at: Jardim-Gonçalves, Ricardo (HerausgeberIn). <http://ieeexplore.ieee.org/servlet/opac?punumber=8269762> .

Gopsill, J. A., Shindler, J. and Hicks, B. J., 2018. Using finite element analysis to influence the infill design of fused deposition modelled parts. *Progress in Additive Manufacturing*, [e-journal] 3(3), pp. 145–163. <http://dx.doi.org/10.1007/s40964-017-0034-y>.

Hajihosseini, F. H., 2009. *Importance of Simulation in Manufacturing*. World Academy of Science, Engineering and Technology, 285-288.

Jardim-Gonçalves, R., ed., 2017. *"Engineering, technology & innovation management beyond 2020: new challenges, new approaches": 2017 International Conference on Engineering, Technology and Innovation (ICE/ITMC) : conference proceedings*. Piscataway, NJ: IEEE. Available at: Jardim-Gonçalves, Ricardo (HerausgeberIn). <http://ieeexplore.ieee.org/servlet/opac?punumber=8269762>.

Jee, H. and Sachs, E., 2000. A visual simulation technique for 3D printing. *Advances in Engineering Software*, [e-journal] 31(2), pp. 97–106. [http://dx.doi.org/10.1016/S0965-9978\(99\)00045-9](http://dx.doi.org/10.1016/S0965-9978(99)00045-9).

*Low Cost 3D Printing for Rapid Prototyping and its Application* - Scientific Figure on ResearchGate. Available from: [https://www.researchgate.net/figure/Workflow-of-3D-Printing-Process\\_fig1\\_338600152](https://www.researchgate.net/figure/Workflow-of-3D-Printing-Process_fig1_338600152) [accessed 14 Apr, 2022]

Lo Valvo, E., Licari, R. and Adornetto, A., 2012. CNC Milling Machine Simulation in Engineering Education. *International Journal of Online and Biomedical Engineering (iJOE)*, [e-journal] 8(2), p. 33–33. <http://dx.doi.org/10.3991/ijoe.v8i2.2047>.

Luo, X., Wang, J., Liu, N., Zhao, Z. and Zhou, Y., 2014. YaRep: A Personal 3D Printing Simulator. In: *2014 International Conference on Virtual Reality and Visualization. 2014 International Conference on Virtual Reality and Visualization (ICVRV)*. Shenyang, China, 30/8/2014 - 31/8/2014: IEEE, pp. 408–411.

Materialise Software, 2020. *Tutorial: The 3D printing workflow*. 11 November. Available at: <https://youtu.be/6ZO4YPgoAGM> (Accessed: 14 April 2022).

Manus, 2015. *BENEFITS OF USING CNC SIMULATION SOFTWARE*. [online] Available at: <https://www.manusnc.com/en/blog/benefit-of-using-cnc-simulation-software> [Accessed 14 April 2022].

Marshall Burns, 1989. *The StL Format* | *fabbers.com*. [online] Available at: <[https://www.fabbers.com/tech/STL\\_Format](https://www.fabbers.com/tech/STL_Format)> [Accessed 15 April 2022].

Minetto, R., Volpato, N., Stolfi, J., Gregori, R. M. and da Silva, M. V., 2017. An optimal algorithm for 3D triangle mesh slicing. *Computer-Aided Design*, [e-journal] 92, pp. 1–10. <http://dx.doi.org/10.1016/j.cad.2017.07.001>.

Mohan Pandey, P., Venkata Reddy, N. and Dhande, S. G., 2003. Slicing procedures in layered manufacturing: a review. *Rapid Prototyping Journal*, [e-journal] 9(5), pp. 274–288. <http://dx.doi.org/10.1108/13552540310502185>.

Pal, B., 2021. FDM Printing Advantages & Disadvantages | Detailed. *TheMechNinja*. [online] 2 Jul. Available at: <https://themechninja.com/07/fdm-printing-advantages-disadvantages-detailed/>[Accessed 15 April 2022].

Pan, X., Chen, K. and Chen, D., 2014. Development of rapid prototyping slicing software based on STL model. In: *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD). Hsinchu, Taiwan, 21/5/2014 - 23/5/2014: IEEE, pp. 191–195.

Rohrer, M. W., 2000. Seeing is believing: the importance of visualization in manufacturing simulation. In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*. WSC 2000, Winter Simulation Conference. Orlando, FL, USA, 10-13 Dec. 2000: IEEE, pp. 1211–1216.

Sam Anand, Omkar Ghalsasi, Botao Zhang, Archak Goel, Srikanth Reddy, Shriyanka Joshi and Gil Morris, 2018. *Peace engineering: Transforming engineers for a sustainable global future : imagine - design - create : building a better world through peace engineering : November 12-16, 2018, Albuquerque, NM, USA*. [e-book]. Piscataway, NJ: IEEE. <http://ieeexplore.ieee.org/servlet/opac?punumber=8622654>

Sagar Shinde, 2000. *Introduction to Simulation and Modeling: Historical Perspective*. [online] Available at: <https://uh.edu/~lcr3600/simulation/historical.html>[Accessed 10 April 2022].

Singh, P. and Dutta, D., 2003. *Multi-Direction layered deposition – An overview of process planning methodologies*, Proceedings of the Solid Freeform Fabrication Symposium.

Topçu O, Taşcıoğlu Y and Ünver H Ö., 2011. *A Method for Slicing CAD Models in Binary STL Format*. 6th Int. Adv. Technol. Symp. 141 – 8

Tractus3D, 2021. *FDM vs SLA 3D printing - What are the advantages and disadvantages*. [online] Available at: <https://tractus3d.com/knowledge/learn-3d-printing/fdm-vs-sla-3d-printing/>[Accessed 13 April 2022].

Wang, D., Wang, H. and Wang, Y., 2021. Continuity Path Planning for 3D Printed Lightweight Infill Structures. In: *2021 IEEE Conference on*

*Telecommunications, Optics and Computer Science (TOCS). 2021 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS). Shenyang, China, 10/12/2021 - 11/12/2021: IEEE, pp. 959–962.*

Wikifactory, 2021. *FDM 3D Printing: Common problems and how to solve them* by +bitfab. [online] Available at: <<https://wikifactory.com/+bitfab/stories/fdm-3d-printing-common-problems-and-how-to-solve-them>> [Accessed 15 April 2022]

Xu, H., Jing, W., Li, M. and Li, W., 102016. A slicing model algorithm based on STL model for additive manufacturing processes. In: *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). Xi'an, China, 3/10/2016 - 5/10/2016: IEEE, pp. 1607–1610.*

Xu, J., Gu, X., Ding, D., Pan, Z. and Chen, K., 2018. A review of slicing methods for directed energy deposition based additive manufacturing. *Rapid Prototyping Journal*, [e-journal] 24(6), pp. 1012–1025. <http://dx.doi.org/10.1108/RPJ-10-2017-0196>.

Zhang, Z. and Joshi, S., 2015. An improved slicing algorithm with efficient contour construction using STL files. *The International Journal of Advanced Manufacturing Technology*, [e-journal] 80(5-8), pp. 1347–1362. <http://dx.doi.org/10.1007/s00170-015-7071-9>.

3D Spectra Technologies LLP, 2021. *FDM 3D Printing Market Growth, Opportunities & Industry Sectors: Everything You Need to Know*. [online] Available at: <https://www.3dspectratech.com/fdm-3d-printing-market-growth-opportunities-industry-sectors-everything-you-need-to-know/> [Accessed 13 April 2022].

## APPENDICES

### Appendix A: ULTIMAKER CURA

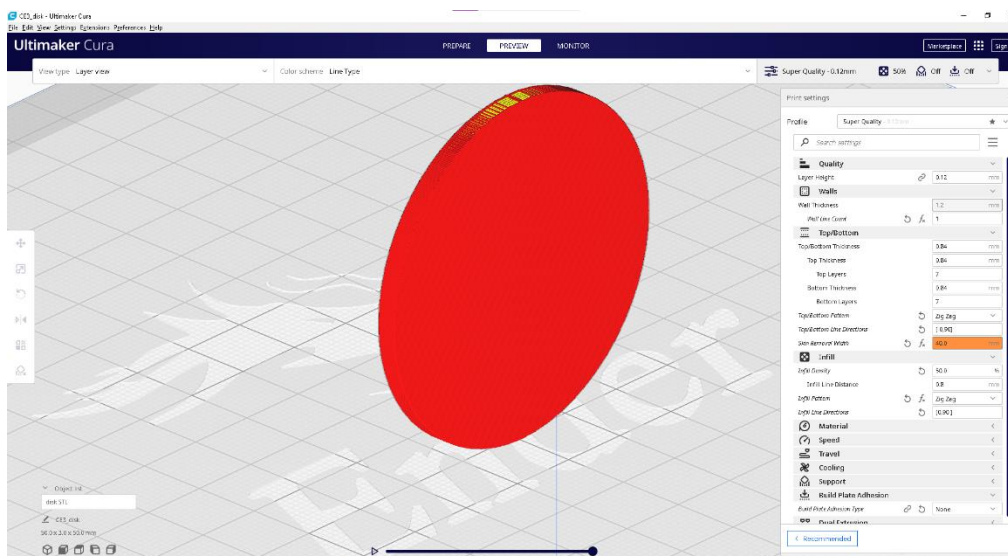


Figure A-1: Generating G-code in ULTIMAKER CURA software.

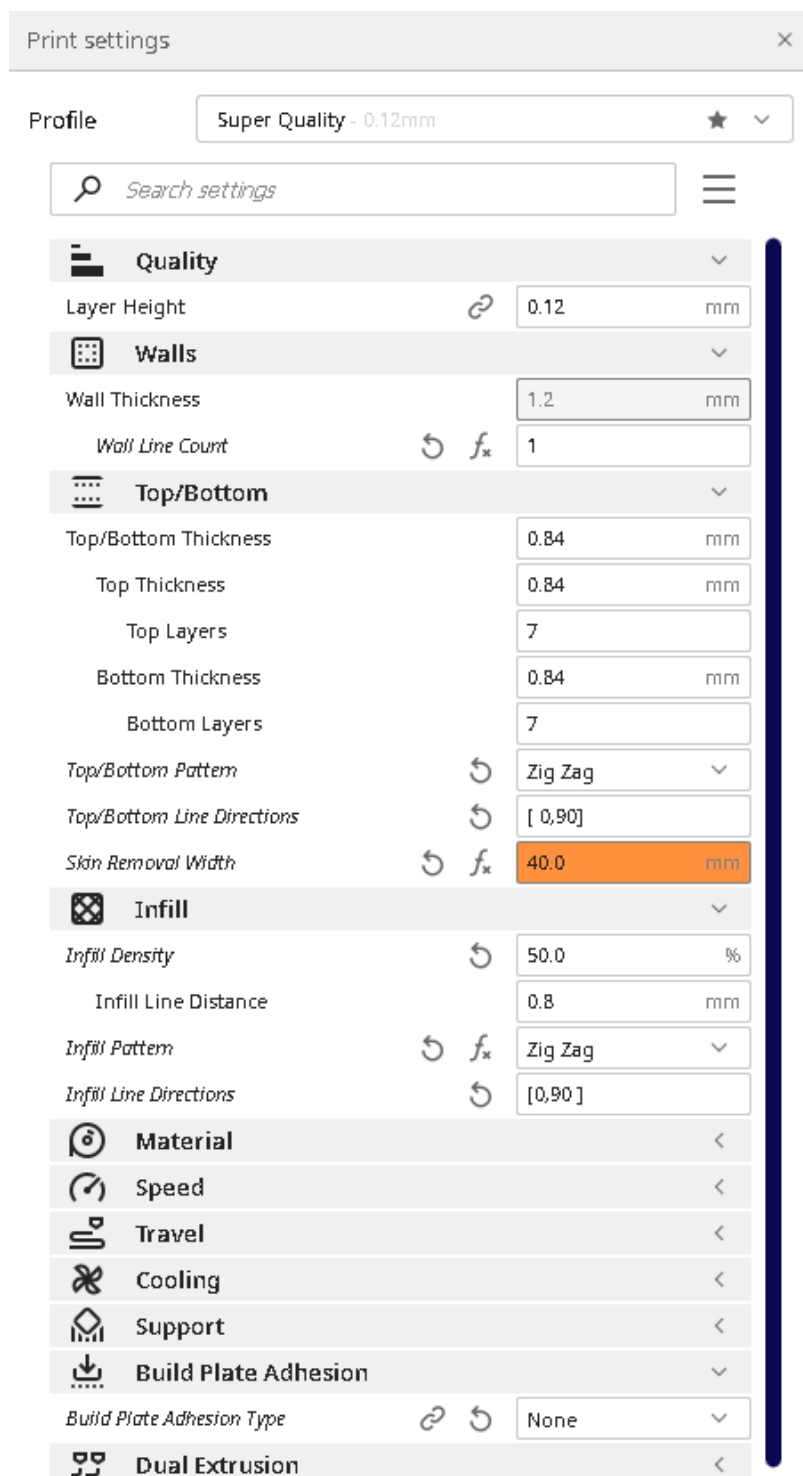


Figure A-2: Print Parameters set in CURA for sample 4.



