CONTOUR GENERATION FOR MASK PROJECTION STEREOLITHOGRAPHY 3D PRINTING

FAEIZ AZIZI BIN ADNAN

Master of Science

UNIVERSITI MALAYSIA PAHANG

UNIVERSITI MALAYSIA PAHANG

DECLARATION OF THESIS AND COPYRIGHT

Author's Full Name	: FAEIZ AZIZI BIN ADNAN
Date of Birth	: 6 AUGUST 1990
Title	: CONTOUR GENERATION FOR MASK PROJECTION STEREOLITHOGRAPHY 3D PRINTING
Academic Session	: SEM II 2018/2019

I declare that this thesis is classified as:

☐ CONFIDENTIAL	(Contains confidential information under the Official Secret Act 1997)*
☐ RESTRICTED	(Contains restricted information as specified by the organization where research was done)*
☑ OPEN ACCESS	I agree that my thesis to be published as online open access (Full Text)

I acknowledge that Universiti Malaysia Pahang reserves the following rights:

1. The Thesis is the Property of Universiti Malaysia Pahang
2. The Library of Universiti Malaysia Pahang has the right to make copies of the thesis for the purpose of research only.
3. The Library has the right to make copies of the thesis for academic exchange.

Certified by:

(Student's Signature)	(Supervisor's Signature)
900806-06-5377	Dr. Fadhlur Rahman Mohd Romlay
Date: 12/07/19	Date: 12/07/19

NOTE: * If the thesis is CONFIDENTIAL or RESTRICTED, please attach a thesis declaration letter.

SUPERVISOR'S DECLARATION

I hereby declare that I have checked this thesis, and in my opinion, this thesis is adequate in terms of scope and quality for the award of the degree of Master of Science

____

(Supervisor's Signature)

Full Name	: DR. FADHLUR RAHMAN BIN MOHD ROMLAY
Position	: SENIOR LECTURER
Date	: 12 July 2019

STUDENT'S DECLARATION

I hereby declare that the work in this thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at Universiti Malaysia Pahang or any other institutions.

____

(Student's Signature)

Full Name	: FAEIZ AZIZI BIN ADNAN
ID Number	: MFM15002
Date	: 12 July 2019

CONTOUR GENERATION FOR MASK PROJECTION STEREOLITHOGRAPHY 3D PRINTING

FAEIZ AZIZI BIN ADNAN

Thesis submitted in fulfilment of the requirements
for the award of the degree of
Master of Science

Faculty of Mechanical & Manufacturing Engineering
UNIVERSITI MALAYSIA PAHANG

JULY 2019

ACKNOWLEDGEMENTS

Here I would to express my deepest gratitude upon the thesis completion. It is all thanks to my supportive parent. They have been supporting me all way from the beginning of this study until the end. Without their support, I would not have managed to complete this thesis. Their encouragement keeps me motivated throughout the study.

I would also like to thank my supervisor Dr. Fadhlur Rahman Bin Mohd Romlay for his guidance during my study. The research field is totally new to me. For someone who has no prior knowledge and skill for research works, I often find myself lost and misdirected. Most of the time, Dr. Fadhlur patiently guided me and taught me how to be a good researcher.

Next, I would like to thank Siti Noor Sheeren Binti Manan for her supports on my study. She always kept pushing me to finish my study. This motivation helps me in pursuing the thesis completion. Her support also encourages me to complete this study.

This research is funded by Universiti Malaysia Pahang (UMP) by the research grant number RDU160387. I would like to thank UMP for the fund. Without it, this research would be difficult to complete. This fund helped in purchasing the components needed for this research. The fund also supported me to overcome my financial issues. Other than that, UMP also provided excellent equipment and facilities to assist the research. I wish that the government will increase their research budget allocations for UMP in the future so that many more high impact researches can be produced. Also, special thanks to Faculty of Manufacturing Engineering UMP for providing the space and equipment for this study.

ABSTRAK

Kemajuan terkini dalam teknologi pencetakan 3D telah membawa kepada penghasilan mesin pencetakan 3D berasaskan pancaran-bertopeng. Proses ini menggunakan tenaga cahaya UV bagi membentuk objek nyata dari resin penyembuhan-foto. Pancaran kontur dijanakan dengan mengiris model CAD STL kepada lapisan-lapisan kontur 2D yang kemudiannya disalurkan kepada alat pemancar lapisan demi lapisan berasaskan ketinggian binaan. Pengkomputan bagi penjanaan lapisan-lapisan kontur 2D adalah sangat intensif. Algoritma penjanaan kontur yang sedia ada memerlukan masa pengkomputan yang lama. Ini kerana algoritma tersebut perlu mengiris dan mengkomput setiap satu lapisan sesebuah model STL sebelum proses pencetakan bermula. Dalam usaha bagi mengurangkan masa pengkomputan, algoritma yang baru dan lebih pantas diperlukan. Lantaran itu, algoritma penjanaan kontur lantas dibentangkan di dalam kajian ini. Kaedah ini menghasilkan satu lapisan kontur secara lantas apabila parameter ketinggian binaan disuapkan ke dalam algoritma tersebut. Algoritma tersebut mengandungi beberapa algoritma seperti algoritma pengirisan, algoritma pemetaan garisan pixel, dan algoritma gelungan kontur. Algoritma pengirisan menggunakan model persilangan garisan-satah untuk menghasilkan segmen garisan rawak apabila ia menerima satu faset STL. Segmen-segmen garisan ini kemudiannya dipetakan berdasarkan resolusi alat pemancar dengan menggunakan algoritma pemetaan garisan pixel. Kemudian, garisan-garisan pixel tersebut dihubungkan untuk membentuk satu atau lebih gelungan kontur melalui algorithm gelungan kontur. Hasil dari setiap algoritma-algoritma tersebut dikaji secara mendalam. Pengukuran masa pengkomputan diambil menggunakan objek <QElapsedTimer> di dalam Qt Creator dan diukur dalam millisaat. Keputusan hasil kajian menyatakan algoritma-algoritma tersebut menjanakan lapisan-lapisan kontur dengan tepat. Malah dengan menggunakan model STL berpoligon tinggi, algoritma penjanaan kontur masih dapat menjanakan lapisan kontur secara purata 960.15% lebih pantas dari algorithm Park dan 169.15% lebih pantas dari perisian komersial Slic3r.

ABSTRACT

Recent advancement in 3D printing technology has led to the development of projection mask stereolithography 3D printing process. This process harnesses the power of UV light contour projection to cure photocurable resin. The contour projection is generated by slicing STL CAD model into layers of 2D contours which is then fed into the UV projection device layer-by-layer with respect to the build height. Generation of the layers are computationally intensive. Existing contour generation algorithm requires long computational time to generate the contour layers especially for high polygon models. This is because the existing approach has to slice and compute every single layer of the STL model before the printing process starts. In an effort to reduce the computational time, a new and faster algorithm is required. Thus, a real-time contour generation algorithm is presented in this research. The real-time contour generation approach instantly generates single layer of contour whenever the build height parameter is fed into the algorithm. The algorithm composes of multiple algorithms such as slicing algorithm, pixel line mapping algorithm, and the contour loop algorithm. The proposed slicing algorithm uses line-plane intersection model to generate arbitrary line segment when it receives an STL facet. These line segments are mapped based on the projection device display resolution by the pixel-line mapping algorithm. Then, the pixelated line segments are connected to form single/multiple contour loops using contour loop algorithm. The results of each algorithms are thoroughly evaluated. Computation time measurement is taken using <QElapsedTimer> object in Qt Creator and measured in milliseconds. It is later found that the algorithms able to correctly generates the contour projection layers. Even with the high polygon STL model, the contour generation algorithm able to perform on average 960.15% faster than Park algorithm and 169.18% faster than commercial software Slic3r.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

C_d	Cure Depth
D_p	Depth of Penetration
E_{max}	Maximum Energy of Laser
E_c	Critical Energy Dosage
α	Photochemical Parameter
β	Photonics Parameter
c	Speed of Light
h	Planck's Constant
N_{av}	Avogadro Constant
P_L	Laser Power
W_o	Beam Width
k_t	Termination Constant
k_p	Propagation Constant
p_c	Extent of Polymerization
ϵ	Molar Extinction Coefficient
λ	Wavelength
ϕ	Quantum Yield
PI	Photoinitiator Concentration
z_c	Cure Depth
x	X component
y	Y component
z	Z component
P_o	Starting point of the line segment
P_f	Ending point of the line segment
s	Interpolation parameter
AR	Aspect Ratio
AR'	Modified Aspect Ratio
R, V, W	Piecewise Variable
NC	Normalized Correlation
O	Big-O Notation

LIST OF ABBREVIATIONS

2D	Two-Dimensional
2PP	Two-Photon Polymerization
3D	Three-Dimensional
3DP	Binder Jetting
AM	Additive Manufacturing
ASCII	American Standard Code for Information Interchange
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CLIP	Continuous Liquid Interface Printing
CMM	Coordinate Measurement Machine
CNC	Computer Numerical Control
CPU	Central Processing Unit
CSV	Comma-separated Value
CT	Contour Time
DIW	Robocasting
DLP	Digital Light Processing
DMD	Digital Micro-mirror Device
EBM	Electron Beam Melting
ECC	Efficient Contour Construction
FDM	Fused Deposition Modeling
GB	Giga-Byte
IF	Intersecting Facet
LC	Loop Count
LM	Layered Manufacturing
LOM	Laminated Object Manufacturing
PC	Personal Computer
RAM	Random Access Memory
RP	Rapid Prototyping
SD	Standard Deviation
SLA	Stereolithography 3D Printing
SLM	Selective Laser Melting

SLS	Selective Laser Sintering
STL	STereoLithography CAD format
UV	Ultraviolet

CHAPTER 1

INTRODUCTION

1.1 DLP Projection Mask Stereolithography

Three-dimensional (3D) printing is an additive manufacturing (AM) process and also known as rapid prototyping (RP). Unlike conventional subtractive manufacturing method such as milling that cuts and removes material to manufacture the product, an additive manufacturing process performs the opposite of the milling method. Instead of removing material which cause material waste and tool weariness, the process stacks the material on top of one layer and another. This is also called as layered manufacturing (LM). Most of the material waste in 3D printing comes from its scaffold/support during the printing process which is minimal compared to subtractive manufacturing.