## Appendix B: Coding

```

#include<iostream>
#include<fstream>
#include<sstream>
#include<string>
#include<vector>
#include<GL/freeglut.h>
#include <algorithm>
#include <chrono>
using namespace std::chrono;
using namespace std;

// Headers for generating pipe (filament)
#include "Vectors.h"
#include "Matrices.h"
#include "Plane.h"
#include "Line.h"
#include "Pipe.h"

// Function Prototype//
void grid();
void origin();
void Rendering();
void Solid_Render();
void display();
void reshape(int w, int h);
void speed(int);
void STLreader();
void Slicer();
void ContourCrea();
void Infill();
void initSharedMem(); //
void GetInfo();
void ParDesrp(); // Parameters desrcption
void DisInfo();
void drawPipe();
void drawPath();
void renderView();
void renderPrinting();
void renderPrinted();
void CurrPrinted();
void renderPreview();
void renderInfill();
void CurrInfill();
void CurrLayer();
void PrintedInfill();
void nozzle();
void loadingBar();
void printTime();

Pipe pipe;
std::vector<Vector3> path;
std::vector<Vector3> path2;
std::vector<Vector3> path3;
std::vector<Vector3> path4;
std::vector<Vector3> path5;
std::vector<Vector3> path6;

```

```

std::vector<Vector3> path7;
std::vector<Vector3> path8;
std::vector<Vector3> path9;
std::vector<Vector3> circle;
std::vector<Vector3> buildPath(float startpoint[3], float endpoint[3]);
std::vector<Vector3> buildCircle(float radius, int steps, bool X);

// Pi
const double pi = 3.14159265358979323846;

// keyboard
float cx = 0, cy = 0, cz = 0;
float ax = 0, ay = 0, az = 0;
float rotx = 0, roty = 0, rotz = 0;
float zoom = 0;
int nxt_lyr = 0;
int nxt_pnt[10000];
int pnt_Infill[10000];
bool preview;
bool renderMode;
float fr = 1;
int viewAngle;
int tempView;

// STL reader variables
string stl;
float m_vertex[100000][3];
float m_normal[100000][3];
float m_original_vertex[100000][3];
float m_original_normal[100000][3];
int z;
float novertex[30], nonormal[30];
int i, j, k, ll, l;
char coor[30];
char h[] = "endsolid";

// Get info
char customize;

// Slicer variables
float sliced_vertex[1000][1000][3]; // #1st [] = layer and #2nd [] = the point number of that layer and #3rd [] = the point size (x,y,z)
int slicedVetx_num[1000][2];
float layer_gap;
float tempCase1[1000][2]; // store the possible circle center
int pnt_C1 = 0;
bool check_C1 = false;
float temp_vertex[1000][3];
float cirCen[2] = { 0,0 };
const int N = 1000;
int seen[N];

// Contour creation variables
int startlyr = 0;
float startList[1000][1000][3]; // #1st [] = layer ; #2nd [] = start/end (odd is start/ even is end) ; #3rd [] = x,y,z;

```

```

float endList[1000][1000][3]; // #1st [] = layer ; #2nd [] = sta
rt/end (odd is start/ even is end) ; #3rd [] = x,y,z;
float order_list[1000][1000][50][3]; // #1st [] = layer ; #2nd [
] = start/end (odd is start/ even is end) ; #3rd [] = no. contou
r ; #4th [] = x,y,z;
float flag[1000][1000]; // flag (1 = visited / 0 = hasnt visited
) ## 1st [] = layer ; 2nd [] = nth start point;
int contour_[1000]; // number of contour for each layer; [] = la
yer
int pntNum_order[1000][1000]; // number of point in order list;
1st [] = layer; 2nd [] = contour

// Infill
char infillPattern;
float bottomThickness;
float topThickness;
float bottom_layer;
float top_layer;
float InfillGap_Max = 40; // Maximum infill gap
float infill[1000][2000][30][3]; // #1st [] = layer ; #2nd [] =
no. point ; #3rd [] = no. contour ; #4th [] = x,y,z;
int pntNum_infill[10000][10000]; // point
float contNum_infill[1000]; // contour
float InfillPerc; // Infill Density (Percentage)
int PercBotTop = 100; // 100% for top and bottom layers

// Rendering
int time_;
char RenMode;
float eyeX = 0.0f, eyeY = 500.0f, eyeZ = 600.0f;
int renderLyr = 0;
int renderContr[1000];
int infillContr[1000];
int printedLyr = 0;
int infillLyr = 0;
int printedInfillLyr = 0;
int printedCurrLyr;
bool infill_preview = false;

// Filament rendering
const int CIRCLE_SECTORS = 48;
double X_step = 0.0125; // 0.0125 mm per step
int currIndex = 0;
bool DonePrint;
bool iDonePrint;

// Nozzle
float nx, ny, nz; // nozzle movement x y z
bool Reach = false; // to check the status of initial nozzle mov
ement
int count_n = 0;
float nozzle_path[10000][3];
float pntNum_n;
int index_n;

//lighting
GLfloat yellow[] = { 1,1,0,1 };
GLfloat white[] = { 1,1,1,1 };
GLfloat black[] = { 0,0,0,1 };

```

```

GLfloat lowam[] = { 0.2f,0.2f,0.2f,1.0f };
GLfloat hiam[] = { 1,1,1,1 };
GLfloat red[] = { 1,0,0,0 };
GLfloat blue[] = { 0, 0, 1, 0};
GLfloat green[] = { 0, 1, 0, 0 };
GLfloat purple[] = { 1,0,1,0,0 };

void keyboard(unsigned char key, int x, int y)
{
    //Movements
    if (key == 'w') { if (cy < 500.0) cy += 1; }
    if (key == 's') { if (cy > -500.0) cy -= 1; }
    if (key == 'a') { if (cx > -500.0) cx -= 1; }
    if (key == 'd') { if (cx < 500.0) cx += 1; }
    if (key == 'q') { if (cz > -500.0) cz -= 1; }
    if (key == 'e') { if (cz < 500.0) cz += 1; }

    //Rotations
    if (key == 'i') { rotx += 1; }
    if (key == 'k') { rotx -= 1; }
    if (key == 'j') { rotz += 1; }
    if (key == 'l') { rotz -= 1; }

    // Preview the layered object
    if (key == 'f') { preview = true; }
    if (key == 'g') { preview = false; }

    // Zoom in/out
    if (key == 'z') { zoom = zoom + 10; }
    if (key == 'x') { zoom = zoom - 10; }

    //Speed control
    if (key == 'b') { fr = 500; }
    if (key == 'n') { fr = 1; }

    // View Angle
    if (key == '1') { viewAngle = 1; } // Isometric
    if (key == '2') { viewAngle = 2; } // Top
    if (key == '3') { viewAngle = 3; } // Right side
    if (key == '4') { viewAngle = 4; } // Left side
    if (key == '5') { viewAngle = 5; } // Front

    // Render Mode (solid/ printing)
    if (key == 'p') { renderMode = true; }
    if (key == 'o') { renderMode = false; }
    if (key == 't') { infill_preview = true; }
    if (key == 'y') { infill_preview = false; }

    glutPostRedisplay();
}

// Different eyes view
void renderView()
{
    if (viewAngle == 1) // Isometric view
    {
        cx = 0; cy = 0; cz = 0;
        rotx = 0; roty = 0; rotz = 0;
        zoom = 0;
    }
}

```

```

        eyeX = 500.0f; eyeY = 400.0f; eyeZ = 600.0f;
        tempView = 1;
        viewAngle = 0;
    }
    else if (viewAngle == 2) // Top View
    {
        cx = 0; cy = 0; cz = 0;
        rotx = 0; roty = 0; rotz = 0;
        zoom = 0;
        eyeX = 10.0f; eyeY = 0.0f; eyeZ = 700.0f;
        tempView = 2;
        viewAngle = 0;
    }
    else if (viewAngle == 3) // Right Side View
    {
        cx = 0; cy = 0; cz = 0;
        rotx = 0; roty = 0; rotz = 0;
        zoom = 0;
        eyeX = 700.0f; eyeY = 0.0f; eyeZ = 300.0f;
        tempView = 3;
        viewAngle = 0;
    }
    else if (viewAngle == 4) // Left Side View
    {
        cx = 0; cy = 0; cz = 0;
        rotx = 0; roty = 0; rotz = 0;
        zoom = 0;
        eyeX = -700.0f; eyeY = 0.0f; eyeZ = 300.0f;
        tempView = 4;
        viewAngle = 0;
    }
    else if (viewAngle == 5) // Front View
    {
        cx = 0; cy = 0; cz = 0;
        rotx = 0; roty = 0; rotz = 0;
        zoom = 0;
        eyeX = 0.0f; eyeY = 750.0f; eyeZ = 400.0f;
        tempView = 5;
        viewAngle = 0;
    }
}

void init()
{
    // Set background color to black and opaque
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    // Set background depth to farthest
    glClearDepth(1.0f);
    // Enable depth testing for z-culling
    glEnable(GL_DEPTH_TEST);
    // Set the type of depth-test
    glDepthFunc(GL_LEQUAL);
    // Enable smooth shading
    glShadeModel(GL_SMOOTH);
    // Nice perspective corrections
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
    // to enable the lighting in the viewing volume/ rendering
    // space
    glEnable(GL_LIGHTING);
}

```

```

// the type of light
glEnable(GL_LIGHT0);
// the parameter of that particular lighting
GLfloat lmodel_ambient[] = { 1, 0, 1, 0.0 }; // set to purple color
// The glLightfv function returns light source parameter values.
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
}

// Explanation of the parameters
void ParDesrp()
{
    cout << "\n=====
=====
\n";
    cout << "Print Setting" << endl;
    cout << "=====
=====
\n";
    cout << "Layer Height\n" << endl;
    cout << ">> The height of each layer in mm. Higher values
produce faster prints in lower\n resolution, lower values pro
duce slower prints in higher resolution." << endl;
    cout << " (Range: 0.12 mm ~ 0.28 mm)" << endl;
    cout << "-----
-----" << endl;
    cout << "Infill Density\n" << endl;
    cout << ">> Adjusts the density of infill of the print."
<< endl;
    cout << "-----
-----" << endl;
    cout << "Infill Pattern\n" << endl;
    cout << ">> The pattern of the infill material of the pri
nt. The linear and line infill\n swap direction on alternate l
ayers." << endl;
    cout << " [Line pattern(L) / Linear pattern(Z).]" << en
dl;
    cout << "-----
-----" << endl;
    cout << "Bottom Thickness\n" << endl;
    cout << ">> The thickness of bottom layers in the print,
this value is divided by the\n layer height defines the number
of bottom layers." << endl;
    cout << "-----
-----" << endl;
    cout << "Top Thickness\n" << endl;
    cout << ">> The thickness of top layers in the print, thi
s value is divided by the\n layer height defines the number of
top layers." << endl;
    cout << "-----
-----" << endl;
    cout << "Bottom Layers\n" << endl;
    cout << ">> The number of bottom layers, this value is ro
unded to a whole number." << endl;
    cout << "-----
-----" << endl;
    cout << "Top Layers\n" << endl;
    cout << ">> The number of top layers, this value is round
ed to a whole number.\n" << endl;
}

```



```

        if (infillPattern == 'l' || infillPattern == 'L') { cout
<< "Line Pattern" << endl; }
        else { cout << "Linear Pattern" << endl; }
        cout << "Bottom Thickness\t\t\t\t\t: " << bottomThickness
<< " mm" << endl;
        cout << "Bottom Layer(s)\t\t\t\t\t: " << bottom_layer <
< " layer(s)" << endl;
        cout << "Top Thickness\t\t\t\t\t: " << topThickness <<
" mm" << endl;
        cout << "Top Layer(s)\t\t\t\t\t: " << top_layer << " la
yer(s)" << endl;
        cout << "Wall Print Speed\t\t\t\t\t: " << "25.0 mm/s" <<
endl;
        cout << "Infill Print Speed\t\t\t\t\t: " << "50.0 mm/s" <
< endl;
        cout << "Skin Print Speed\t\t\t\t\t: " << "25.0 mm/s" <<
endl;
        cout << "Initial Layer Print Speed\t\t\t\t\t: " << "20.0 mm
/s" << endl;
        cout << "Rendering Mode\t\t\t\t\t: ";
        if (RenMode == 'l' || RenMode == 'L') { cout << "Line Mod
e" << endl; }
        else { cout << "Pipe Mode" << endl; }
        cout << "=====\n" << endl;

        cout << "Press C to customize the print setting ... \nPre
ss ANY KEY to confirm the default print setting ... \n>> ";

        while (!(cin >> customize))
        {
            // Explain error

            cerr << "\nPress C to customize the print setting ... ";

            // Clear input stream
            cin.clear();

            // Discard previous input
            cin.ignore(123, '\n');
        }

        if(customize == 'C' || customize == 'c')
        {

            ParDesrp(); // describe the parameters that will get from
user

            cout << "Customize Print Setting" << endl;

            cout << "====="
            << endl;
            cout << "Layer height (mm): ";

            while (!(cin >> layer_gap) || (layer_gap < 0.11 || layer_
gap > 0.29))
            {
                // Explain error

```



```

cerr << "\n## ERROR! Please enter a valid number. ";
cout << "\n\nLayer height (mm): ";
// Clear input stream
cin.clear();

// Discard previous input
cin.ignore(123, '\n');
}

cout << "Infill density (%): ";

while (!(cin >> InfillPerc) || (InfillPerc < 0 || InfillPerc > 100))
{
// Explain error

cerr << "\n## ERROR! Please enter a valid number. ";
cout << "\n\nInfill density (%): ";
// Clear input stream
cin.clear();

// Discard previous input
cin.ignore(123, '\n');
}

cout << "Infill pattern: ";

while (!(cin >> infillPattern) || (infillPattern != 'z' &
& infillPattern != 'Z' && infillPattern != 'L' && infillPattern
!= 'l'))
{
// Explain error

cerr << "\nLine pattern (L) / Linear pattern (Z). ";

// Clear input stream
cin.clear();

// Discard previous input
cin.ignore(123, '\n');
}

cout << "Bottom thickness (mm): ";

while (!(cin >> bottomThickness))
{
// Explain error

cerr << "\n## ERROR! Please enter a valid number. ";
cout << "\n\nBottom thickness (mm): ";
// Clear input stream
cin.clear();

// Discard previous input
cin.ignore(123, '\n');
}
}

```



```

    cout << "Layer Height\t\t\t\t\t: " << layer_gap << " mm
" << endl;
    cout << "Quality\t\t\t\t\t: ";
    if (layer_gap > 0.11) {
        if (layer_gap > 0.15) {
            if (layer_gap > 0.19) {
                if (layer_gap > 0.27) { cout << "Low Quality"; }
                else { cout << "Standard Quality"; }
            }
            else { cout << "Dynamic Quality"; }
        }
        else { cout << "Super Quality"; }
    }

    cout << "\nInfill Density\t\t\t\t\t: " << InfillPerc <<
" %" << endl;

    cout << "Infill Line Distance\t\t\t\t\t: " << InfillGap_M
ax / InfillPerc << " mm" << endl;
    cout << "Infill Pattern\t\t\t\t\t: ";
    if (infillPattern == 'l' || infillPattern == 'L') { cout
<< "Line Pattern" << endl; }
    else { cout << "Linear Pattern" << endl; }

    cout << "Bottom Thickness\t\t\t\t\t: " << bottomThickness
<< " mm" << endl;

    cout << "Bottom Layer(s)\t\t\t\t\t: " << bottom_layer <
< " layer(s)" << endl;

    cout << "Top Thickness\t\t\t\t\t: " << topThickness <<
" mm" << endl;

    cout << "Top Layer(s)\t\t\t\t\t: " << top_layer << " la
yer(s)" << endl;

    cout << "Wall Print Speed\t\t\t\t\t: " << "25.0 mm/s" <<
endl;

    cout << "Infill Print Speed\t\t\t\t\t: " << "50.0 mm/s" <
< endl;

    cout << "Skin Print Speed\t\t\t\t\t: " << "25.0 mm/s" <<
endl;

    cout << "Initial Layer Print Speed\t\t\t\t\t: " << "20.0 mm
/s" << endl;
    cout << "Rendering Mode\t\t\t\t\t: ";
    if (RenMode == 'l' || RenMode == 'L') { cout << "Line Mod
e" << endl; }
    else { cout << "Pipe Mode" << endl; }

```

```

    }

    std::cout << "\n=====
=====";
    std::cout << "\n\t\t\t Keyboard controls\n";
    std::cout << "=====
=====
\n";
    std::cout << "Camera controls\n";
    std::cout << "Key\tFunction\t\t Key\tFunction\n";
    std::cout << "w\tMove camera forward\t i\tRotate X axis c
ounter clockwise\n";
    std::cout << "s\tMove camera backward\t k\tRotate X axis
clockwise\n";
    std::cout << "a\tMove camera left\t j\tRotate Y axis coun
ter clockwise\n";
    std::cout << "d\tMove camera right\t l\tRotate Y axis clo
ckwise\n";
    std::cout << "q\tMove camera up\t\t e\tMove camera down\n
";
    std::cout << "z\tZoom in\t\t\t x\tZoom out\n";
    std::cout << "1\tIsometric View\t\t 2\tTop View\n";
    std::cout << "3\tRight View\t\t 4\tLeft View\n";
    std::cout << "5\tFront View\n";
    std::cout << "p\tExit solid body mode and starts printing
process\n";
    std::cout << "-----
\n";
    std::cout << "3D printing process control\n";
    std::cout << "b\tFast forward\t\t\t n\tNormal speed\t\t\t\n
";
    std::cout << "f\tShow entire 3D print body\t g\tBack to p
rinting simulation\n";
    std::cout << "-----
\n";
    cout << "Height of the object
    cout << "Total number of printing layer(s)
    printTime();
    std::cout << "-----
\n";
    cout << "\nPress S to proceed to the simulation.....";

    char proceed;

    while (!(cin >> proceed) || (proceed != 'S' && proceed !=
's'))
    {
        // Error

        cerr << "\nPress S to proceed to the simulation.....";

        // Clear input stream
        cin.clear();

        // Discard previous input
        cin.ignore(123, '\n');
    }
}

// Display a progress bar

```

: " &lt;&lt; layer\_

: " &lt;&lt; sliced



```

glLineWidth(1);
int count = pipe.getContourCount();
for (int i = 0; i < count; ++i)
{
    std::vector<Vector3> contour = pipe.getContour(i);
    std::vector<Vector3> normal = pipe.getNormal(i);
    glBegin(GL_LINES);
    for (int j = 0; j < (int)contour.size() - 1; ++j)
    {
        glNormal3fv(&normal[j].x);
        glVertex3fv(&contour[j].x);
        glNormal3fv(&normal[j + 1].x);
        glVertex3fv(&contour[j + 1].x);
    }
    glEnd();
}

// surface
for (int i = 0; i < count - 1; ++i)
{
    std::vector<Vector3> c1 = pipe.getContour(i);
    std::vector<Vector3> c2 = pipe.getContour(i + 1);
    std::vector<Vector3> n1 = pipe.getNormal(i);
    std::vector<Vector3> n2 = pipe.getNormal(i + 1);
    glBegin(GL_TRIANGLE_STRIP);
    for (int j = 0; j < (int)c2.size(); ++j)
    {
        glNormal3fv(&n2[j].x);
        glVertex3fv(&c2[j].x);
        glNormal3fv(&n1[j].x);
        glVertex3fv(&c1[j].x);
    }
    glEnd();
}
}

////////////////////////////////////
////////////////////////////////////
// draw lines along the path
////////////////////////////////////
////////////////////////////////////
void drawPath()
{
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    // lines
    glColor3f(1.0f, 0.5f, 0.0f);
    glLineWidth(2.0f);
    glBegin(GL_LINES);

    int count = pipe.getPathCount();
    for (int i = 0; i < count - 1; ++i)
    {
        glVertex3fv(&pipe.getPathPoint(i).x);
        glVertex3fv(&pipe.getPathPoint(i + 1).x);
    }
    glEnd();
}

```

```

// points
glColor3f(0.0f, 1.0f, 1.0f);
glPointSize(5.0f);
glBegin(GL_POINTS);
for (int i = 0; i < count; ++i)
{
    glVertex3fv(&pipe.getPathPoint(i).x);
}
glEnd();
glPointSize(1); // reset

glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);
}

// Build circle for pipe
std::vector<Vector3> buildCircle(float radius, int steps, bool X)
{
    std::vector<Vector3> points;
    if (steps < 2) return points;

    const double PI2 = acos(-1) * 2.0f;
    float x, y, a;
    for (int i = 0; i <= steps; ++i)
    {
        a = PI2 / steps * i;
        x = radius * cosf(a);
        y = radius * sinf(a);

        if (X)
        {
            points.push_back(Vector3(0, x, y)); // parallel to x-axis
        }
        else
        {
            points.push_back(Vector3(x, 0, y)); // parallel to y-axis
        }
    }
    return points;
}

// Generate path in straight line
std::vector<Vector3> buildPath(float startpoint[3], float endpoint[3])
{
    double dist_x, dist_y, euclidean_d;
    int steps;
    std::vector<Vector3> vertices;
    Vector3 vertex;

    dist_x = abs(startpoint[0] - endpoint[0]);
    dist_y = abs(startpoint[1] - endpoint[1]);
    euclidean_d = sqrt(pow(startpoint[0] - endpoint[0], 2) + pow(startpoint[1] - endpoint[1], 2));

    steps = euclidean_d / X_step; // number of steps
}

```

```

pntNum_n = steps; // store for nozzle movement

// For distance is smaller than 0.0125 mm
if (dist_x < X_step && dist_y < X_step)
{
    dist_x = 0.0001;
    dist_y = 0.0001;
    steps = 1;
}

// Change the sign based on the position of x
if (startpoint[0] > endpoint[0])
{
    // distance becomes -ve
    dist_x = -dist_x;
}

if (startpoint[1] > endpoint[1])
{
    // distance become -ve
    dist_y = -dist_y;
}

for (int i = 0; i < steps; ++i)
{
    vertex.x = startpoint[0] + dist_x * i / steps;
    vertex.y = startpoint[1] + dist_y * i / steps;
    vertex.z = startpoint[2];
    vertices.push_back(vertex);

    // Store for nozzle movement
    nozzle_path[i][0] = vertex.x;
    nozzle_path[i][1] = vertex.y;
    nozzle_path[i][2] = vertex.z;
}

return vertices;
}

void printTime()
{
    float sec, ttl_time, i_ttl_t, tbTime;
    float dis;
    sec = 0;
    ttl_time = 0;
    i_ttl_t = 0;
    tbTime = 0; // top and bottom
    float printSpeed_i = 50.0f; // 50 mm/s infill
    float printSpeed_o = 25.0f; // 25 mm/s wall
    int seconds, hours, minutes;

    for (int lyr = 0; lyr < slicedVetx_num[999][1]; lyr++)
    {
        for (int cont = 0; cont < contour_[lyr]; cont++)
        {
            for (int pnt = 0; pnt < slicedVetx_num[lyr][0]; pnt++)
            {

```



```

        dis = sqrt(pow(order_list[lyr][pnt][cont][0] -
            order_list[lyr][pnt + 1][cont][0], 2) + pow(order_list[lyr][pnt
][cont][1] - order_list[lyr][pnt + 1][cont][1], 2));
            sec = dis / printSpeed_o;
            ttl_time += sec;

        if (pnt == slicedVetx_num[lyr][0] - 1)
            {

            if (cont == contour_[lyr] - 1)
                {

                dis = sqrt(pow(order_list[lyr][pnt][cont][0] -
                    order_list[lyr + 1][pnt][cont][0], 2) + pow(order_list[lyr][pnt
][cont][1] - order_list[lyr + 1][pnt][cont][1], 2));

                sec = dis / printSpeed_o;
                                                                    ttl_time += sec;
                }
            }
        }
    }
    seconds = ttl_time;
    minutes = seconds / 60;
    hours = minutes / 60;
    cout << "=====
=====
=====
cout << "\t\t\t Time Estimation " << endl;
    cout << "=====
=====
=====
cout << "Outer wall\t\t\t\t\t: " << int(hours) << " hou
rs " << int(minutes % 60)
    << " minutes " << endl;

    for (int lyr = 0; lyr < slicedVetx_num[999][1]; lyr++)
    {
        for (int cont = 0; cont < contour_[lyr]; cont++)
        {

            for (int pnt = 0; pnt < pntNum_infill[lyr][cont]; pnt++)
            {

                if (lyr < bottom_layer || lyr > (slicedVetx_num[999][1] -
                    top_layer - 1))
                    {
                        if (lyr < 2)
                            {
                                printSpeed_i = 20;
                            }
                        else
                            {
                                printSpeed_i = 25;
                            }
                    }
                else
                    {

```

```

        printSpeed_i = 50;
    }

    dis = sqrt(pow(infill[lyr][pnt][cont][0] -
infill[lyr][pnt + 1][cont][0], 2) + pow(infill[lyr][pnt][cont][
1] - infill[lyr][pnt + 1][cont][1], 2));
    sec = dis / printSpeed_i;

    if (lyr < bottom_layer || lyr >(slicedVetx_num[999][1] -
top_layer - 1))
    {
        tbTime += sec;
    }
    else
    {
        i_ttl_t += sec;
    }
    }
}

seconds = i_ttl_t;
minutes = seconds / 60;
hours = minutes / 60;
cout << "Infill\t\t\t\t\t\t\t\t: " << int(hours) << " hours
" << int(minutes % 60)
    << " minutes " << endl;

seconds = tbTime;
minutes = seconds / 60;
hours = minutes / 60;
cout << "Skin\t\t\t\t\t\t\t\t: " << int(hours) << " hours "
<< int(minutes % 60)
    << " minutes " << endl;

seconds = ttl_time + i_ttl_t + tbTime;
minutes = seconds / 60;
hours = minutes / 60;
std::cout << "-----\n";
cout << "Total\t\t\t\t\t\t\t\t: " << int(hours) << " hours
" << int(minutes % 60)
    << " minutes " << endl;
}

int main(int argc, char** argv)
{
    STLreader();
    GetInfo();
    Slicer();
    ContourCrea();
    Infill();
    DisInfo();

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);