In 1981, 3D printing was firstly introduced by Hideo Kodama (Kodama, 1981). The study proposed a new method of fabrication using photopolymer which solidifies upon exposure to ultraviolet (UV) light source (Xenon lamp and Mercury lamp) controlled by XY interpolation mechanism for contour routing and elevated build plate for Z-axis. Ever since then, researches have revolutionized the methods of 3D printing. Table 1.1 shows the classification of 3D printing according to current technology of 3D printing.

Recent advancement in 3D printing leads to the development of Digital Light Processing (DLP) projection mask stereolithography which utilizes UV light to cure photocurable resin into solid model. Like conventional 3D printing, it is a layer-by-layer process. Instead of traversing along XY axis to construct the layer, the process uses contour projection-based curing technique to uniformly cure each layer. Thus, this improves the printing speed and maintain uniformity of the cured part. The printed part generated by this technique becomes monolithic due to continuous curing process. Thus, improving its mechanical properties and its quality. The DLP projection mask stereolithography is known to have the best printing quality compared to other 3D printing technique.

Table 1.1 Classification of 3D printer

Process	**Technique**	**Materials**
Extrusion	Fused deposition modeling (FDM)	Thermoplastics filament (ABS, PLA, etc.), glass, metal, etc.
	Robocasting (DIW)	Plastics, ceramic, food, organic cell, composites
Powder based	Selective laser sintering (SLS)	Thermoplastics, metals
	Selective laser melting (SLM)	Metals
	Electron beam melting (EBM)	Metals
	Binder jetting (3DP)	Any material in particulate form
Lamination	Laminated object manufacturing (LOM)	Sheets (paper, metal, plastic, etc.)
Photopolymerization	Stereolithography (SLA)	Photopolymers
	Material jetting	Photopolymers
	Continuous liquid interface printing (CLIP)	UV-curable resins
	Two-photon polymerization (2PP)	UV-curable resins

Source: Hemant et al. (2015); Wong et al. (2012)

All methods stated in Table 1.1 share similarities in its process thread or also known as digital manufacturing pipeline. Before any of the printing process can takes place, a Computer Aided Design (CAD) file containing the information of the desired geometry will undergo a tessellation process that converts it into STL formatted file. Contour generation algorithm is then implemented to slice the 3D model of STL file into layers of contours which can be used for toolpath computation (for multi-axis 3D printer) or layer projection (projection-based 3D printer). The STL file and contour generation algorithm are considered as standard process flow for any 3D printing process.

1.2 Contour Generation Algorithm in Projection Mask Stereolithography

Contour generation process involves multiple algorithms to be implemented. First, the process starts with slicing algorithm which slices each facet of an STL file into multiple line segments with respect to the slicing height. Next, the process uses the generated line segments to connect each line segment into one or more closed contour loops using contour loop algorithm. Finally, a contour filling algorithm shades the closed loop contours to form a mask which cures the photopolymer or UV curable resin. In the past, researchers implemented the contour generation algorithm at the process planning stage (Choi & Kwok, 1999; Pandey et al., 2003; Zhang & Joshi, 2015). Each level of contours is generated before the printing process took place. However, in order for the printed model to appear seamless, the slicing thickness must be very small. This consumed a lot of memory utilized by the thousand layers of contours for the model to appear seamless. Another flaw for this approach is that the possibility of backlash of the elevation mechanism of Z-axis. For an open loop system, stepper motor is often used as the main actuators. A stepper motor usually has the tendency to misstep at a point when the rotor lag. This causes error in layer projection due to error in elevation height hence affects the printed model.

There are two types of slicing algorithm which are: uniform slicing and adaptive slicing. Adaptive slicing is an advanced slicing method which varies the slicing height depending on the features of the geometry. The algorithm works differently than uniform slicing. It performs comparison between layers and varies the slicing thickness depending on the geometry features to generate close approximation of the 3D model. In both slicing algorithms, issue of cusp height also commonly known as staircase effect often affecting the surface roughness of the printed model. Figure 1.1 shows rough edges that appear visible to naked eye if the layer resolution is low. This happens due to the DLP 3D printer works in single Z-axis. The layer cures vertically as the build platform elevates upward and the projected contour remains unchanged until it reaches the height for next contour. Instead of smooth slope transition between layers, the layer cures into stack of layers. Past study shows that layer stacking weakens the mechanical strength of the printed model especially when the layer resolution is low (Dizon et al., 2018; Lederle et al., 2016). Seamless layer formation is achievable by continuously generates new contour with respect to the smallest change of elevation height. The resolution of the printing output is subjected to the printer mechanism itself such as the pitch of the lead/ball screw, its diameter, and the resolution of the motor rotation.

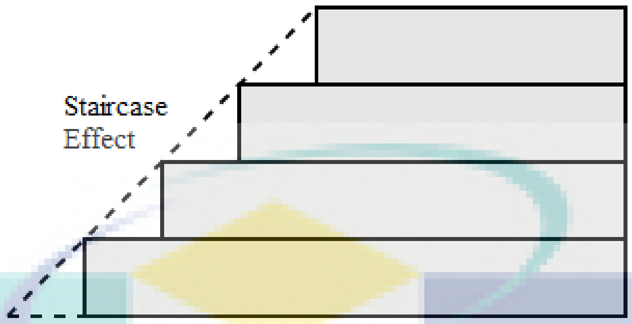


Figure 1.1 Staircase effect caused by uniform slicing thickness

The issue with STL slicing has been addressed multiple times by the rapid prototyping research community. But, most of the issue addressed mainly focuses on the quality of the printed model, improvements on the slicing process, and memory usage. None of the researches addressed the issue of computational time for the slicing algorithm which can consume up to 60% of the entire process planning time (Gregori et al., 2014; Kirschman & Jara-Almonte, 1992). Optimizing the computational time taken for slicing algorithm can improves the performance of the DLP 3D Printer and allows the contour generation algorithm to be implemented in real-time.

1.3 Problem Statement

Mask projection stereolithography process is a layer stacking process. Each layer is cured one by one until the printing process completed. In mask projection stereolithography printing process, these layers become monolithic due to continuous curing process. The contour layers are generated by intensive computational process. However, existing contour generation algorithm requires long computational time due to every layer had to be computed before the printing process. Higher resolution printing will require more computational time. More computational time is also required for high polygon STL model. Thus, a real-time contour generation algorithm is presented in an effort to reduce the computational time to generate the contour layer for mask projection stereolithography 3D printing process.

1.4 Research Objectives

The following objectives is developed to achieve the aim of the study. Objectives are classified into three stages which are:

i. To develop the real-time contour generation algorithm for projection mask stereolithography 3D printing process based on STL CAD model

ii. To evaluate the performances of the proposed algorithm based on computational time measurement

iii. To validate the performances of the proposed algorithm against literature

1.5 Research Scope

The scope of this research covers the projection aspect of the DLP 3D printing process. The algorithm for developing the contour projection is thoroughly studied and measured based on its computational time. Generated contour is directly generated from a raw STL model without any support generation algorithm. Each model tested are sliced with respect to only Z-component of the printer. This research does not cover the slicing process with different slicing orientation. The main objective is the development of real-time contour generation algorithm which will give results of the generated contour layers based on specific STL model. This will be thoroughly studied and discussed. Next, in order to evaluate the performance of the algorithm, execution time measurements of the algorithms are recorded. Finally, to results of computational time measurements are compared with the result obtained from the journal using similar STL model and same specifications for the workstation.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter describes critical review on process of photopolymerization in order to get better understanding on photopolymerization process before the implementation of the contour generation algorithm. The understanding of the photopolymerization chemistry will contributes on how the algorithm should be constructed. Other than that, this chapter also discusses on previous works done by other researchers in slicing and contour loop algorithms to develop the best approach in constructing the algorithm. The methodology and analysis which were developed by other researchers can be useful to support this work. Literature review on algorithms also give fundamental knowledge on how the slicing and contour generation algorithm work.

2.2 Mask Projection Stereolithography

The mask projection stereolithography is an additive manufacturing technique which harness UV-light projection to solidify photocurable resin into solid model. This method does not require any tooling or fixture as in milling process (Mu et al., 2017). The difference between the mask projection and traditional stereolithography process is that the use of digital micromirror device (DMD) by Texas Instrument to generate the projection (Pan et al., 2012). Traditional Stereolithography (SLA) process requires CNC routing for traversing the UV laser beam onto the resin to build each layer. This is time consuming due to traversing laser beam. Instead of traversing the laser, mask projection projects the whole contour onto the resin and uniformly cures the layer.

2.2.1 Photopolymerization

The process of polymerization using photopolymer is called photopolymerization process. Photopolymer usually consists of oligomer/binder, photoinitiator, and monomer. Typical photopolymer mixture contains at about 50-80% of oligomer, 10-40% monomer, and the rest of the portion is photoinitiator. In photopolymer, the oligomer usually used as ink, adhesives, and coating purpose. There are several families of oligomer which are: Methacrylate, Styrene, Vinylalcohol, Olefine, Polypropylene, and Glycerol family (Pandey, 2014). The oligomer also defines the basic property of the photopolymer such as glass transition, stress-strain, and adhesion. Meanwhile, the monomer defines the wetting property, crosslink, elasticity, and the viscosity. The photoinitiator formulation usually around 0.1-5% of the whole composition of photopolymer (Kitano, 2012).

Photoinitiator is highly reactive substance to light exposure usually UV light. There are also studies have been conducted for visible light photopolymerization (Gao et al., 1999). There are two types of photoinitiator: radical and cationic. Upon exposure to UV light, the photoinitiator generates free radicals that react with the monomers to form reactive species. Reactive species forms chain with another monomer causes chain reaction which forms the polymer. This chain reaction terminates when a reactive species reacts with each other forming dead radicals. Oxygen inhibition also causes this chain reactive to stop. When the oxygen reacts with the reactive radical, it forms an unreactive peroxide that terminates the chain reaction (Boddapati, 2010; Dendukuri et al., 2008).