```

```

glutInitWindowSize(1200, 800);
glutCreateWindow("3D Printing Simulator");
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutTimerFunc(0, speed, 0);
glutIgnoreKeyRepeat(0);
glutKeyboardFunc(keyboard);
//
initSharedMem(); // initiate pipe rendering
init(); // initiate window setting

glutMainLoop();

return 0;
}

void display()
{
    float zoomX, zoomY, zoomZ;
    float moveX, moveY, moveZ;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // position of the light source (x,y,z, w)
    // If w is 1.0, we are defining a light at a point in space.
    // If w is 0.0, the light is at infinity.
    GLfloat lightPos0[] = { 0.0f, 0.0f, 30.0f, 1.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos0);

    // Change view angle
    renderView();

    // Camera view
    //gluLookAt ( eyeX , eyeY , eyeZ , centerX , centerY , centerZ , upX , upY , upZ )
    gluLookAt(
        eyeX, eyeY, eyeZ,
        0.0f, 0.0f, 0.0f,
        0.0f, 0.0f, 1.0f
    );

    // Default setting
    zoomY = zoom; zoomZ = zoom; zoomX = 0;
    moveX = cx; moveY = cy; moveZ = cz;

    // Reconfigure with respective viewing angle
    if (tempView == 1) // Isometric
    {
        zoomX = zoom; zoomY = zoom; zoomZ = zoom;
        moveX = cx; moveY = cy; moveZ = cz;
    }
    else if (tempView == 2) // Top
    {
        zoomX = 0; zoomY = 0; zoomZ = zoom;
        moveX = cy; moveY = -cx; moveZ = cz;
    }
    else if (tempView == 3) // Side 1
    {

```

```

        zoomX = zoom; zoomY = 0; zoomZ = 0.5 * zoom;
        moveX = cy; moveY = -cx; moveZ = cz;
    }
    else if (tempView == 4) // Side 2
    {
        zoomX = -zoom; zoomY = 0; zoomZ = 0.5 * zoom;
        moveX = cy; moveY = cx; moveZ = cz;
    }
    else if (tempView == 5) // Front
    {
        zoomX = 0; zoomY = zoom; zoomZ = 0.5 * zoom;
        moveX = cx; moveY = cy; moveZ = cz;
    }

    // Constraint the range of rotation
    if (rotx > 360 || rotx < -360) { rotx = 0; }
    if (rotz > 360 || rotz < -360) { rotz = 0; }

    glTranslatef(zoomX, zoomY, zoomZ);
    glRotatef(rotx, 1, 0, 0);
    glRotatef(roty, 0, 1, 0);
    glRotatef(rotz, 0, 0, 1);
    glTranslatef(moveX, moveY, moveZ);

    // Rendering
    grid();
    origin();
    if (renderLyr > 0) { loadingBar(); }

    if (RenMode == 'P' || RenMode == 'p')
    {
        if (renderMode)
        {
            GLfloat lmodel_ambient[] = { 1, 0, 1, 0.0 }; // set to purple color
            glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
            if (Reach)
            {
                if (preview)
                {
                    renderPreview();
                }
                else
                {
                    int printspeed = fr * 1;
                    if (DonePrint)
                    {
                        GLfloat red[] = { 1, 0, 0, 0.0 }; // set to red color
                        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, red);
                        CurrLayer();
                    }

                    GLfloat green[] = { 0, 1, 0, 0.0 }; // set to green color
                }
            }
        }
    }

```

```

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, green);
                    renderPrinted();

    GLfloat blue[] = { 0, 0, 1, 0.0 }; // set to blue color
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, blue);
                    CurrPrinted();

    GLfloat yellow[] = { 1, 1, 0, 0.0 }; // set to yellow col
or
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, yellow);
                    CurrInfill();

    GLfloat cyan[] = { 0, 255, 255, 0.0 }; // set to cyan col
or
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, cyan);
                    PrintedInfill();

    for (int loop = 0; loop <= printspeed; loop++)
        {

    GLfloat white[] = { 1, 1, 1, 0.0 }; // set to white color

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, white);
                    renderPrinting();

                    if (DonePrint)
                    {

or
    GLfloat pink[] = { 255, 0, 127, 0.0 }; // set to pink col

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, pink);

    renderInfill();

                    }

                    }

                    }
                    nozzle();
                }
            else
            {

    GLfloat lmodel_ambient[] = { 1, 0, 1, 0.0 }; // set to pu
rple color

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
                    Solid_Render();

                }
            }
        else { Rendering(); }

    glutSwapBuffers();

```

```

}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei)w, (GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_SMOOTH);
    gluPerspective(20.0, w/h, 1, 1000.0);
    glMatrixMode(GL_MODELVIEW);
}

// Draw grid
void grid()
{
    glDisable(GL_LIGHTING);

    // Set to grey color
    glColor4f(0.5f, 0.5f, 0.5f, 0.5f);

    // Variables
    GLfloat axis_point, startpoint, endpoint, gapBTWgrid;
    startpoint = -110.0; // starting point of the grid line
    endpoint = 110.0; // ending point of the grid line
    gapBTWgrid = 10.0; // gap between each grid line

    // Grid lines rendering //
    for (axis_point = startpoint; axis_point <= endpoint; axis_
s_point += gapBTWgrid)
    {
        // Variable for x_axis //
        GLfloat gridLineVertex_X[] =
        {
            axis_point, endpoint, 0, // axis point is an increment va
riable for line formation
            axis_point, startpoint, 0
        };

        // Variable for y_axis //
        GLfloat gridLineVertex_Y[] =
        {
            startpoint, axis_point, 0,
            endpoint, axis_point, 0
        };

        // X_axis //
        glLineWidth(1);
        glBegin(GL_LINES);
        glVertex3f(axis_point, endpoint, 0);
        glVertex3f(axis_point, startpoint, 0);
        glEnd();

        // Y_axis //
        glLineWidth(1);
        glBegin(GL_LINES);
        glVertex3f(startpoint, axis_point, 0);
        glVertex3f(endpoint, axis_point, 0);
    }
}

```

```

        glEnd();
    }

    glEnable(GL_LIGHTING);
}

// Draw origin of the 3D space
void origin()
{
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    // draw origin point at left corner //
    // Set to red color
    glPointSize(20);
    glBegin(GL_POINTS);
    glColor4f(1.0f, 0, 0, 0.5f);
    glVertex3f(110, 110, 0);
    glEnd();

    // draw Z-axis at origin //
    // Set to blue color
    glLineWidth(5);
    glBegin(GL_LINES);
    glColor4f(0.0f, 0, 1.0f, 0.5f);
    glVertex3f(110, 110, 0);
    glVertex3f(110, 110, 20);
    glEnd();

    // draw Y-axis at origin //
    // Set to green color
    glBegin(GL_LINES);
    glColor4f(0, 1.0f, 0, 0.5f);
    glVertex3f(110, 110, 0);
    glVertex3f(110, 90, 0);
    glEnd();

    // draw X-axis at origin //
    // Set to yellow color
    glBegin(GL_LINES);
    glColor4f(1.0f, 1.0f, 0, 0.5f);
    glVertex3f(110, 110, 0);
    glVertex3f(90, 110, 0);
    glEnd();

    // Draw the frames
    glLineWidth(0.5);
    glBegin(GL_LINES);
    glColor4f(1.0f, 1.0f, 1.0f, 0.5f); // set to white color
    //
    glVertex3f(110, 110, 0);
    glVertex3f(110, 110, 250);
    //
    glVertex3f(-110, 110, 0);
    glVertex3f(-110, 110, 250);
    //
    glVertex3f(110, -110, 0);
    glVertex3f(110, -110, 250);
    //
}

```

```

    glVertex3f(-110, -110, 0);
    glVertex3f(-110, -110, 250);
    //
    glVertex3f(110, 110, 250);
    glVertex3f(-110, 110, 250);
    //
    glVertex3f(-110, 110, 250);
    glVertex3f(-110, -110, 250);
    //
    glVertex3f(110, -110, 250);
    glVertex3f(-110, -110, 250);
    //
    glVertex3f(110, 110, 250);
    glVertex3f(110, -110, 250);
    glEnd();

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
}

void speed(int)
{
    if (RenMode == 'p' || RenMode == 'P') { time_ = 0; }
    else { time_ = 0; }

    glutPostRedisplay();
    glutTimerFunc(time_ / 3, speed, 0);
}

// STL READER - extract the vertex and normal points
void STLreader()
{
    // Read the STL file
    cout << "\t\t\t3D Printer Simulation" << endl;
    cout << "=====  

=====\\n";
    cout << "Place STL file into the same folder as this solu  

tion file.\\n";
    cout << "Enter name of STL file : ";
    getline(cin, stl);
    string filename = stl + ".stl";
    ifstream stlfile;
    stlfile.open(filename);

    // STL file error validating
    while (stlfile.fail())
    {
        cerr << "\\n## Error -  

Failed to open " << stl << endl;

        cout << "\\n=====  

=====\\n";
        cout << "\t\t\t3D Printer Simulation" << endl;

        cout << "=====  

=====\\n";

        cout << "Place STL file into the same folder as this solu  

tion file.\\n";

```



```

        cout << "Enter name of STL file : ";
        getline(cin, stl);
        string filename = stl + ".stl";
        stlfile.open(filename);
    }

    // j - coordinate size ; k - triangle count ; ll -
normal point count
    k = 0; ll = 0;
    for (i = 0; i < 500000; i++)
    {
        stlfile >> coor; // Read 30 chars from the file

        if (strcmp(coor, h) != 0) // strcmp = string compare the
coor with "endsolid". == 0 means equal, != 0 means not equal
        {

            if (strcmp(coor, "vertex") == 0) // reach "vertex" lines
            {
                for (j = 0; j < 3; j++) // 3 -
coordinate size
                {

                    stlfile >> m_original_vertex[k][j]; // read the vertice
s

                    m_vertex[k][j] = m_original_vertex[k][j]; // store them
                    }
                    k++;

                    novertex[z]++; // number of triangulated facet
                }

                if (strcmp(coor, "normal") == 0) // strcmp = string compa
re the coor with "normal"
                {
                    for (j = 0; j < 3; j++)
                    {

                        stlfile >> m_original_normal[ll][j];

                        m_normal[ll][j] = m_original_normal[ll][j];
                        }
                        ll++;

                        nonormal[z]++; // number of normal
                    }
                }
                else
                    break;
            }
        }

        /* Center the vertices to the origin
float maxX, maxY, minX, minY, minZ;
float offsetX, offsetY, offsetZ;

maxX = 0; minX = 10000;
maxY = 0; minY = 10000;

```

```

minZ = 10000;

for (int i = 0; i < novertex[z]; i++)
{
    if (maxX < m_vertex[i][0]) { maxX = m_vertex[i][0]; }
    if (minX > m_vertex[i][0]) { minX = m_vertex[i][0]; }
    if (maxY < m_vertex[i][1]) { maxY = m_vertex[i][1]; }
    if (minY > m_vertex[i][1]) { minY = m_vertex[i][1]; }
    if (minZ > m_vertex[i][2]) { minZ = m_vertex[i][2]; }
}

float averX = (maxX + minX) / 2;
float averY = (maxY + minY) / 2;

offsetX = 0 - averX;
offsetY = 0 - averY;
offsetZ = 0 - minZ;

for (int i = 0; i < novertex[z]; i++)
{
    m_vertex[i][0] = m_vertex[i][0] + offsetX;
    m_vertex[i][1] = m_vertex[i][1] + offsetY;
    m_vertex[i][2] = m_vertex[i][2] + offsetZ;
}
}

// Taking the vertex then compare with a z plane
void Slicer()
{
    // Get the highest Z point //
    float highest;
    highest = m_vertex[0][2];
    // Loop to store largest number to m_vertex[0]
    for (i = 0; i < novertex[z]; i++){if (highest < m_vertex[
i][2]){ highest = m_vertex[i][2]; }}
    int no_lyr;
    no_lyr = highest / layer_gap;

    if (no_lyr * layer_gap < highest) { no_lyr++; }

    int case_, layer_count, point_count;
    int line_point[3]; // useful point for line equation (arr
ange the point to ease the code)
    float vektor[3]; // the vector of 2 points for case 3 & c
ase 4
    float t; // t of line parametric equation

    // Initialize the count
    layer_count = 0;
    point_count = 0;

    // compare and save the points into another variable (lay
er by layer)