2.2.2 Curing Depth Model of Photopolymerization

The photopolymerization curing depth model defines the fundamental equation governing the relationship between irradiance and the chemical reaction of the photopolymerization process. Back in 1992, Jacobs presented the standard design equation of stereolithography using Beer-Lambert law. The standard design equation presented is as follows:

$$C_d = D_p \ln\left(E_{max}/E_c\right) \qquad 2.1$$

where C_d is the curing depth of the resin. D_p is the depth of penetration which governs by Beer-Lambert law that suggests the irradiance at the resin surface is reduced by $1/e$ with respect to depth of the resin due to light absorption by the resin. E_{max} is the maximum energy of the laser, and E_c is the critical dosage of the resin (Jacobs, 1992).

The study on photocuring model of stereolithography also has been done by Lee et al. (2001). The study focuses on derivation of the photocuring mathematical model and incorporates both photochemical properties and the light intensity as the curing parameter. Multifunctional monomer that has been used in the study was *2,2-bis{4-[2-hydroxy-3-(methacryloxy)propoxy]phenyl}-propane* (Bis-GMA). Photoinitiator that has been used was *2-benzyl-2-N,N-(dimethylamino)-1-(4-morpholinophenyl)-1-butanone* (DBMP). In the experiment, the photopolymer mixtures were exposed to scanning He-Cd 325 nm UV laser. The photopolymer contains the mixtures of DBMP which was varied from 0.34 until 99.70 mmol/l that corresponded to 0.01 to 3.00 wt% of the solution. The conducted experiments also varied the laser dosage ranging from 0.931, 1.702, and 22.255 J/cm^2. It was found that the concentration of photoinitiator in the photopolymer enhances the cure depth but only up to its critical point before the reaction rate starts to plateau. It was due to high concentration of photoinitiator that limits the UV laser penetration depths. High photoinitiator concentration gives greater photon absorption but localizes the free radical concentration near the surface of the resin thus limiting the laser penetration. The authors distinguished the photochemical parameters and the photonics parameters as α and β which were derived as:

$$\alpha^2 = \frac{k_t[\ln(1 - p_c)]^2}{{k_p}^2 \phi \epsilon} \qquad 2.2$$

$$\beta^2 = \frac{chN_{av}P_L}{\lambda {W_o}^2 (2\pi)^{1/2}} \qquad 2.3$$

where in Equation 2.2, the k_t represents the termination constant and k_p is the propagation constant of the photopolymerization process. The p_c is the extent of polymerization. Molar extinction coefficient, ϵ of the DBMP which has been used is 23000 M^{-1}cm^{-1}. Whereas, the ϕ represents the quantum yield of the photoinitiator. Together, these parameters describe the photochemical terms of the photopolymer in a single non-dimensional variable, α. Equation 2.3 describes the photonics term of the UV laser exposure with λ as the wavelength of the laser emission, W_o as the beam width, the c is the speed of light, h as the Planck's constant, N_{av} is the Avogadro constant, and P_L as the laser power. Using both parameters, the authors have derived the equation that defines the cure depth as the function of both photochemical and photonics parameters as state in the equation below:

$$z_c = \frac{2}{2.303\epsilon[PI]} \ln\left(\frac{E_{max}[PI]^{1/2}}{\alpha\beta}\right) \qquad 2.4$$

Equation 2.4 is the derivation of the cure depth based on the photochemical and photonics parameters. In the equation, [PI] stands for the photoinitiator concentration and E_{max} represents maximum energy per unit area of the laser exposure. The presented cure depth model in Equation 2.4 is equivalent to the model presented by Jacobs (1992) in Equation 2.1. The authors also presented a 3D map of the curing relationship between the photoinitiator concentration and the energy dosage with respect to the curing depth.

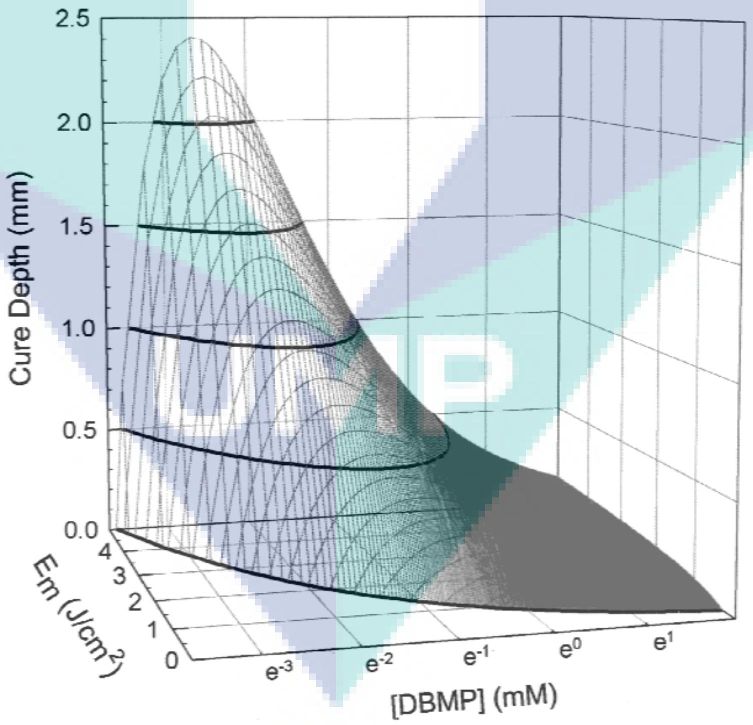


Figure 2.1 Surface topology of the curing space
Source: Lee et al. (2001)

Figure 2.1 shows that the increase in energy dosage will increase the cure depth. The same goes for photoinitiator concentration. At the beginning, increasing the photoinitiator concentration, rapid increase of the cure depth can be seen. However, up to some point, the cure depth starts to plateau with respect to increasing photoinitiator concentration (Lee et al., 2001). The surface topology that has been presented helps researchers to develop an optimal photopolymer formulation and algorithms for stereolithography 3D printing process.

In 2005, a study was conducted on stereolithography cure process modelling (Tang, 2005). In his work, the author claims that previous curing model presented by Jacobs is an oversimplification of the whole process. The model presented by Jacobs only considers the exposure threshold terms whilst disregarding the effect of photochemical process as presented by Lee. The author also stated that the process of photopolymerization is an exothermic. It means that the process generates heat during the reaction. Plus, the photopolymer resins often have low thermal conductivities. This causes thermally initiated polymerization to occur which reduces the resolution of the printed model and causing thermal stresses on the printed model. Hence, the mathematical model which incorporates the photopolymerization, mass diffusion, and heat transfer were developed starting with consideration of single axis laser scanning along X-axis on X-Z plane. The curing profile of a single axis laser scanning is shown in Figure 2.2 and Figure 2.3 below.

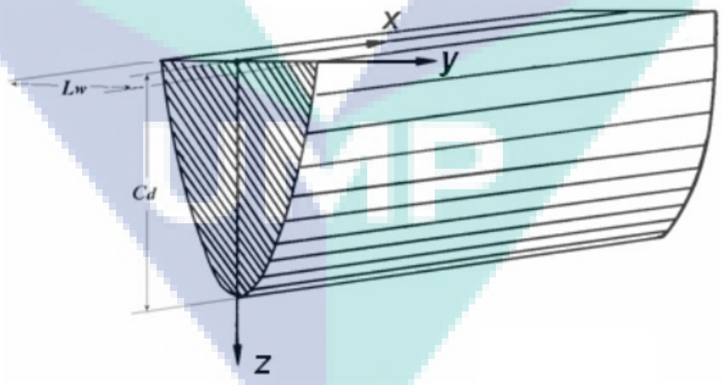


Figure 2.2 Curing profile of single axis laser scanning
Source: Tang (2005)

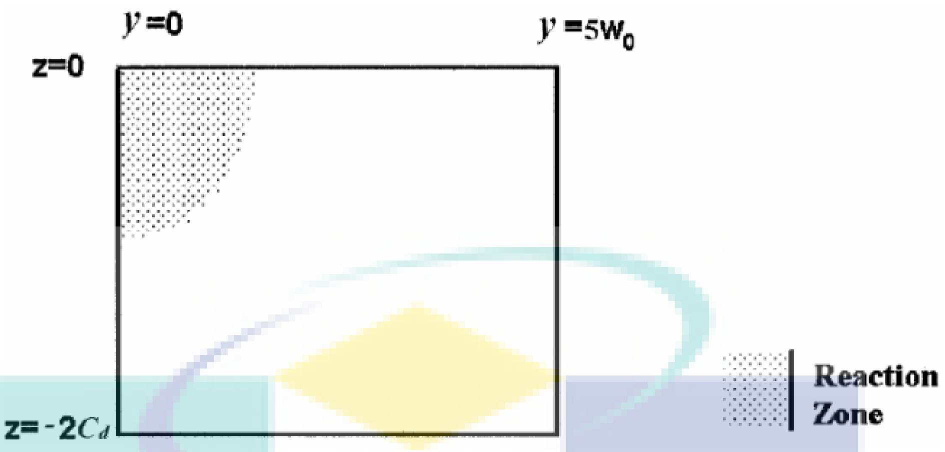


Figure 2.3 Domain of single axis laser scanning model
Source: Tang (2005)