```

```

    for (float layer_height = 0 ; layer_height <= highest; la
yer_height += layer_gap)
    {

        // Determine case type for each triangulated facet
        for (i = 0; i < novertex[z]; i += 3)
        {

            // Case 1

            if (m_vertex[i][2] == layer_height && m_vertex[i + 1][2]
== layer_height
&& m_vertex[i + 2][2] == layer_height)
                case_ = 1;

            // Case 2
            else if

                ((m_vertex[i][2] == layer_height && m_vertex[i + 1][2] ==
layer_height) ||

                (m_vertex[i][2] == layer_height && m_vertex[i + 2][2] ==
layer_height) ||

                (m_vertex[i + 1][2] == layer_height && m_vertex[i + 2][2]
== layer_height))
                    case_ = 2;

            // Case 3

            else if (m_vertex[i][2] == layer_height || m_vertex[i + 2
][2] == layer_height || m_vertex[i + 1][2] == layer_height)
                case_ = 3;

            // Case 4

            else if ((m_vertex[i][2] > layer_height && (m_vertex[i +
1][2] < layer_height || m_vertex[i + 2][2] < layer_height)) ||

                (m_vertex[i][2] < layer_height && (m_vertex[i + 1][2] > l
ayer_height || m_vertex[i + 2][2] > layer_height)))
                    case_ = 4;

            // Case 5
            else
                case_ = 5;

            // Store the sliced vertex with their respective case typ
e//

            switch (case_) {
            case 1:

                break;

            case 2:

                // Store two intersection points

                for (int count = 0; count < 3; count++)

```

```

        {
            if (m_vertex[i + count][2] == layer_height)
                {
                    sliced_vertex[layer_count][point_count][0] = m_vertex[i +
count][0]; // x
                    sliced_vertex[layer_count][point_count][1] = m_vertex[i +
count][1]; // y
                    sliced_vertex[layer_count][point_count][2] = m_vertex[i +
count][2]; // z
                    point_count++;
                }
            }
            break;

            case 3:
                for (int count = 0; count < 3; count++)
                    {
                        if (m_vertex[i + count][2] == layer_height)
                            {
                                // Determine the other two vertices for finding the other
intersection point
                                if (count == 0) { line_point[0] = 1; line_point[1] = 2; }
                                else if (count == 1) { line_point[0] = 0; line_point[1] =
2; }
                                else if (count == 2) { line_point[0] = 0; line_point[1] =
1; }

                                if ((m_vertex[i + line_point[0]][2] > layer_height && m_v
ertex[i + line_point[1]][2] < layer_height) ||
                                    (m_vertex[i + line_point[1]][2] > layer_height && m_verte
x[i + line_point[0]][2] < layer_height))
                                    {
                                        // store the only intersection vertex
                                        sliced_vertex[layer_count][point_count][0] = m_vertex[i +
count][0]; // x
                                        sliced_vertex[layer_count][point_count][1] = m_vertex[i +
count][1]; // y
                                        sliced_vertex[layer_count][point_count][2] = m_vertex[i +
count][2]; // z

                                        point_count++; // found 1st intersection vertex

```

```

        // Line equation //

        // Compute vector of the line between the points
        vector[0] = m_vertex[i + line_point[0]][0] -
m_vertex[i + line_point[1]][0]; // x0 - x1

        vector[1] = m_vertex[i + line_point[0]][1] -
m_vertex[i + line_point[1]][1]; // y0 - y1

        vector[2] = m_vertex[i + line_point[0]][2] -
m_vertex[i + line_point[1]][2]; // z0 - z1

        t = (layer_height -
m_vertex[i + line_point[0]][2]) / vector[2]; // find t of param
etric eq

        // 2nd intersection point

        sliced_vertex[layer_count][point_count][0] = m_vertex[i +
line_point[0]][0] + vector[0] * t; // x

        sliced_vertex[layer_count][point_count][1] = m_vertex[i +
line_point[0]][1] + vector[1] * t; // y

        sliced_vertex[layer_count][point_count][2] = m_vertex[i +
line_point[0]][2] + vector[2] * t; // z

        point_count++; // found 2nd intersection vertex
    }
}
break;

case 4:
    // Find two intersection points

    if ((m_vertex[i][2] > layer_height && (m_vertex[i + 1][2]
< layer_height && m_vertex[i + 2][2] < layer_height)) ||

        (m_vertex[i][2] < layer_height && (m_vertex[i + 1][2] > l
ayer_height && m_vertex[i + 2][2] > layer_height)))
    {

        // Assign the center point
        line_point[0] = 0;
        line_point[1] = 1;
        line_point[2] = 2;
    }

    else if ((m_vertex[i + 1][2] > layer_height && (m_vertex[
i][2] < layer_height && m_vertex[i + 2][2] < layer_height)) ||

```

```

        (m_vertex[i + 1][2] < layer_height && (m_vertex[i][2] > layer_height && m_vertex[i + 2][2] > layer_height)))
        {
            // Assign the center point
            line_point[0] = 1;
            line_point[1] = 0;
            line_point[2] = 2;
        }

        else if ((m_vertex[i + 2][2] > layer_height && (m_vertex[i][2] < layer_height && m_vertex[i + 1][2] < layer_height)) ||
        (m_vertex[i + 2][2] < layer_height && (m_vertex[i][2] > layer_height && m_vertex[i + 1][2] > layer_height)))
        {
            // Assign the center point
            line_point[0] = 2;
            line_point[1] = 0;
            line_point[2] = 1;
        }

        // Line equation //
        // Compute vector of the 1st line

        vector[0] = m_vertex[i + line_point[0]][0] -
        m_vertex[i + line_point[1]][0]; // x0 - x1

        vector[1] = m_vertex[i + line_point[0]][1] -
        m_vertex[i + line_point[1]][1]; // y0 - y1

        vector[2] = m_vertex[i + line_point[0]][2] -
        m_vertex[i + line_point[1]][2]; // z0 - z1

        t = (layer_height -
        m_vertex[i + line_point[0]][2]) / vector[2]; // find t of param
        etric eq

        // 1st intersection point

        sliced_vertex[layer_count][point_count][0] = m_vertex[i +
        line_point[0]][0] + vector[0] * t; // x

        sliced_vertex[layer_count][point_count][1] = m_vertex[i +
        line_point[0]][1] + vector[1] * t; // y

        sliced_vertex[layer_count][point_count][2] = m_vertex[i +
        line_point[0]][2] + vector[2] * t; // z

        point_count++; // found 1st intersection vertex

        // Compute vector of the 2nd line

        vector[0] = m_vertex[i + line_point[0]][0] -
        m_vertex[i + line_point[2]][0]; // x0 - x2

```

```

    vektor[1] = m_vertex[i + line_point[0]][1] -
m_vertex[i + line_point[2]][1]; // y0 - y2

    vektor[2] = m_vertex[i + line_point[0]][2] -
m_vertex[i + line_point[2]][2]; // z0 - z2

    t = (layer_height -
m_vertex[i + line_point[0]][2]) / vektor[2]; // find t of param
etric eq

    // 2nd intersection point

    sliced_vertex[layer_count][point_count][0] = m_vertex[i +
line_point[0]][0] + vektor[0] * t; // x

    sliced_vertex[layer_count][point_count][1] = m_vertex[i +
line_point[0]][1] + vektor[1] * t; // y

    sliced_vertex[layer_count][point_count][2] = m_vertex[i +
line_point[0]][2] + vektor[2] * t; // z

    point_count++; // found 2nd intersection vertex
        break;

        case 5:
            break;

        }
    }

    // Process to find out the center of circle if exists
    int temp_count = 0;
    if (check_C1)
    {
        for (int i = 0; i < N; i++)
            seen[i] = 0;

        for (int i = 0; i < pnt_C1; i++)
        {
            if (seen[i] == 0)
            {
                int count = 0;

                for (int j = i; j < pnt_C1; j++)
                {

                    if (tempCase1[j][0] == tempCase1[i][0] && tempCase1[j][1]
== tempCase1[i][1])
                        {
                            count ++;

                            seen[j] = 1;
                        }
                }
            }
        }
    }

```

```

if (count > 5) // similar points intersection
    {

//cout << layer_count << endl;

cirCen[0] = tempCase1[i][0];
cirCen[1] = tempCase1[i][1];

// Store original vertices to temporary storage
for (int t = 0; t < point_count; t++)
    {

temp_vertex[t][0] = sliced_vertex[layer_count][t][0];
temp_vertex[t][1] = sliced_vertex[layer_count][t][1];
temp_vertex[t][2] = sliced_vertex[layer_count][t][2];
    }

// Remove the unwanted circle center
for (int l = 0; l < point_count; l++)
    {

if (temp_vertex[l][0] != cirCen[0] || temp_vertex[l][1] !=
= cirCen[1])
        {

sliced_vertex[layer_count][temp_count][0] = temp_vertex[l
][0];
sliced_vertex[layer_count][temp_count][1] = temp_vertex[l
][1];
sliced_vertex[layer_count][temp_count][2] = temp_vertex[l
][2];

temp_count++;
        }
    }

point_count = temp_count;
    }
}

slicedVetx_num[layer_count][0] = point_count; // store th
e number of points of each layer
layer_count++; // to next layer
point_count = 0; // reset the point count
check_C1 = false; // reset the check case 1;
}

```



```

    // Get highest sliced layer
    if (layer_count == no_lyr && layer_count != 0)
    {
        slicedVetx_num[layer_count][0] = slicedVetx_num[layer_count - 1][0];

        for (int copy_pnt = 0; copy_pnt < slicedVetx_num[layer_count][0]; copy_pnt++)
        {
            sliced_vertex[layer_count][copy_pnt][0] = sliced_vertex[layer_count - 1][copy_pnt][0]; // x

            sliced_vertex[layer_count][copy_pnt][1] = sliced_vertex[layer_count - 1][copy_pnt][1]; // y

            sliced_vertex[layer_count][copy_pnt][2] = highest; // z
        }
        slicedVetx_num[999][1] = layer_count; // save the total number of layer
    }

// Arrange the sliced data in order
void ContourCrea()
{
    // Variables
    int ttl_lyr; // total sliced layer
    int endPnt_count, startPnt_count; // number of start/end points
    ttl_lyr = slicedVetx_num[999][1];
    float decimal_xy = 100.0; // 2 decimal points

    // Transfer sliced vertices into matrixA for polygon creation
    for (int lyr_count = startlyr; lyr_count < ttl_lyr; lyr_count++)
    {
        // Reset the point count
        endPnt_count = 0;
        startPnt_count = 0;

        for (int pnt_count = 0; pnt_count < slicedVetx_num[lyr_count][0]; pnt_count++)
        {
            if (pnt_count % 2 != 0)
            // odd point number of the layer = end point
            {
                // end point of all combination of sliced points

                endList[lyr_count][endPnt_count][0] = floor(sliced_vertex[lyr_count][pnt_count][0] * decimal_xy) / decimal_xy;
                // x (round to 2 dp)
            }
        }
    }
}

```

```

        endList[lyr_count][endPnt_count][1] = floor(sliced_vertex
[lyr_count][pnt_count][1] * decimal_xy) / decimal_xy;
        // y (round to 2 dp)

        endList[lyr_count][endPnt_count][2] = layer_gap * lyr_cou
nt; // z

                endPnt_count++; // end point + 1
            }
            else if (pnt_count % 2 == 0)
                // even point number of the layer = start point
            {

                // start point of all combination of sliced points

                startList[lyr_count][startPnt_count][0] = floor(sliced_ve
rtex[lyr_count][pnt_count][0] * decimal_xy) / decimal_xy;
                // x (round to 2 dp)

                startList[lyr_count][startPnt_count][1] = floor(sliced_ve
rtex[lyr_count][pnt_count][1] * decimal_xy) / decimal_xy;
                // y (round to 2 dp)

                startList[lyr_count][startPnt_count][2] = layer_gap * lyr
_count; // z

                startPnt_count++; // start Point + 1
            }
        }

        // Polygon creation algorithm
        int checkFlag = 0;
        int pnt_; // point count
        float searcher[3] = {0,0,0};
        float switchPoint[3] = { 0,0,0 }; // temporary point for
interchange end and start point
        bool NoPoint = false; //
        bool searchStatus = false; // whether can find the next p
oint or not
        int pntRecorder = 0; // record the set of start/end point
that is found coincident

        for (int lyr_count = startlyr; lyr_count < ttl_lyr; lyr_c
ount++)
        {

            contour_[lyr_count] = 0; // Reset the contour

            pntRecorder = 0; // Reset pntRecorder for next layer

            do // Check flag if all went through then go next layer
            {

                checkFlag = 0; // reset the checkFlag

```

```

        pntNum_order[lyr_count][contour_[lyr_count]] = 0; // reset the number of point for next contour in order list

        do // Check if contour has ended, if yes then find next contour
        {

            // Assign searcher to end point (initial pntRecorder is 0 to start with 0th point set of each layer)

            searcher[0] = endList[lyr_count][pntRecorder][0];
            searcher[1] = endList[lyr_count][pntRecorder][1];
            searcher[2] = endList[lyr_count][pntRecorder][2];

            NoPoint = false;
            searchStatus = false;

            // check got end = start or not

            for (pnt_ = 0; pnt_ < slicedVetx_num[lyr_count][0]/2; pnt_++)
            {
                if (!searchStatus)
                {

                    // if end = start AND has yet in order list

                    if ((searcher[0] == startList[lyr_count][pnt_][0] && searcher[1] == startList[lyr_count][pnt_][1]) &&
                        flag[lyr_count][pnt_] != 1)
                    {

                        //order_list[500][200][500][3]; // #1st [] = layer ; #2nd [] = no contour ; #3rd [] = point number; #4th [] = x,y,z;

                        flag[lyr_count][pnt_] = 1; // the flag change to 1 as this point has been gone through

                        // store start point

                        order_list[lyr_count][pntNum_order[lyr_count][contour_[lyr_count]]][contour_[lyr_count]][0] = startList[lyr_count][pnt_][0]; // x

                        order_list[lyr_count][pntNum_order[lyr_count][contour_[lyr_count]]][contour_[lyr_count]][1] = startList[lyr_count][pnt_][1]; // y

                        order_list[lyr_count][pntNum_order[lyr_count][contour_[lyr_count]]][contour_[lyr_count]][2] = startList[lyr_count][pnt_][2]; // z

                        pntNum_order[lyr_count][contour_[lyr_count]]++;

```

```

        // store end point
        order_list[lyr_count][pntNum_order[lyr_count][contour_[lyr_count]]][contour_[lyr_count]][0] = endList[lyr_count][pnt_][0]
; // x
        order_list[lyr_count][pntNum_order[lyr_count][contour_[lyr_count]]][contour_[lyr_count]][1] = endList[lyr_count][pnt_][1]
; // y
        order_list[lyr_count][pntNum_order[lyr_count][contour_[lyr_count]]][contour_[lyr_count]][2] = endList[lyr_count][pnt_][2]
; // z

        pntNum_order[lyr_count][contour_[lyr_count]]++; // order
list + 1

        pntRecorder = pnt_; // set pointer to the new point
        searchStatus = true; // able to search the next point
    }
}
/**
// check got end = end or not, if there is no end = start
if (!searchStatus)
{
    for (pnt_ = 0; pnt_ < slicedVetx_num[lyr_count][0]/2; pnt
_++)
    {
        // if end1 = end2 AND not the same point (eg. end1 = end1
) AND has yet in order list
        if ((searcher[0] == endList[lyr_count][pnt_][0] && search
er[1] == endList[lyr_count][pnt_][1]) &&
            pnt_ != pntRecorder && flag[lyr_count][pnt_] != 1)
            {
                //Interchange start and end
                switchPoint[0] = endList[lyr_count][pnt_][0];
                switchPoint[1] = endList[lyr_count][pnt_][1];
                switchPoint[2] = endList[lyr_count][pnt_][2];

                endList[lyr_count][pnt_][0] = startList[lyr_count][pnt_][
0];
                endList[lyr_count][pnt_][1] = startList[lyr_count][pnt_][
1];

```

```

2];
    endList[lyr_count][pnt_][2] = startList[lyr_count][pnt_][
2];

    startList[lyr_count][pnt_][0] = switchPoint[0];
    startList[lyr_count][pnt_][1] = switchPoint[1];
    startList[lyr_count][pnt_][2] = switchPoint[2];

    flag[lyr_count][pnt_] = 1; // the flag change to 1 as thi
s point has been gone through

    order_list[lyr_count][pntNum_order[lyr_count][contour_[ly
r_count]]][contour_[lyr_count]][0] = startList[lyr_count][pnt_][
0]; // x

    order_list[lyr_count][pntNum_order[lyr_count][contour_[ly
r_count]]][contour_[lyr_count]][1] = startList[lyr_count][pnt_][
1]; // y

    order_list[lyr_count][pntNum_order[lyr_count][contour_[ly
r_count]]][contour_[lyr_count]][2] = startList[lyr_count][pnt_][
2]; // z

    pntNum_order[lyr_count][contour_[lyr_count]]++; // order
list + 1

    order_list[lyr_count][pntNum_order[lyr_count][contour_[ly
r_count]]][contour_[lyr_count]][0] = endList[lyr_count][pnt_][0]
; // x

    order_list[lyr_count][pntNum_order[lyr_count][contour_[ly
r_count]]][contour_[lyr_count]][1] = endList[lyr_count][pnt_][1]
; // y

    order_list[lyr_count][pntNum_order[lyr_count][contour_[ly
r_count]]][contour_[lyr_count]][2] = endList[lyr_count][pnt_][2]
; // z

    pntNum_order[lyr_count][contour_[lyr_count]]++; // order
list + 1

    pntRecorder = pnt_; // set pointer to the new point

    searchStatus = true; // able to search the next point
    }
    }
}

// both conditions did not meet then use new search point
to get a new contour

```

```

        if (!searchStatus)
        {
            NoPoint = true;
            searchStatus = false;

            contour_[lyr_count]++; // next contour
        }

    } while (!NoPoint); // assign new point set to find next
point (new contour is formed)

    // check the flag of all startpoint of each layer/contour
    bool firstSet;
    firstSet = true;

    for (int count = 0; count < slicedVetx_num[lyr_count][0]
/ 2; count++)
    {
        if (flag[lyr_count][count] == 0)
        {
            if (firstSet)
            {

                pntRecorder = count;

                if (count > contour_[lyr_count]) { firstSet = false; }

            }

            checkFlag++; // checkFlag + 1 if any flag of point is 0

        }

        flag[lyr_count][pntRecorder] = 1; //

    } while (checkFlag != 0 && contour_[lyr_count] < 50); //
loop if any flag is 0;
}

// Returns true if x is in range [low..high], else false
bool inRange(unsigned low, unsigned high, unsigned x)
{
    return (low <= x && x <= high);
}

void Infill()
{
    float x_max, x_min, y_max, y_min;
    float infillDensity; //
    int ttl_lyr = slicedVetx_num[999][1];
    float t; // t for parametric eq
    float vector[3] = { 0,0,0 };
    float tol = 0.8; // boundary
}

```

```

for (int layer_c = 0; layer_c < ttl_lyr; layer_c++)
{
    // Vertical Infill
    if (layer_c % 2 == 0) // even number layer
    {

        // Get the highest and lowest coor on x axis
        x_max = sliced_vertex[layer_c][0][0];

        for (i = 0; i < slicedVetx_num[layer_c][0]; i++)
        {

            if (x_max < sliced_vertex[layer_c][i][0])
            {

                x_max = sliced_vertex[layer_c][i][0];
            }

            x_min = sliced_vertex[layer_c][0][0];

            for (i = 0; i < slicedVetx_num[layer_c][0]; i++)
            {

                if (x_min > sliced_vertex[layer_c][i][0])
                {

                    x_min = sliced_vertex[layer_c][i][0];
                }

                // Infill Density -
                // the gap(constant) / the percentage obtained from user
                if (InfillPerc > 0)
                {

                    infillDensity = InfillGap_Max / InfillPerc;
                }
                // If zero percentage then no infill
                else
                {

                    infillDensity = x_max - x_min;
                }

                // Top and bottom layer has full infill

                if (layer_c < bottom_layer || layer_c > (ttl_lyr -
top_layer - 1))
                {

                    //infillDensity = Line width;
                    infillDensity = 0.4;
                }

                float tempX1[3], tempX2[3];

                // Get the infill vertices

```

```

    for (int contour_c = 0; contour_c < contour_[layer_c]; contour_c++)
    {
        pntNum_infill[layer_c][contour_c] = 0;

        for (float x = x_min + infillDensity / 2; x < x_max; x +=
infillDensity) // x plane
            {
                for (int pnt_c = 0; pnt_c < pntNum_order[layer_c][contour_c] - 1; pnt_c++)
                    {
                        vector[0] = order_list[layer_c][pnt_c][contour_c][0] -
order_list[layer_c][pnt_c + 1][contour_c][0];

                        vector[1] = order_list[layer_c][pnt_c][contour_c][1] -
order_list[layer_c][pnt_c + 1][contour_c][1];

                        vector[2] = order_list[layer_c][pnt_c][contour_c][2] -
order_list[layer_c][pnt_c + 1][contour_c][2];

                        t = (x -
order_list[layer_c][pnt_c][contour_c][0]) / vector[0];

                        if (!isinf(t) && (abs(t) > 0 && abs(t) < 1)) // if not same
vector && (point is between line segment)
                            {
                                tempX1[0] = order_list[layer_c][pnt_c][contour_c][0] + t
* vector[0];

                                tempX1[1] = order_list[layer_c][pnt_c][contour_c][1] + t
* vector[1];

                                tempX1[2] = order_list[layer_c][pnt_c][contour_c][2] + t
* vector[2];

                                if (tempX1[0] != tempX2[0] || tempX1[1] != tempX2[1] || t
empX1[2] != tempX2[2]) // check for not similar point
                                    {

                                        if (!inRange(tempX1[1] -
tol, tempX1[1] + tol, tempX2[1])) // check for not near to the c
urrent point
                                            {

                                                infill[layer_c][pntNum_infill[layer_c][contour_c]][contou
r_c][0] = order_list[layer_c][pnt_c][contour_c][0] + t * vector[
0];

                                                infill[layer_c][pntNum_infill[layer_c][contour_c]][contou
r_c][1] = order_list[layer_c][pnt_c][contour_c][1] + t * vector[
1];

```



```

        infill[layer_c][pntNum_infill[layer_c][contour_c]][contou
r_c][2] = order_list[layer_c][pnt_c][contour_c][2] + t * vector[
2];

        tempX2[0] = infill[layer_c][pntNum_infill[layer_c][contou
r_c]][contour_c][0];

        tempX2[1] = infill[layer_c][pntNum_infill[layer_c][contou
r_c]][contour_c][1];

        tempX2[2] = infill[layer_c][pntNum_infill[layer_c][contou
r_c]][contour_c][2];

        pntNum_infill[layer_c][contour_c]++;
    }
}
}
}
}
}
// Horizontal Infill
else if(layer_c % 2 != 0)
{
    // Get the highest and lowest coor on y axis
    y_max = sliced_vertex[layer_c][0][1];

    for (i = 0; i < slicedVetx_num[layer_c][0]; i++)
    {
        if (y_max < sliced_vertex[layer_c][i][1])
        {
            y_max = sliced_vertex[layer_c][i][1];
        }
        y_min = sliced_vertex[layer_c][0][1];

        for (i = 0; i < slicedVetx_num[layer_c][0]; i++)
        {
            if (y_min > sliced_vertex[layer_c][i][1])
            {
                y_min = sliced_vertex[layer_c][i][1];
            }

            // Infill Density -
            the gap(constant) / the percentage obtained from user
            if (InfillPerc > 0)
            {

```

```

    infillDensity = InfillGap_Max / InfillPerc;
    }
    // If zero percentage then no infill
    else
    {
        infillDensity = y_max - y_min;
    }

    // Top and bottom layer has full infill

    if (layer_c < bottom_layer || layer_c > (ttl_lyr -
top_layer - 1))
    {
        //infillDensity = Line width
        infillDensity = 0.4;
    }

    // Get the infill vertices
    float tempY1[3], tempY2[3];

    for (int contour_c = 0; contour_c < contour_[layer_c]; co
ntour_c++)
    {

        pntNum_infill[layer_c][contour_c] = 0;

        for (float y = y_min + infillDensity / 2; y < y_max; y +=
infillDensity) // y plane
            {

                for (int pnt_c = 0; pnt_c < pntNum_order[layer_c][contour
_c] - 1; pnt_c++)
                    {
                        {
                            /**/

                            vector[0] = order_list[layer_c][pnt_c][contour_c][0] -
order_list[layer_c][pnt_c + 1][contour_c][0];

                            vector[1] = order_list[layer_c][pnt_c][contour_c][1] -
order_list[layer_c][pnt_c + 1][contour_c][1];

                            vector[2] = order_list[layer_c][pnt_c][contour_c][2] -
order_list[layer_c][pnt_c + 1][contour_c][2];

                            t = (y -
order_list[layer_c][pnt_c][contour_c][1]) / vector[1];

                            if (!isinf(t) && (abs(t) > 0 && abs(t) < 1)) // if not sa
me value && (point is between line segment)
                                {

                                    tempY1[0] = order_list[layer_c][pnt_c][contour_c][0] + t
* vector[0];

```

```

        tempY1[1] = order_list[layer_c][pnt_c][contour_c][1] + t
* vector[1];

        tempY1[2] = order_list[layer_c][pnt_c][contour_c][2] + t
* vector[2];

        if (tempY1[0] != tempY2[0] || tempY1[1] != tempY2[1] || t
empY1[2] != tempY2[2])
            {
                if (!inRange(tempY1[0] - tol, tempY1[0] + tol, tempY2[0]))
                    {

                        infill[layer_c][pntNum_infill[layer_c][contour_c]][contou
r_c][0] = order_list[layer_c][pnt_c][contour_c][0] + t * vector[
0];

                        infill[layer_c][pntNum_infill[layer_c][contour_c]][contou
r_c][1] = order_list[layer_c][pnt_c][contour_c][1] + t * vector[
1];

                        infill[layer_c][pntNum_infill[layer_c][contour_c]][contou
r_c][2] = order_list[layer_c][pnt_c][contour_c][2] + t * vector[
2];

                        tempY2[0] = infill[layer_c][pntNum_infill[layer_c][contou
r_c]][contour_c][0];

                        tempY2[1] = infill[layer_c][pntNum_infill[layer_c][contou
r_c]][contour_c][1];

                        tempY2[2] = infill[layer_c][pntNum_infill[layer_c][contou
r_c]][contour_c][2];

                        pntNum_infill[layer_c][contour_c]++;
                    }
            }
        }
    }
}

// ZigZag pattern
if (infillPattern == 'z' || infillPattern == 'Z')
{
    float tempLine[3];
    for (int ly = 0; ly < ttl_lyr; ly++)
    {

        for (int cont = 0; cont < contour_[ly]; cont++)
        {