Since the curing profile is assumed to be symmetrical, only half of the laser beam is taken as the domain. Derivation of the curing model is based on the energy balance, mass balance for the monomer, and the mass balance of radicals as shown in equations below:

$$\rho C_P \frac{\partial T}{\partial t} = k\left\{\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right\} + \Delta H_P R_P \tag{2.5}$$

$$\frac{\partial [M]}{\partial t} = D_M\left\{\frac{\partial^2 [M]}{\partial x^2} + \frac{\partial^2 [M]}{\partial y^2} + \frac{\partial^2 [M]}{\partial z^2}\right\} + (-R_P) \tag{2.6}$$

$$\frac{\partial [P\bullet]}{\partial t} = D_{P\bullet}\left\{\frac{\partial^2 [P\bullet]}{\partial x^2} + \frac{\partial^2 [P\bullet]}{\partial y^2} + \frac{\partial^2 [P\bullet]}{\partial z^2}\right\} + (-R_i) \tag{2.7}$$

Free radical photopolymerization kinetic models are presented with the derivation based on photochemical reaction during initiation phase, propagation phase, and termination phase. The reaction is described as:

$$\begin{aligned}
&PI \xrightarrow{hv} R\bullet && \text{Initiation}\\
&M + R\bullet \xrightarrow{k_i} P_1 && \\
&P_n\bullet + M \xrightarrow{k_p} P_{n+1}\bullet && \text{Propagation}\\
&P_n\bullet + P_m\bullet \xrightarrow{k_{tc}} M_{n+m} && \text{Termination by Combination}
\end{aligned} \tag{2.8}$$

$$\mathrm{P_n} \bullet + \mathrm{P_m} \bullet \xrightarrow{\mathrm{k_{td}}} \mathrm{M_n} + \mathrm{M_m}$$ Termination by Disproportionation

$$R \bullet + In \xrightarrow{k_{in}} Q$$ Inhibition

where in the Equation 2.8 above, the PI is the photoinitiator that decays upon exposure to light energy into the initial radicals [R •]. The radical reacts with a monomer [M] to start a polymer chain $\mathrm{P_n}$ • in the initiation phase. The polymer chain propagates to react with another monomer forming longer polymer chain. There are 3 cases of termination of the propagation phase. Either it is caused by reaction with another polymer chain by combination or disproportionation, or radicals inhibition commonly caused by oxygen inhibition that forms a non-reactive peroxy (Tang, 2005; Tang et al., 2004). Similar work has been done by Boddapati (2010). The author used the same principle but also incorporates oxygen inhibition model in the curing depth model.

Kang et al. (2012) presented pixel-based curing model for projection-based stereolithography printing process. The model is developed by applying Beer-Lambert law to model the depth of light penetration through liquid curable resin. Gaussian distribution is also used to model the light distribution profile. The light distribution is constrained to the square pixel shape of the projection device. The mathematical model of pixel-based curing includes time, critical energy dosage, light intensity, penetration depth, and other photochemical parameters. These are the important parameters that needs to be taken into account when developing the contour generation algorithm because the curing model will define the final shape of the printed model.

Tumbleston et al. (2015) presented slightly different curing model but with oxygen inhibition taken into consideration. The model is also based on Beer-Lambert law which model the depth of penetration of the light. The curing technique in the authors work on continuous liquid interface production takes advantages of the oxygen inhibition to accurately control the curing and provides continuous layer separation. Thus, this allows the printer to continuously cure every contour layer and allows faster printing time. The oxygen inhibition is modelled as dead zone which is a controlled uncured region for each layer.

2.3 STL Format

STereoLithography (STL) is a CAD file format that was developed by Albert Consulting Group for 3D Systems. The format was introduced as a means to transfer CAD data into rapid prototyping machine when Chuck Hall invented the first stereolithography (SLA) 3D printing machine back in 1987. Since then, STL has become a de facto in rapid prototyping industry and still widely supported by modern CAD software such as Autodesk, SolidWorks, Blender, CATIA, Rhinoceros 3D, and several other CAD software (Cătălin IANCU et al., 2010; Jacob et al., 1999; Królikowski & Grzesiak, 2014; Wu & Cheung, 2006). The popularity gained was due to its non-encrypted data, open-source, and simplicity (Hayasi & Asiabanpour, 2009). Most of other CAD formats are encrypted and licenses are required for the software developer to incorporate the CAD format compatibility in their applications.

STL is also known as the abbreviation for "Standard Tessellation Language" by some scholars. It is because the STL file is constructed using a tessellation process. Tessellation is a process that converts the surface geometry of a CAD model into meshes of small triangle. This triangle is called Facet. It has three vertices in 3D Cartesian Coordinate System that form the triangle. Together all the Facets made up a shell representation of the original CAD model. Tessellation process can also be applied to point clouds data usually obtained from Coordinate Measuring Machine (CMM) to construct an STL model. This is done by connecting all the point clouds into triangular mesh to construct the meshed surface geometry of the model (Cătălin IANCU et al., 2010; Koc et al., 2000; Tyvaert et al., 1999; Wu & Cheung, 2006). Thus, this make the STL formatted CAD models more robust and simpler.

2.3.1 Types of STL

The STL has two different types of data format which are ASCII and Binary. The ASCII STL format are human readable text format. ASCII STL format begins with *solid name* syntax. Usually, a model name is optional and often omitted with white spaces. Next, the syntax followed by *facet* syntax along with its normal vector coordinates. Vertices are enclosed with outer loop and *endloop* syntaxes. The vertex indicates a beginning for each vertex which are used as $P_1(x,y,z)$, $P_2(x,y,z)$, and $P_3(x,y,z)$

respectively in the proposed algorithms. The n and v is a formatted floating number of *sign-mantissa-"e"-sign-exponent*, e.g. "2.999381e-002" separated with white spaces. Each facet data will end with an *endfacet* syntax. Depending on the complexity of the geometry, an STL file may consists of more than one facet; usually thousands. When a new facet syntax is located after the previous *endfacet* syntax, this indicates the start of a new facet. Finally, an STL file normally ends with *endsolid* name syntax (Cătălin IANCU et al., 2010; Wu & Cheung, 2006). An example of ASCII STL format is shown below.

```
solid name
facet normal nᵢ nⱼ nₖ
   outer loop
         vertex v1ₓ v1ᵧ v1𝓏
         vertex v2ₓ v2ᵧ v2𝓏
         vertex v3ₓ v3ᵧ v3𝓏
   endloop
endfacet
endsolid name
```

Due to ASCII STL using ASCII text as its data, it often has larger file size compared to its Binary counterpart.

On the other hand, Binary STL file uses structured data format using binary representation of the data. The data can be read in Bytes with the first 80 Bytes of the Binary STL file is the header of the file. Most of the time, the first 80 Bytes are skipped to improve the reading time. In some cases, the header section contains the metadata of the STL file which is not as important as the facets data. After that, Binary STL contains another 4 Bytes of data that represents the facets count of the STL file. The facets count is read as Unsigned Integer data type in programming code. Then, the facets data starts with 12 Bytes of Normal vector data in which each 4 Bytes are the vector components for X, Y, and Z respectively. Each vector components are read and casted as Float data type. Next, the following 12 Bytes of data contains the first vertex of the facet with each 4 Bytes as its vector components similar to the Normal vector. The second and third vertex follow similar structure to the Normal and first vertex data structure. Then, the Binary STL allocated another 2 Bytes for attribute data for the facet. Overall, each facet data has the size of exactly 50 Bytes. Each 50 Bytes until the end of Binary STL file contains only the facet data of the STL model (Cătălin IANCU et al., 2010). The structure of Binary STL file is shown below.

```
Byte[80]       - Header
Byte[4]        - Facets Count

For each facet
   Byte[12]    - Normal vector(x, y, z)
   Byte[12]    - First Vertex (x, y, z)
   Byte[12]    - Second Vertex(x, y, z)
   Byte[12]    - Third Vertex (x, y, z)
   Byte[2]     - Attribute
Loop
```

The Binary STL has several advantages over ASCII STL data format because the data is more compact and reading time is faster than the ASCII STL data format. The 4 Bytes facets count gives useful information regarding the STL model. The Binary STL file sizes are smaller than ASCII STL file.

Recent advancement in rapid prototyping technology demands more information from an STL model such as colour. Thus, in the work of DX Wang, they proposed a Colour STL format derived from Binary STL format. Using the 2 Bytes in the attributes, an RGB565 colour code was inserted to represent the colour of the specified facet as shown below.

```
Byte[80]       - Header
Byte[4]        - Facets Count

For each facet
   Byte[12]    - Normal vector(x, y, z)
   Byte[12]    - First Vertex (x, y, z)
   Byte[12]    - Second Vertex(x, y, z)
   Byte[12]    - Third Vertex (x, y, z)
   Byte[2]     - RGB565 Colour
Loop
```

These bytes have the range of 65536 different colour levels that can be coded (Wang et al., 2006). However, the Colour STL format is rarely found because of its limited colour palette and inaccurate representation of the model colouring caused by arbitrary triangular meshes.

2.3.2 Issues of STL

Problems that occur in STL format are still being discussed up until now by numerous researchers ever since it was introduced back in 1987. STL format is known to have issues with incorrect and inconsistency in its normal vector. This occurs when the CAD software generated facet normal vector differs from the calculated normal based on the facet vertices (Huang et al., 2002; Kumar & Dutta, 1997; Wu & Cheung, 2006). Most of the time, programmer would prefer calculated normal based on the facet vertices coordinates rather than the generated facet normal due to this inconsistency problem. Thus, the generated facet normal is often ignored or skipped.

Another known error that occurs in STL format is when there is a gap or crack between the facets as shown in Figure below. This error is caused by truncation error in the CAD software generated vertices. Each facet usually shares at least one of its vertices with another facet within close proximity. According to STL rule, for two adjacent facets, there will be two shared vertices (Barequet & Sharir, 1995; Bloomenthal, 1988; Huang et al., 2002; Leong et al., 1996; Piegl & Richard, 1995). The mismatch of these vertices due to truncation error forms a crack or hole in the tessellated model (Kumar & Dutta, 1997; Wu & Cheung, 2006). Although this error can be fixed using algorithms such as K-Nearest Neighbors (k-NN) algorithm, it is still less efficient compared to other CAD formats. The truncation error of the vertices also causes the facets to overlap due to incorrect vertex generated in either facet as shown in Figure below.