```

```

    for (int set = 0; set < pntNum_infill[ly][cont] / 2; set++)
    {
        if (set % 2 != 0)
        {
            tempLine[0] = infill[ly][set * 2][cont][0];
            tempLine[1] = infill[ly][set * 2][cont][1];
            tempLine[2] = infill[ly][set * 2][cont][2];

            infill[ly][set * 2][cont][0] = infill[ly][set * 2 + 1][co
nt][0];
            infill[ly][set * 2][cont][1] = infill[ly][set * 2 + 1][co
nt][1];
            infill[ly][set * 2][cont][2] = infill[ly][set * 2 + 1][co
nt][2];

            infill[ly][set * 2 + 1][cont][0] = tempLine[0];
            infill[ly][set * 2 + 1][cont][1] = tempLine[1];
            infill[ly][set * 2 + 1][cont][2] = tempLine[2];
        }
    }
}

// Render object in solid mode
void Solid_Render()
{
    i = 0;
    j = 0;
    k = 1;
    l = 2;

    for (i = 0; i < nonormal[z]; i++)
    {
        glBegin(GL_TRIANGLES);

        glColor3ub(1.0f, 0.0f, 0.0f);

        {
            glNormal3f(m_normal[i][0], m_normal[i][1], m_normal[i][2]
);
            glVertex3f(m_vertex[j][0], m_vertex[j][1], m_vertex[j][2]
);
            glVertex3f(m_vertex[k][0], m_vertex[k][1], m_vertex[k][2]
);

```

```

);
    }

    j = j + 3;
    k = k + 3;
    l = l + 3;

    glEnd();
}
}

/**
////////////////////////////////////
void Rendering()
{
    int printspeed = fr * 10;

    for (int loop = 0; loop <= printspeed; loop++)
    {
        bool finishPrint_R = false;
        bool finishPrint_P = false;
        bool finishPrint_P1 = false;
        bool finishPrint_I = false;
        bool finishPrint_R1 = false;

        //////////////////////////////////////
        //////////////////////////////////////
        // Display current printing layer
        glMaterialfv(GL_FRONT, GL_AMBIENT, green);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, blue);
        glMaterialfv(GL_FRONT, GL_SPECULAR, green);
        glMaterialf(GL_FRONT, GL_SHININESS, 128);
        glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

        if (renderLyr < slicedVetx_num[999][1] && renderLyr >= 0
&& !preview)
            {

                if (renderContr[renderLyr] < contour_[renderLyr])
                    {
                        nxt_pnt[renderLyr]++;

                        if (nxt_pnt[renderLyr] < pntNum_order[renderLyr][renderCo
ntr[renderLyr]])
                            {

                                for (int k = 0; k < nxt_pnt[renderLyr]; k++)
                                    {
                                        glPointSize(7.0f);
                                        glBegin(GL_LINES);

                                        glColor3ub(1.0f, 1.0f, 1.0f);
                                    }
                                }
                            }
                    }
            }
}

```



```

    glColor3ub(1.0f, 1.0f, 1.0f);
    {
        glVertex3f
        (
            order_list[printedCurrLyr][k][j][0],
            order_list[printedCurrLyr][k][j][1],
            order_list[printedCurrLyr][k][j][2]
        );
        glVertex3f
        (
            order_list[printedCurrLyr][k + 1][j][0],
            order_list[printedCurrLyr][k + 1][j][1],
            order_list[printedCurrLyr][k + 1][j][2]
        );
    }
    glEnd();
}
}

////////////////////////////////////
////////////////////////////////////
    glMaterialfv(GL_FRONT, GL_AMBIENT, blue);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, blue);
    glMaterialfv(GL_FRONT, GL_SPECULAR, green);
    glMaterialf(GL_FRONT, GL_SHININESS, 128);
    glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

    infillLyr = renderLyr;

    if (infillLyr < slicedVetx_num[999][1] && infillLyr >= 0
    && !preview && finishPrint_R1)
    {
        if (infillContr[infillLyr] < contour_[infillLyr]) //contour_[infillLyr]
        {
            pnt_Infill[infillLyr]++;

            if (pnt_Infill[infillLyr] < pntNum_infill[infillLyr][infillContr[infillLyr]])
            {
                for (int k = 0; k < pnt_Infill[infillLyr]; k++)
                {
                    glPointSize(7.0f);
                    glBegin(GL_LINES);

```

```

glColor3ub(1.0f, 1.0f, 1.0f);
{
    glVertex3f
    (
        infill[infillLyr][k][infillContr[infillLyr]][0],
        infill[infillLyr][k][infillContr[infillLyr]][1],
        infill[infillLyr][k][infillContr[infillLyr]][2]
    );
    glVertex3f
    (
        infill[infillLyr][k + 1][infillContr[infillLyr]][0],
        infill[infillLyr][k + 1][infillContr[infillLyr]][1],
        infill[infillLyr][k + 1][infillContr[infillLyr]][2]
    );
}
    glEnd();
}
}
else { infillContr[infillLyr]++; }
}

else { finishPrint_I = true; renderLyr++; }
}

////////////////////////////////////
////////////////////////////////////
// Render the printed layer
glMaterialfv(GL_FRONT, GL_AMBIENT, red);
glMaterialfv(GL_FRONT, GL_DIFFUSE, blue);
glMaterialfv(GL_FRONT, GL_SPECULAR, green);
glMaterialf(GL_FRONT, GL_SHININESS, 128);
glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

/**
printedLyr = renderLyr;

if (printedLyr < slicedVetx_num[999][1] && printedLyr >=
0 && !preview)
{
    // Display layer by layer
    for (i = 0; i < printedLyr; i++)
    {
        for (j = 0; j < contour_[i]; j++)
        {

            for (int k = 0; k < pntNum_order[i][j] - 1; k++)
            {
                glVertex3f
                (
                    contour_[i][j][k],
                    contour_[i][j][k],
                    contour_[i][j][k]
                );
                glVertex3f
                (
                    contour_[i][j][k + 1],
                    contour_[i][j][k + 1],
                    contour_[i][j][k + 1]
                );
            }
        }
    }
}

glPointSize(5.0f);
glBegin(GL_LINES);

```



```

        glColor3ub(1.0f, 1.0f, 1.0f);
        {
            glVertex3f
            (
                order_list[i][k][j][0],
                order_list[i][k][j][1],
                order_list[i][k][j][2]
            );
            glVertex3f
            (
                order_list[i][k + 1][j][0],
                order_list[i][k + 1][j][1],
                order_list[i][k + 1][j][2]
            );
        }
        glEnd();
    }
    }
    finishPrint_P1 = true;
}
else { finishPrint_P = true; }

////////////////////////////////////
////////////////////////////////////
// Printed infill
glMaterialfv(GL_FRONT, GL_AMBIENT, white);
glMaterialfv(GL_FRONT, GL_DIFFUSE, blue);
glMaterialfv(GL_FRONT, GL_SPECULAR, white);
glMaterialf(GL_FRONT, GL_SHININESS, 128);
glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

printedInfillLyr = infillLyr - 1;
if (printedInfillLyr >= 0 && !preview)
{
    for (j = 0; j < contour_[printedInfillLyr]; j++) //contour_[printedInfillLyr]
    {
        for (int k = 0; k < pntNum_infill[printedInfillLyr][j] -
1; k++)
        {
            glPointSize(5.0f);
            glBegin(GL_LINES);

            glColor3ub(1.0f, 1.0f, 1.0f);
            {
                glVertex3f

```

```

(
    infill[printedInfillLyr][k][j][0],
    infill[printedInfillLyr][k][j][1],
    infill[printedInfillLyr][k][j][2]
);

    glVertex3f
    (
        infill[printedInfillLyr][k + 1][j][0],
        infill[printedInfillLyr][k + 1][j][1],
        infill[printedInfillLyr][k + 1][j][2]
    );
    }
    glEnd();
}
}

// ## When finish print, render the printed object
glMaterialfv(GL_FRONT, GL_AMBIENT, yellow);
glMaterialfv(GL_FRONT, GL_DIFFUSE, blue);
glMaterialfv(GL_FRONT, GL_SPECULAR, red);
glMaterialf(GL_FRONT, GL_SHININESS, 128);
glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

    if ((finishPrint_R && finishPrint_P) || preview)
    {
        if (!infill_preview)
        {
            for (int f_lyr = 0; f_lyr < slicedVetx_num[999][1]; f_lyr
++)
                {
                    for (int f_contour = 0; f_contour < contour_[f_lyr]; f_co
ntour++)
                        {
                            for (int f_pnt = 0; f_pnt < pntNum_order[f_lyr][f_contour
] - 1; f_pnt++)
                                {

                                    glBegin(GL_LINES);

                                    glColor3ub(1.0f, 1.0f, 1.0f);

                                    {

                                        glVertex3f
(

```

```

order_list[f_lyr][f_pnt][f_contour][0],
order_list[f_lyr][f_pnt][f_contour][1],
order_list[f_lyr][f_pnt][f_contour][2]
);

glVertex3f
(

order_list[f_lyr][f_pnt + 1][f_contour][0],
order_list[f_lyr][f_pnt + 1][f_contour][1],
order_list[f_lyr][f_pnt + 1][f_contour][2]
);
}
glEnd();
}
}
}

// Infill

glMaterialfv(GL_FRONT, GL_AMBIENT, blue);
glMaterialfv(GL_FRONT, GL_DIFFUSE, white);
glMaterialfv(GL_FRONT, GL_SPECULAR, white);
glMaterialf(GL_FRONT, GL_SHININESS, 128);
glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

int lyr = slicedVetx_num[999][1] - 1;
    if (lyr > 0)
    {

for (int j = 0; j < contour_[lyr]; j++)
    {

for (int k = 0; k < pntNum_infill[lyr][j] - 1; k++)
    {

glBegin(GL_LINES);

glColor3ub(1.0f, 1.0f, 1.0f);
{

glVertex3f
(

infill[lyr][k][j][0],

```

```

infill[lyr][k][j][1],
infill[lyr][k][j][2]
);

glVertex3f
(

infill[lyr][k + 1][j][0],
infill[lyr][k + 1][j][1],
infill[lyr][k + 1][j][2]
);
}
glEnd();
}
}
}
}
else
{

glMaterialfv(GL_FRONT, GL_AMBIENT, blue);
glMaterialfv(GL_FRONT, GL_DIFFUSE, white);
glMaterialfv(GL_FRONT, GL_SPECULAR, white);
glMaterialf(GL_FRONT, GL_SHININESS, 128);
glLightfv(GL_LIGHT0, GL_AMBIENT, hiam);

for (int lyr = 0; lyr < slicedVetx_num[999][1]; lyr++)
{
for (int j = 0; j < contour_[lyr]; j++)
{
for (int k = 0; k < pntNum_infill[lyr][j] - 1; k++)
{

glBegin(GL_LINES);

glColor3ub(1.0f, 1.0f, 1.0f);
{

glVertex3f
(

infill[lyr][k][j][0],
infill[lyr][k][j][1],
infill[lyr][k][j][2]

```



```

    path = buildPath(startpoint, endpoint);

    // To turn the cylinder direction
    float theta, OA;
    OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)

    theta = float(abs(atan(OA)) * 180 / pi);

    if (theta < 45) // theta < 25 change the circle direction
face to x axis
        {
            y = true;
        }

    // Configure the circle for path
    circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

    std::vector<Vector3> p(1, path[0]);
    pipe.set(p, circle);

    ++currIndex;
    if (currIndex < path.size())
    {

    pipe.addPathPoint(path[currIndex]);
        }
        else
        {

    currIndex = 0; // Reset the current index
    nxt_pnt[renderLyr] ++; // to next point
        }
        }
        else
        {

    // If no more points try next contour
        renderContr[renderLyr]++;
        }
    }
    else
    {

    DonePrint = true; // Done printing go for infill
        }
    }
    drawPipe();
}

```

```

// Render printed layers
void renderPrinted()
{
    for (int lyr = 0; lyr < renderLyr; lyr++)
    {
        for (int cont = 0; cont < contour_[lyr]; cont++)
        {
            for (int pnt = 0; pnt < pntNum_order[lyr][cont] -
1; pnt++)
                {
                    bool y = false;

                    float startpoint[3] = {

order_list[lyr][pnt][cont][0],

order_list[lyr][pnt][cont][1],

order_list[lyr][pnt][cont][2]
                    };
                    float endpoint[3] = {

order_list[lyr][pnt + 1][cont][0],

order_list[lyr][pnt + 1][cont][1],

order_list[lyr][pnt + 1][cont][2]
                    };

                    path2 = buildPath(startpoint, endpoint);

// To turn the cylinder direction
float theta, OA;
OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)

                    theta = float(abs(atan(OA)) * 180 / pi);

                    if (theta < 45)
                    {
                        y = true;
                    }

// Configure the circle for path

                    circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

                    std::vector<Vector3> p2(1, path2[0]);
                    pipe.set(p2, circle);

                    pipe.addPathPoint(path2[path2.size() - 1]);

                    drawPipe();
                }
            }
        }
    }
}