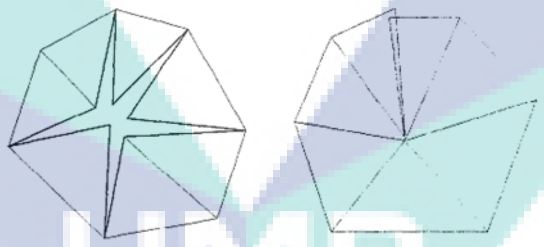


Figure 2.4 Hole at a vertex and overlapping facets
Source: Szilvśi-Nagy & Mátyási (2003)

Aside from the gap error and inconsistent normal, the major flaw in STL format is that every facet is generated in random order or arbitrarily. There are no pointers that show the relationship and proximity between each element (Szilvśi-Nagy & Mátyási, 2003). This leads to difficulty in processing the STL model since it will require complex algorithm to piece the facet together as if piecing a puzzle which is time consuming. This, in fact, lower the performances of the operation involving STL model. Some researchers suggested to use Octree data structure to correctly assign and store each facet for optimized slicing and other processes (Wong et al., 2017).

STL files are also known to consume large memory allocation to be stored. This make it less portable compared to other CAD formats (Wu & Cheung, 2006). Typical high polygon STL model consists of 1,175,288 facets has the file size of 56 MB in Binary STL and 273 MB in ASCII STL. In ASCII STL format, each chunk of data is stored as *char* or character which consume 1 Byte or 8 Bits for every chunk of data. This is wasteful for the case of numerical data. For example, each digit in the number "0.12345e+3" is individually regarded as *char* based on ASCII code. Thus, this number will consume 10 Bytes of memory. Although it is human readable, it is still inefficient in terms of resource. Thus, the Binary STL is developed in order to reduce this wasteful memory consumption by storing the numerical data in *float* data type which are 4 Bytes or 32 Bits. However, the Binary STL file size is still larger compared to other CAD formats. Redundancy of the STL vertices also contributes to the large STL file size.

Recent advancement in 3D application demands more information out of a CAD model. The information that often required by most modern CAD software nowadays demand information on the multiple material type, multiple colour information, surface texture, and etc. (Cătălin IANCU et al., 2010). This information which are lacking in STL model leads to its major downfall compared to other CAD formats which are more robust and practical. An attempt has been done to improve the STL format. One of it is the usage of 2 Bytes of attributes data to indicate the colour of the facet. However, the colour is only limited to 16-Bits colour RGB565 palette. The triangular shaped facet also causes inaccuracy in color representation of the STL model (Wang et al., 2006). Up until now, STL format is still unable to fulfill these new demands from the modern CAD software.

Based on the literature done on issues involving STL format, we can classify that there are two distinct cause of errors mentioned above. One, where the errors are caused by the CAD software generation process of STL format. The errors involving cracks and overlapping facets are caused by bad tessellation algorithm by the CAD software itself. Hence, it is unfair to regard it as a downfall of the STL model. These errors can be prevented if the CAD software performs a verification or linkage check algorithm on the generated STL model to detect the error. The other type of error that can be classified, is the limitations by the STL format itself such as the file sizes, arbitrary facets, and lack of required information. This is in fact, the major downfall of the STL format which has not been changed for the last 30 years since it was introduced.

2.4 Slicing Algorithm

Projection stereolithography 3D printing machine requires the 3D model to undergo process planning stage before the printing process. This process planning stage has a series of tasks which include: model orienting and positioning, slicing the model into 2D contours based on Z-axis of the printer workspace, and if necessary, add support structures (Kulkarni et al., 2000; Minetto et al., 2017). Slicing thickness is the crucial parameter that needs to be properly set as it defines the quality of the printed model. Large slicing thickness leads to "stair-case" effect. Small slicing thickness or higher slicing resolution provides accuracy and better printing quality but consumes larger memory and higher computational time. To overcome this issue, the slicing process must be computationally fast and efficient.

The process of converting triangular facet into line segment is called slicing process. The slicing process use an algorithm that relies on computation of the intersecting points between the slicing plane and the STL facet. Each facet is made of three vertices. When paired, the vertices become lines which form the triangle facet. When these lines intersect with the slicing plane, it will intersect at single intersection point. If two of the lines intersect with the slicing plane at the same time, connecting both intersection points form a line segment that exist on the slicing plane. An STL file contains multiple facets. Multiple interactions between the facet and the slicing plane form the 2D contour on the slicing plane that can be process into contour projection for DLP 3D printing process. In other application, these 2D contour can also be used for G-Code generation for CAD/CAM process in a CNC machine (Pandey et al., 2003).

2.4.1 Fundamental of Slicing Algorithm

The slicing algorithm relies on mathematical computation to compute the intersection points that form the line segments. It is derived based on line-plane intersection model in calculus math as represented by the Figure 2.5 below.

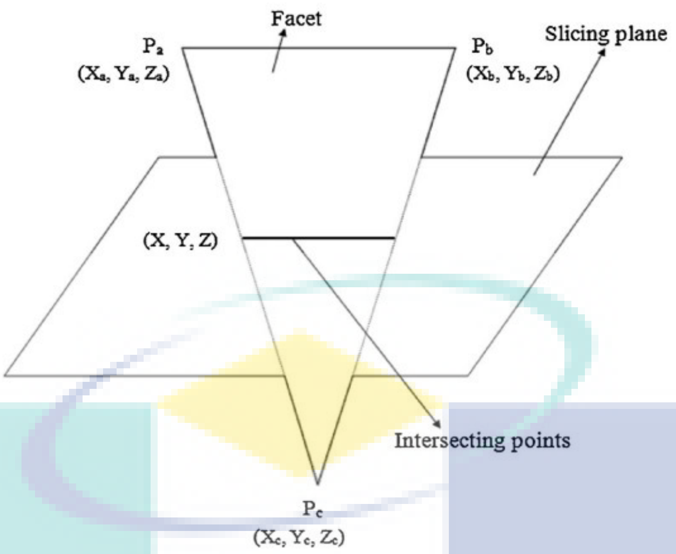


Figure 2.5 Facet intersecting with slicing plane
Source: Manmadhachary et al. (2016)

Figure 2.5 shows an STL facet intersecting with the slicing plane located at certain slicing height. The pair between the vertices P_a, P_b, and P_c are the lines intersecting with the plane. The two points that exist on the plane are the projected contour line of a single facet. This contour line is called as line segment. As can be seen, the line from P_a to P_b does not intersects with the plane, thus, no intersection point can be computed. To check whether the line intersects or vice versa, the height of the slicing plane must be in between the z coordinates of the two vertices. The closed loop contours at this particular slicing height are generated by multiple intersection between the STL facets at that slicing height (Manmadhachary et al., 2016). However, the set of these line segments are not programmatically connected. A contour loop algorithm is required to connect each of the generated line segment to form single/multiple closed loop contours.

In many literatures (Huang et. al, 2012; Hu, 2017), the most commonly used mathematical equation is the linear extrapolation method where the equation is defined as:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1} \qquad 2.9$$

In the above equation, the subscript 1 denotes the beginning of the line segment and subscript 2 denotes the end point of the line segment. By setting the slicing height, z, the unknown x and y can be solved. Thus, the intersection point is $P(x, y)$ as long the x and y exist in between P_1 and P_2 (Huang et al., 2012; Xu et al., 2017). This method is often used due to simple and fast computation. However, there are a few drawbacks of using this method. For example, the STL formatted CAD model uses fixed position vectors of its facets. Thus, to change the slicing orientation of the model will requires each position vectors to be modified. This process can be time consuming especially for high polygon model.

2.4.2 Facet-Plane Intersection Case Handling

The slicing algorithm involves intersection between the slicing plane and the STL facet. Studies have shown that certain type of intersection between these two causes geometrical errors and redundancies during the contour generation process (Jing Hu, 2017; Topçu et al., 2011). Thus, these facet-plane intersections are classified into 5 cases as shown in Figure 2.6 below. Each of the cases are treated with each respective case handling.

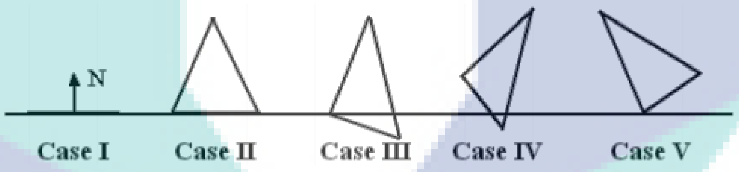


Figure 2.6 Possible intersection cases
Source: Topçu et al. (2011)

Case I describes the case where the facet is in parallel with the slicing plane. Thus, all sides intersect with the slicing plane. Usually facet with this case usually omitted because there will be another facet that shares the same side with the one in parallel. This normally occurs at the flat surface of the STL model. Commonly, at top and bottom side. Case I can also be used by directly storing all vertices as the contour points. But, to avoid redundancy of contour points, the facet is often omitted.

Case II describes the scenario where one side of the facet is in parallel with the slicing plane and two vertices intersect with the slicing plane. This case is usually handled by removing or ignoring the side in parallel with the slicing plane. The other two non-parallel sides are then sliced to generate the required line segment. Case II sometime shares its parallel side with the facet in Case I. Most of the time Case II is given priority over Case I.

Case III shows the facet intersects with the slicing plane at one side and one of its vertices. In this case, all sides are considered intersecting with the slicing plane. This issue will cause errors to the line segment generation because the line segment only requires two distinct points to form a line. Since the intersection happens at the vertex, two similar points will be generated. Removing one of the points can solve the issue.

Case IV shows an ideal case where only two sides intersect with the slicing plane. Slicing the two sides will produce only two distinct points that form the right line segment. The side that does not intersects is ignored.

Case V represents the occurrence where only one vertex of the facet touches the slicing plane. The algorithm might assume this condition as two sides intersecting with the slicing plane. Slicing this facet will produces two similar points. Thus, the line segment will end up becoming a single point in the 2D space. This leads to redundancies of contour points for contour generation process.

2.4.3 Data Structure

The STL files usually contain large quantity of facets information. These facets need to be properly managed so that the algorithm will performs better. Choosing the right data structure to store the facet information allows the algorithm to quickly access the necessary data needed without having to look into each element in the list. Aside from that, different STL models have different numbers of facet. Thus, the data structure should be able to scale itself to match the size of data. An array data structure requires fixed size allocation before the data can be stored. If the allocation size is too big, it will consume a lot of computer memory. On the other hand, if the size is too small, to program might crash due to array overflow when handling large STL file.

Huang et al. (2012) implemented hash table data structure in their work on slicing algorithm for G-Code generation for CNC Milling using STL file. The hash table stores the results of the slicing algorithm according to the incremental of the slicing height. The code is executed on a low-end PC operating on Intel Core 2 1.6 GHz RAM 2GB running on Windows XP SP3. The result shows that the execution time increases with respect to slicing thickness. Based on comparison, the result shows slight improvement than the original method. Considering that the program runs on low-end PC, the results are relatively fast with the implementation of hash table data structure. However, the test is implemented only on a single STL model. The cylinder-like shaped STL model always has single contour loop at each slicing height. STL model with multiple contour loops are not tested and reported in the journal.

Wong et al. (2017) utilized Octree data structure in their work on real-time slicing for light painting rendering application using STL formatted CAD model. The use of Octree data structure is mainly to reduce computational time for STL slicing. The algorithm first determines the axis-aligned bounding box of the STL model. The bounding box is set to be the root of the Octree structure. Then, the model is recursively subdivided into eight octants as the nodes of the tree. Each of these nodes contains a collection of facets of the STL model bounded by each respective node boundary. The algorithm is implemented using 4 different STL models having different number of facets. The number of tree levels are varied and the computational time of each cases are recorded. It is later found that, model with a greater number of facets requires more computational time to be sliced. Varying the tree level can reduce the computational time but only up to a certain limit. It is observed that after 3 tree level, the computational time started to rise due to more time is spent on the divide-and-conquer approach.

Pan et al. (2014) used linked list data structure in the development of rapid prototyping STL model slicing software. The linked list is used to store the facet and also contains a pointer that points to the next pairing facet. This kind of implementation can be advantageous since the algorithm does not need contour loop algorithm to connect each line segment because the facets are already arranged in such manner. But, since the slicing height varies, the pairing might also change. Thus, the list needs to be reconstructed which is also time consuming (Ye et al., 2017).

2.4.4 Type of Slicing Algorithm

Many methods have been developed on slicing algorithm to improve its computational time, accuracy, and memory efficiency. Among popular methods proposed by the researchers are uniform slicing, adaptive slicing, and direct slicing algorithm. These methods have its own advantages and limitations.

Uniform slicing has been popularized since the early years the slicing algorithm has been presented. It utilizes constant slicing thickness for all of the layers (Choi & Kwok, 1999). It is the simplest method for slicing approach. However, stair-case effect is known to occur when using this method. The stair-case effect is the case where there are losses of geometric data in between the slicing thickness interval since the fixed slicing thickness skipped these intervals. Some important features of the geometric model might be skipped which resulted in lower accuracy of the printed model. Reducing the slicing thickness can mitigate this effect (Zheng et al., 2018; Zhou et al., 2004) but the slicing output will consume more memory to store the slicing results.

In an effort to reduce the stair-case effect whilst reducing the memory consumption, adaptive slicing method is introduced. Adaptive slicing method uses variable slicing thickness that depends on the value of allowable cusp height. Pandey explained the concept of cusp height in their work on adaptive slicing algorithm. The cusp height is based on theoretical calculation of surface roughness and the build orientation. By limiting the allowable surface roughness parameter, variable slicing thickness can be obtained (Pandey et al., 2003). Zhou presented their work on non-uniform cusp height which is different than the work of Pandey. The non-uniform cusp height is based on circular approximation and user specified allowable cusp height as shown in Figure 2.7. The layer thickness model presented are able to solve the containment issues that occur during the printing process (Zhou et al., 2004).

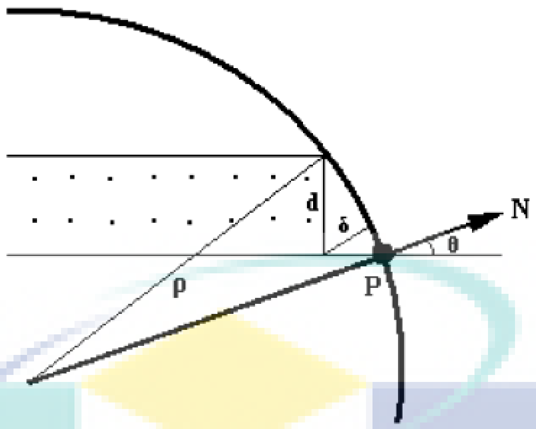


Figure 2.7 Circular approximation for layer error and thickness
Source: Zhou et al. (2004)

The adaptive slicing technique reduces memory consumption by also eliminating the repetitive features of the geometric model. For example, a cube STL model will always has the same contour from bottom to top when the slicing orientation is perpendicular to the cube. Thus, adaptive slicing eliminates the needs to reconstruct the same contour that can cause memory inefficiency.

Direct slicing algorithm is more recent approach in CAD slicing. This approach does not require tessellated CAD model such as STL format. Instead, the algorithm is implemented on the original CAD format without involving any tessellation process. This is because the tessellation process is a surface approximation process of the original CAD model. This approximation often leads to reduction in geometric accuracy (Jing Hu, 2017). Other reason the direct slicing algorithm is proposed due to the size of the STL file. Complex STL file often requires a lot of memory space to be stored compares to other CAD formats (Choi & Kwok, 1999). The direct slicing algorithm can be implemented using either uniform or adaptive slicing technique. The only difference is the CAD format.

2.5 Contour Loop Algorithm

In order to complete the contour generation for the projection mask stereolithography process, each line segments generated by the slicing algorithm must be connected to form closed-loop contours (Tian et al., 2018). These line segments are in arbitrary order due to STL facets are sorted in similar fashion (Zhang & Joshi, 2015). It is also possible to have multiple closed-loop contours at the same slicing height. Thus, it is crucial to differentiate to which group does a contour loop belongs to because the contours will define the geometrical features of the printed model.

One of the most common and naïve methods applied by the previous researchers are the head-to-tail search algorithm (Choi & Kwok, 1999; Wang et al., 2006). The algorithm works by joining neighboring line segments until closed-loop contour is formed. Each line segments contain two distinct points P_o to P_f. The P_o of the first line segments from the list is assigned as the head of the contour group. Then, the algorithm searches for the similar point that matches the P_f of the first line. The algorithm stops when the found P_f matches the assigned head. This indicates a closed-loop contour. Then, the remaining line segments are considered as new contour group and the algorithm assigns new head for the next contour group. The process will repeat until every line segment from the list is checked. The head-to-tail contour loop algorithm is known to have the worst case of $O(nk)$. This happen when the algorithm has to loop through each line segment from the list if the neighboring line is located at the end of the list. Since the k element decreases as n element increases, on average, this algorithm will run as $O(n)$. However, a study was done back in 2002 by Huang proves that STL formatted CAD model are susceptible to flaws such as cracks which may appear in between two side-by-side facets (Huang et al., 2002). Thus, the resulting line segments give incorrect pairs hence breaking the closed-contour loop formation. Even the smallest truncation errors between the pairs can be catastrophic to the head-to-tail contour loop algorithm.

An algorithm that uses shortest distance calculation is introduced to prevent the mispairing issue. This method is applied in the work of Manmadhachary in an attempt to improve surface smoothness of rapid prototyping printed medical product (Manmadhachary et al., 2016). The shortest distance approach eliminates truncation error that can cause contour dysconnectivity. The equation used for the shortest distance is defined as:

$$D = \sqrt{(x - x_t)^2 + (y - y_t)^2} \qquad 2.10$$

The equation above is used in comparison to compare the points of current line segment with the next line segment. If the point coincides, the value of D should be near or equal to zero (Vatani et al., 2009). However, the computation uses square root function which is more computationally intensive than normal mathematical operation. Thus, the shortest distance requires more computational time compared to the naïve head-to-tail search algorithm. It is a tradeoff between error-tolerance and the performance of the algorithm.

Zhang & Joshi introduced Efficient Contour Construction (ECC) algorithm in their work. The authors used linked list data structure for the ECC algorithm. The algorithm checks for the insertion position to construct the contour. The insertion process is decided by checking the first and last elements from the intersection linked list (Zhang & Joshi, 2015). The contour grouping process which differs between which group does the contour loop belongs is not clearly stated in the ECC algorithm. There could be more than one closed loop contour at different slicing height depending on the geometry features.

2.6 Summary

Mask projection stereolithography 3D printing process uses UV-light projection to cure photocurable resin. The curing process is called photopolymerization. Photopolymer resin used in this process contains 50-80% oligomer, 10-40% monomer, and the rest is photoinitiator. Each of the components in the photopolymer defines the properties of the printed model. Photoinitiator is the photo-reactive substance that initiates the polymerization process upon exposure to light with specific wavelength. The concentration of the photoinitiator in the photopolymer mixture highly affects the curing depth of the photopolymer as shown in Figure 2.1. Other parameters which affect the curing process include the light intensity, the critical energy dosage, time, and other photochemical parameters. This shows that the photopolymerization is a time dependent process. Thus, the proposed algorithm must be fast enough to keep up with the photopolymerization process. The elevation speed of the printer must also correlate to the curing speed in order to generate accurate printed model.

STL file is a de facto CAD format in 3D printing industry. There are two types of conventional STL format which are Binary STL and ASCII STL. Each STL model consists of multiple facets which are made of a normal vector and three vertices that form the facet. Ever since it was first developed back in 1987, STL format remains unchanged.

There are a lot of known issues with the STL format. Among the flaws associated with the STL format is the possibility of crack due to mismatch of the facet coordinates. The proposed algorithm must be able to correct this error since it can cause failures during the contour formation.