```

```

    }
}

// Render printed current layer
void CurrPrinted()
{
    for (int pnt = 0; pnt < nxt_pnt[renderLyr] - 1; pnt++)
    {
        bool y = false;

        float startpoint[3] = {
            order_list[renderLyr][pnt][renderContr[renderLyr]][0],
            order_list[renderLyr][pnt][renderContr[renderLyr]][1],
            order_list[renderLyr][pnt][renderContr[renderLyr]][2]
        };
        float endpoint[3] = {
            order_list[renderLyr][pnt + 1][renderContr[renderLyr]][0],
            order_list[renderLyr][pnt + 1][renderContr[renderLyr]][1],
            order_list[renderLyr][pnt + 1][renderContr[renderLyr]][2]
        };

        path3 = buildPath(startpoint, endpoint);

        // To turn the cylinder direction
        float theta, OA;
        OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)
        theta = float(abs(atan(OA)) * 180 / pi);

        if (theta < 45)
        {
            y = true;
        }

        // Configure the circle for path

        circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

        std::vector<Vector3> p3(1, path3[0]);
        pipe.set(p3, circle);
        pipe.addPathPoint(path3[path3.size() - 1]);

        drawPipe();
    }
}

// Render the finished printing layer
void CurrLayer()
{

```



```

    for (j = 0; j < contour_[renderLyr]; j++)
    {
        for (int pnt = 0; pnt < pntNum_order[renderLyr][j] -
1; pnt++)
            {
                bool y = false;

                float startpoint[3] = {
                    order_list[renderLyr][pnt][j][0],
                    order_list[renderLyr][pnt][j][1],
                    order_list[renderLyr][pnt][j][2]
                };
                float endpoint[3] = {
                    order_list[renderLyr][pnt + 1][j][0],
                    order_list[renderLyr][pnt + 1][j][1],
                    order_list[renderLyr][pnt + 1][j][2]
                };

                path7 = buildPath(startpoint, endpoint);

                // To turn the cylinder direction
                float theta, OA;
                OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)
                theta = float(abs(atan(OA)) * 180 / pi);

                if (theta < 45)
                {
                    y = true;
                }

                // Configure the circle for path

                circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

                std::vector<Vector3> p7(1, path7[0]);
                pipe.set(p7, circle);
                pipe.addPathPoint(path7[path7.size() -
1]);

                drawPipe();
            }
    }

// Preview of the printed object
void renderPreview()
{
    for (int lyr = 0; lyr < slicedVetx_num[999][1]; lyr++)
    {

```

```

        for (int cont = 0; cont < contour_[lyr]; cont++)
        {
            for (int pnt = 0; pnt < pntNum_order[lyr][cont] -
1; pnt++)
                {
                    bool y = false;

                    float startpoint[3] = {

order_list[lyr][pnt][cont][0],
order_list[lyr][pnt][cont][1],
order_list[lyr][pnt][cont][2]
                    };
                    float endpoint[3] = {

order_list[lyr][pnt + 1][cont][0],
order_list[lyr][pnt + 1][cont][1],
order_list[lyr][pnt + 1][cont][2]
                    };

                    path4 = buildPath(startpoint, endpoint);

// To turn the cylinder direction
float theta, OA;
OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)

                    theta = float(abs(atan(OA)) * 180 / pi);

                    if (theta < 45)
                    {
                        y = true;
                    }

// Configure the circle for path

                    circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

                    std::vector<Vector3> p4(1, path4[0]);
                    pipe.set(p4, circle);

                    pipe.addPathPoint(path4[path4.size() - 1]);

                    drawPipe();
                }
        }

int lyri = slicedVetx_num[999][1] - 1;

```

```

if (lyri > 0)
{
    for (int j = 0; j < contour_[lyri]; j++)
    {
        for (int k = 0; k < pntNum_infill[lyri][j] - 1; k++)
        {
            bool y = false;

            float startpoint[3] = {
                infill[lyri][k][j][0],
                infill[lyri][k][j][1],
                infill[lyri][k][j][2]
            };
            float endpoint[3] = {
                infill[lyri][k + 1][j][0],
                infill[lyri][k + 1][j][1],
                infill[lyri][k + 1][j][2]
            };

            path9 = buildPath(startpoint, endpoint);

            // To turn the cylinder direction
            float theta, OA;
            OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)

            theta = float(abs(atan(OA)) * 180 / pi);

            if (theta < 45)
            {
                y = true;
            }

            // Configure the circle for path
            circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

            std::vector<Vector3> p9(1, path9[0]);
            pipe.set(p9, circle);

            pipe.addPathPoint(path9[path9.size() - 1]);

            drawPipe();
        }
    }
}

```

```

// Render infill
void renderInfill()
{
    infillLyr = renderLyr;
    bool y = false;
    iDonePrint = false;

    if (infillLyr < slicedVetx_num[999][1] && infillLyr >= 0)
    {
        if (infillContr[infillLyr] < contour_[infillLyr])
        {
            if (pnt_Infill[infillLyr] < pntNum_infill[infillLyr][infi
llContr[infillLyr]] - 1)
                {
                    float startpoint[3] = {

                        infill[infillLyr][pnt_Infill[infillLyr]][infillContr[infi
llLyr]][0],

                        infill[infillLyr][pnt_Infill[infillLyr]][infillContr[infi
llLyr]][1],

                        infill[infillLyr][pnt_Infill[infillLyr]][infillContr[infi
llLyr]][2]
                    };
                    float endpoint[3] = {

                        infill[infillLyr][pnt_Infill[infillLyr] + 1][infillContr[
infillLyr]][0],

                        infill[infillLyr][pnt_Infill[infillLyr] + 1][infillContr[
infillLyr]][1],

                        infill[infillLyr][pnt_Infill[infillLyr] + 1][infillContr[
infillLyr]][2]
                    };

                    path5 = buildPath(startpoint, endpoint);

                    // To turn the cylinder direction
                    float theta, OA;
                    OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)

                    theta = float(abs(atan(OA)) * 180/ pi);

                    if (theta < 45)
                    {
                        y = true;
                    }

                    // Configure the circle for path

                    circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

```

```

std::vector<Vector3> p5(1, path5[0]);
    pipe.set(p5, circle);

    ++currIndex;
    if (currIndex < path5.size())
    {

pipe.addPathPoint(path5[currIndex]);
    }
    else
    {

currIndex = 0; // Reset the current index

pnt_Infill[infillLyr] ++; // to next point
    }
    }
    else
    {
        infillContr[infillLyr]++;
    }
}
else
{
    renderLyr++;
    iDonePrint = true;
}
}
drawPipe();
}

// Render the printed infill on current layer
void CurrInfill()
{
    for (int pnt = 0; pnt < pnt_Infill[infillLyr]; pnt++)
    {
        bool y = false;

        float startpoint[3] = {

infill[infillLyr][pnt][infillContr[infillLyr]][0],
infill[infillLyr][pnt][infillContr[infillLyr]][1],
infill[infillLyr][pnt][infillContr[infillLyr]][2]
        };
        float endpoint[3] = {

infill[infillLyr][pnt + 1][infillContr[infillLyr]][0],
infill[infillLyr][pnt + 1][infillContr[infillLyr]][1],
infill[infillLyr][pnt + 1][infillContr[infillLyr]][2]
        };

        path6 = buildPath(startpoint, endpoint);

        // To turn the cylinder direction

```

```

        float theta, OA;
        OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)
        theta = float(abs(atan(OA)) * 180 / pi);

        if (theta < 45)
        {
            y = true;
        }

        // Configure the circle for path

        circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

        std::vector<Vector3> p6(1, path6[0]);
        pipe.set(p6, circle);
        pipe.addPathPoint(path6[path6.size() - 1]);

        drawPipe();
    }
}

// Render the printed infill (1 layer before the current layer o
nly)
void PrintedInfill()
{
    printedInfillLyr = renderLyr - 1;

    for (int j = 0; j < contour_[printedInfillLyr]; j++)
    {
        for (int pnt = 0; pnt < pntNum_infill[printedInfillLyr][j
] - 1; pnt++)
        {
            bool y = false;

            float startpoint[3] = {

infill[printedInfillLyr][pnt][j][0],
infill[printedInfillLyr][pnt][j][1],
infill[printedInfillLyr][pnt][j][2]
            };
            float endpoint[3] = {

infill[printedInfillLyr][pnt + 1][j][0],
infill[printedInfillLyr][pnt + 1][j][1],
infill[printedInfillLyr][pnt + 1][j][2]
            };

            path8 = buildPath(startpoint, endpoint);

            // To turn the cylinder direction
            float theta, OA;

```

```

        OA = (endpoint[1] -
startpoint[1]) / (endpoint[0] - startpoint[0]); // oa = (y2-
y1)/(x2-x1)
        theta = float(abs(atan(OA)) * 180 / pi);

        if (theta < 45)
        {
            y = true;
        }

        // Configure the circle for path

        circle = buildCircle(0.2f, CIRCLE_SECTORS, y); // radius,
segments

        std::vector<Vector3> p8(1, path8[0]);
        pipe.set(p8, circle);
        pipe.addPathPoint(path8[path8.size() -
1]);

        drawPipe();
    }
}

// Nozzle movement
void nozzle()
{
    float dist_n, dist_x, dist_y;
    int step;
    const float nozzle_position = 110; // initial position

    // Compute the distance between the first vertex and the
position of the nozzle
    dist_x = order_list[0][0][0][0] - nozzle_position;
    dist_y = order_list[0][0][0][1] - nozzle_position;
    dist_n = float(sqrt(pow(order_list[0][0][0][0] -
nozzle_position, 2) + pow(order_list[0][0][0][1] -
nozzle_position, 2)));
    step = int(0.1 * dist_n / X_step); // 0.1 is to reduce th
e numbers of step to the position (speed up)

    // Initial movement to the printing position
    if (renderMode)
    {
        if (count_n < step)
        {
            nx = nozzle_position + (dist_x * count_n / step);

            ny = nozzle_position + (dist_y * count_n / step);
            nz = 0;
            count_n++;
        }
        else // Moving along the printing path
        {
            Reach = true;
        }
    }
}

```

```

        ++index_n;
        if (index_n < pntNum_n)
        {
            nx = nozzle_path[index_n][0];
            ny = nozzle_path[index_n][1];
            nz = nozzle_path[index_n][2];

            //cout << nx << ", " << ny << ", " << nz << endl;
        }
        else
        {

            index_n = 0; // Reset the current index
        }
    }
    //nozzle tip

    glPushMatrix();

    GLfloat lmodel_ambient[] = { 5, 2, 0, 0.0 }; // set to yellow color
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

    glColor3f(1.f, 0.5f, 0.f);
    glTranslatef(nx, ny, nz + 0.2f);
    GLUquadricObj* tip = gluNewQuadric();
    //nozzle tip
    gluCylinder(tip, 0, 0.5, 0, 20, 20);
    glColor3f(1.f, 0.5f, 10.f);
    gluCylinder(tip, 0.5, 1.6, 2, 20, 20);
    glPopMatrix();

    //nozzle neck
    glPushMatrix();
    glColor3f(1.f, 0.5f, 0.f);
    glTranslatef(nx, ny, nz + 2.2f);
    GLUquadricObj* neck = gluNewQuadric();
    gluCylinder(neck, 0, 4, 0, 6, 1);
    gluCylinder(neck, 4, 4, 3, 6, 1);
    glPopMatrix();

    //nozzle screw
    glPushMatrix();
    glColor3f(1.f, 1.f, 0.f);
    glTranslatef(nx, ny, nz + 5.2f);
    GLUquadricObj* screw = gluNewQuadric();
    gluCylinder(screw, 4, 0, 0, 6, 1);
    glColor3f(1.f, 1.f, 0.f);
    gluCylinder(screw, 2.5, 2.5, 7.5, 20, 1);
    glPopMatrix();
    glPushMatrix();
    glColor3f(1.f, 1.f, 0.f);
    glTranslatef(nx, ny, nz + 12.7f);
    gluCylinder(screw, 2.5, 0, 0, 20, 1);
    glPopMatrix();
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
}

```



## Appendix C: Motion of Ender 3 3D Printer

A screenshot of a 3D printer's LCD screen. The screen is black with white text. At the top, the word "Motion" is displayed on the left and a small upward-pointing arrow symbol is on the right. Below this, there are four lines of text, each representing a different motion parameter and its value. The values are: Xsteps/mm: 80.00, Ysteps/mm: 80.00, Zsteps/mm: 400.00, and Esteps/mm: 93.00.

Motion	↑
Xsteps/mm:	80.00
Ysteps/mm:	80.00
Zsteps/mm:	400.00
Esteps/mm:	93.00

Figure C-1: X, Y, Z and extruder motion for an Ender 3 printer.