Slicing algorithm is an algorithm that slices the 3D CAD model into layers of contour. In mask projection stereolithography, the 3D model is sliced into multiple 2D contour layers. These layers are used in the mask projection to cure each layer of the 3D model with respect to the build height. The fundamental equation used in slicing process uses linear extrapolation method. This method is simple and straightforward. However, this method is susceptible to division by zero which may cause the program to crash. As discussed earlier in Section 2.4.2, there are several cases of interaction between the slicing plane and the facet. Each of these cases must be handled properly in order to generate the correct 2D contour representation of the 3D model. Most commonly studied slicing algorithm are the uniform slicing and adaptive slicing. However, these methods cause stair-case effect and requires long computational time. Since the curing process is continuous, the printer must also continuously track the changes of the curing depth hence modifies the contour layer according to the cure depth.

The process of STL slicing only generates multiple arbitrary line segments. Thus, a contour loop algorithm is needed to reconnect the line segments into one or multiple closed loop contours. Based on literature, many researchers proposed the head-to-tail search algorithm which is simple and naïve. Considering that there are cases of cracks occurring in the STL format, the naïve approach will not be sufficient due to error caused by slicing. Manmadhachary et al. (2016) proposed shortest distance approach to find and connect the line segments. But the shortest distance is more computational intensives. In this research, the proposed contour loop algorithm uses pixel line mapping algorithm to map the line segment based on pixel coordinate of the projection device and uses head-to-tail search algorithm to efficiently connect every line segment.

The issue of computational time in Contour Generation process is rarely been discussed by the rapid prototyping community. The process can make up to 60% of the whole process planning stage (Gregori et al., 2014; Kirschman & Jara-Almonte, 1992; Minetto et al., 2017). One of the factors which contributes to long computational time is usually caused by bad STL file which needs to be checked and corrected as discussed in

Section 2.3.2. Hence, the algorithm will have to include error-checking routine which can be complex and demands more computational time. Another factor is due to the nature of STL facets which are arbitrary and unorganized. This causes the algorithm to take longer time to find facet that intersects with the slicing plane. As discussed in Section 2.4.3, using more organized data structure which handles the facets data can improve the computational time for Contour Generation.

CHAPTER 3

METHODOLOGY

3.1 Introduction

In this chapter, the slicing, contour loop, and line to pixel map algorithms are thoroughly discussed and elaborated based on the fundamentals of the algorithm and the structure of the algorithm. This chapter also discusses the structure of STL formatted CAD models and how the data from this CAD models are read and stored in the concept of programming.

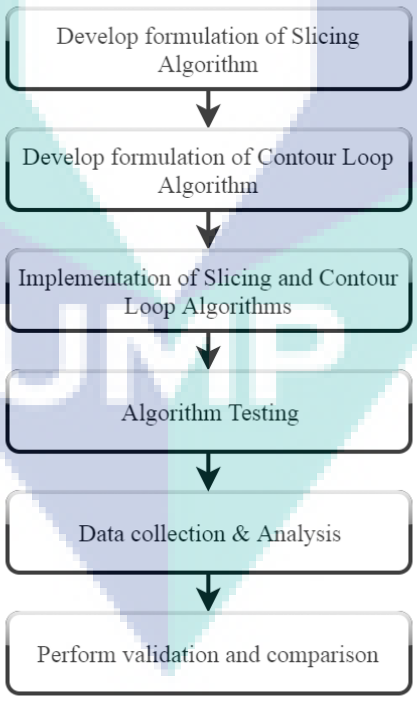

Develop formulation of Slicing Algorithm
Develop formulation of Contour Loop Algorithm
Implementation of Slicing and Contour Loop Algorithms
Algorithm Testing
Data collection & Analysis
Perform validation and comparison

Figure 3.1 Flowchart of Research Methodology

Figure 3.1 shows the methodology flows of this research starting with the development of Slicing algorithm. In this phase, the slicing algorithm is developed based on its fundamental equation, facet data handling, and facet-plane interaction cases handler. Next, in the Contour Loop algorithm development phase, the algorithm is constructed by implementing pixel line algorithm and head-to-tail search algorithm. This phase is followed by implementation for both Slicing and Contour Loop algorithms on actual STL CAD format. Then, the algorithm is tested with different complexity STL model. Computation time is measured and later tabulated. Finally, the results are validated and compared with the existing results obtain from literature.

3.2 STL Data Management

Managing a huge number of facets require proper encapsulation of the data. Hence, the proposed algorithm introduces a list of facet class to store the facet data. These data will be read by the slicing algorithm. Each facet class stores the vertices (P_1, P_2, P_3) and maximum/minimum Z coordinates between the three vertices. The use of maximum/minimum Z value is to filter out other facets except the ones intersecting with the slicing plane by comparison of $z_{min} \leq z_{slice} \leq z_{max}$ for each facet in the STL file. This is to reduce the number of facets from the list by taking only a portion of it and improves the performance of the slicing algorithm.

3.3 Slicing Algorithm

A slicer is an algorithm that slices each triangular facet in STL model which intersects with the slicing plane. The slicing process of each intersecting facet generates line segments which lie on the slicing plane. These line segments are arbitrary because all facets in STL model are also randomly ordered. Hence, the line segment requires a contour loop algorithm to connect each line segment into single or multiple closed loop contours which will be discussed in the next section. By adjusting the slicing plane height, different contour can be generated. This allows layer-by-layer contour generation for layered manufacturing process.

3.3.1 Case Handler for Facet-Plane Interaction

As discussed in Section 2.4.2, the cases of facet-plane intersection can be categorized into 5 cases (6 cases including Case IV in Figure 3.2). This section focuses on how each facet-plane interaction cases is handles in the proposed Contour Generation Algorithm. The most common issue contributing to slicing error is caused by interaction between facet and the slicing plane. A line segment requires only two distinct points. However, some cases of facet-plane interaction cause the slicing algorithm to generate more/less than two distinct points. Known cases of facet-plane interactions are defined in Figure 3.2 and Table 3.1.

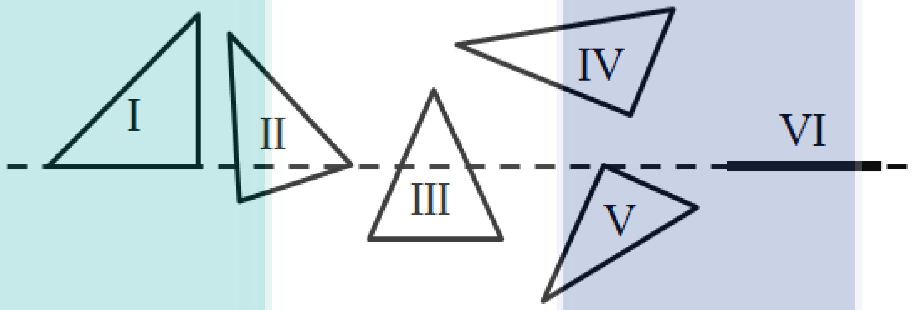


Figure 3.2 Possible facet-plane interaction

Table 3.1 Definition of interaction cases

Case	Interaction of facet and plane	Possible Point
I	Line through one side of the facet	4
II	Line bisecting the facet through one vertex	3
III	Line bisecting the facet through two sides	2
IV	No intersection	0
V	Vertex intersection	2
VI	Parallel intersection	6

Table 3.1 defines the number of points that is generated considering all six possibilities. As stated in Table 3.1, both Case I and VI have a side/sides which in parallel with the plane. This parallel intersection must be eliminated to avoid redundant points. Case VI is ignored because all the sides are in parallel. Case VI occurs at flat surfaces and usually found during slicing the base of the model. It can also be detected when $z_{\min} = z_{\max}$. For Case I, the parallel side is eliminated and the other two sides are sliced. The method of eliminating parallel side is using a dot product criterion which will be discussed later.

Next, Case II happens when the slicing plane intersects at one vertex of the facet and one side passing through the plane. Case II generates three intersection points which are redundancy for line generation. During the slicing routine, the vertex intersection generates two similar points and one distinct point. In the proposed slicing algorithm, the algorithm compares the cross combination between the three generated points to see which of the combinations give the longest line and later stores the combination as a line segment. Intersection at vertex can also be seen in Case V. A two similar point cannot forms a line segment. Hence, Case V is ignored. The same goes for Case IV which is already been filtered out using Z-comparison in the previous section.

3.3.2 Formulation of Slicing

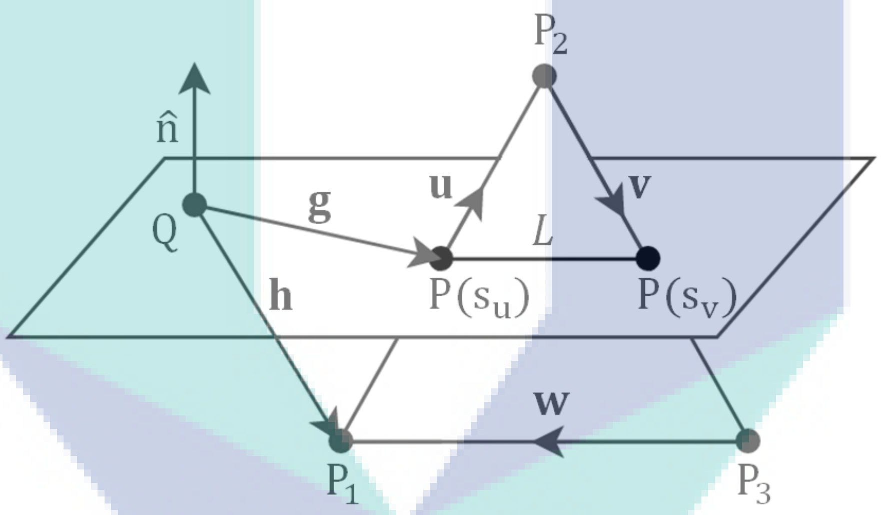


Figure 3.3 Facet-Plane intersection model

The fundamental of the proposed slicing algorithm is based on line-plane intersection mathematical equation which differs than the one discussed in Section 2.4.1. Consider one side of the facet as a line connecting two vertices from P_1 to P_2. In 3D environment, a line can be either parallel to a plane or intersects at one point on the plane (see Figure 3.3). Parametric equation of a line with P_o as the initial point and P_f as the final point is given as:

$$P(s) = P_o + s(P_f - P_o) \tag{3.1}$$

Assume that point Q in Figure 3.3 lies on the same slicing plane where its x and y coordinate can be randomly set (usually set at the origin), and z_{slice} is the slicing plane height. By adjusting the z_{slice} value, slicing algorithm can be implemented at any height of the STL model. Slicing plane normal is given by unit vector $\hat{n} = \langle 0, 0, 1 \rangle$ which is for the case of slicing with respect to Z-axis**.** Unit vector $\hat{n}$ can be change to alter the slicing plane direction. In Figure 3.3, the line $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are direction vectors connecting the three vertices (P_1, P_2, and P_3) in clockwise order (P_1, P_2, P_3, P_1) respectively to represent the sides of the facet. The algorithm initially checks for any intersection which exists between the facet sides and the slicing plane by computing the dot product criterion, $\hat{n} \cdot \mathbf{u} = 0$, $\hat{n} \cdot \mathbf{v} = 0$, $\hat{n} \cdot \mathbf{w} = 0$ respectively. These criterions eliminate the facet parallel sides for both Case I and Case VI as mentioned earlier. The output of the criterion becomes zero when there is no intersection between the direction vector and the slicing plane. Should the line intersect with the plane as seen in Figure 3.3, the criterion output is not equal to zero. As seen in Figure 3.3, the direction vector $\mathbf{u}$ from point P_1 to P_2 intersects with the plane at point $P(s_u)$. Substituting P_1 as the initial point P_o, P_2 as the final point P_f and s_u as the parameter s, Equation 3.1 now becomes:

$$P(s_u) = P_1 + s_u(P_2 - P_1) \tag{3.2}$$

Equation 3.2 above is a parametric equation of the intersection point $P(s_u)$**.** By using a direction vector $\mathbf{g}$ that lies on the same plane, it is known that $\hat{n} \cdot \mathbf{g} = 0$ because the vector is in parallel to the plane. The vector can also be represented as $\mathbf{g} = \mathbf{h} + s_u\mathbf{u}$. Hence, $\hat{n} \cdot \mathbf{g} = \hat{n} \cdot (\mathbf{h} + s_u\mathbf{u}) = 0$. Rearranging this equation, the parameter s_u can be written as:

$$s_u = \frac{-\hat{n} \cdot \mathbf{h}}{\hat{n} \cdot \mathbf{u}} \tag{3.3}$$

Vector $\mathbf{h}$ is given by $\mathbf{h} = P_1 - Q$ and vector $\mathbf{u}$ is given by $\mathbf{u} = P_2 - P_1$. Substituting both vector $\mathbf{h}$ and $\mathbf{u}$ into Equation 3.3 yield:

$$s_u = \frac{\hat{n} \cdot (Q - P_1)}{\hat{n} \cdot (P_2 - P_1)} \tag{3.4}$$

Now, the intersection point $P(s_u)$ can be computed. Based on the previous derivation, it is known that vector $\mathbf{g}$ is a direction vector from point Q to the intersection point. This means that $\mathbf{g} = \mathrm{P}(s) - \mathrm{Q}$. Vector $\mathbf{h}$ is a direction vector from point Q to the initial point of the vector $\mathbf{u}$ which is $\mathbf{h} = \mathrm{P_o} - \mathrm{Q}$. Hence, $\mathbf{g} = \mathbf{h} + \mathrm{s}\mathbf{u}$ where $\mathbf{u} = \mathrm{P_f} - \mathrm{P_o}$. Finally, the general form of the Equation 3.4 above can be derived as:

$$\mathrm{s} = \frac{\hat{\mathrm{n}} \cdot (\mathrm{Q} - \mathrm{P_o})}{\hat{\mathrm{n}} \cdot (\mathrm{P_f} - \mathrm{P_o})} \qquad 3.5$$

Hence, applying Equation 3.1 and 3.5 to another intersecting side of the facet yield another intersection point $\mathrm{P}(\mathrm{s_v})$ forming a line segment L on the slicing plane. However, the value $\mathrm{s_u}$ and $\mathrm{s_v}$ must be within the range $0 \leq \mathrm{s} \leq 1$ to ensure that the intersection points exists only within the line between $\mathrm{P_o}$ and $\mathrm{P_f}$.

3.3.3 Line to Pixel Mapping

The generated line segments are floating numbers, which is computationally expensive, and tends to cause truncation errors that disrupt the performance of the contour generation algorithm. In this context, the floating numbers are irrelevant because the resolutions of the geometry are eventually subjected to the projection device resolution. The algorithm proposed novel method of reducing computational time by converting the line segments floating number coordinates to pixel coordinates.

The Pixel Mapping method starts by computing both aspect ratio of the projection device $\mathrm{AR_{device}}$ and the aspect ratio of the object $\mathrm{AR_{object}}$. This allows the algorithm to detect whether to geometry can be fit to neither height nor width of the projection device while preserving the aspect ratio of the geometry. The aspect ratios are given by:

$$\mathrm{AR_{device}} = \frac{\mathrm{width}}{\mathrm{height}} \qquad 3.6$$

$$\mathrm{AR_{object}} = \frac{\mathrm{x_{max}} - \mathrm{x_{min}}}{\mathrm{y_{max}} - \mathrm{y_{min}}} \qquad 3.7$$

In Equation 3.6 and 3.7, these values are used as comparison to determine whether the object should be fit to width or height. If the $\mathrm{AR_{object}}$ has higher proportion than $\mathrm{AR_{device}}$, it means that the object has longer width and must be fit to width and vice versa when the $\mathrm{AR_{object}}$ is less than $\mathrm{AR_{device}}$. Using this condition, a new variable R is introduced to represent the conditions as:

$$\mathrm{R} = \begin{cases} \mathrm{width} - 1;\ \mathrm{AR_{object}} \geq \mathrm{AR_{device}} \\ \mathrm{height} - 1;\ \mathrm{AR_{object}} < \mathrm{AR_{device}} \end{cases} \qquad 3.8$$

The R value is minus by one because the pixel coordinates are zero based integer and its value is conditional depending on the comparison of $\mathrm{AR_{object}}$ with $\mathrm{AR_{device}}$. Consider a case of fit-to-width $\mathrm{AR_{object}} \geq \mathrm{AR_{device}}$; the following equation can be used to map the floating number of x-coordinate into a horizontal pixel position (the width) of the projection device:

$$\mathrm{x_{pixel}} = \frac{\mathrm{P_x} - \mathrm{x_{min}}}{\mathrm{x_{max}} - \mathrm{x_{min}}} \cdot \mathrm{R} \qquad 3.9$$

However, y-coordinate must be scaled with modified value of $\mathrm{height'}$ to retain the original aspect ratio of the object. This means, the $\mathrm{AR_{device}}$ must equal to $\mathrm{AR_{object}}$. Hence, the new $\mathrm{AR_{device}}$ value is defined as:

$$\mathrm{AR'_{device}} = \mathrm{AR_{object}} = \frac{\mathrm{width'}}{\mathrm{height'}} \qquad 3.10$$

For this case, the $\mathrm{width'} = \mathrm{R}$ where $\mathrm{R} = \mathrm{width} - 1$ because it is a fit-to-width condition. Hence, using Equation 3.10 the modified height is written as:

$$\mathrm{height'} = \frac{\mathrm{width'}}{\mathrm{AR_{object}}} = \frac{\mathrm{R}}{\mathrm{AR_{object}}} \qquad 3.11$$

The equation of y-coordinate now can be derived as:

$$\mathrm{y_{pixel}} = \frac{\mathrm{P_y} - \mathrm{y_{min}}}{\mathrm{y_{max}} - \mathrm{y_{min}}} \cdot \frac{\mathrm{R}}{\mathrm{AR_{object}}} \qquad 3.12$$

Based on Equation 3.12, the y_{pixel} is now scaled to modified value of height for perseverance of its original aspect ratio and it represents the vertical pixel position of the projection device.

Next, for the case of fit-to-height where $AR_{object} < AR_{device}$ and the variable $R = height - 1$, the algorithm must use full scale of projection device height for y-axis and a modification of the width for the x-axis. The equation for y_{pixel} can be written as:

$$y_{pixel} = \frac{P_y - y_{min}}{y_{max} - y_{min}} \cdot R \qquad 3.13$$

But the x_{pixel} must be scaled with modified value by using Equation 3.10 which now becomes:

$$width' = height' \cdot AR_{object} = R \cdot AR_{object} \qquad 3.14$$

The equation of x_{pixel} is now derived as:

$$x_{pixel} = \frac{P_x - x_{min}}{x_{max} - x_{min}} \cdot R \cdot AR_{object} \qquad 3.15$$

Comparing the Equation 3.9 and 3.15, and Equation 3.12 and 3.13, two new conditional variables are introduced to generalize both equations for x_{pixel} and y_{pixel} which are:

$$V = \begin{cases} 1 & ;\ AR_{object} \geq AR_{device} \\ AR_{object} & ;\ AR_{object} < AR_{device} \end{cases} \qquad 3.16$$

$$W = \begin{cases} AR_{object}^{-1} & ;\ AR_{object} \geq AR_{device} \\ 1 & ;\ AR_{object} < AR_{device} \end{cases} \qquad 3.17$$

Finally, the equations for x_{pixel} and y_{pixel} are rewritten as:

$$x_{pixel} = \frac{P_x - x_{min}}{x_{max} - x_{min}} \cdot R \cdot V \qquad 3.18$$

$$y_{pixel} = \frac{P_y - y_{min}}{y_{max} - y_{min}} \cdot R \cdot W \qquad 3.19$$

In the algorithm, Equation 3.18 and 3.19 map the floating number coordinates of the line segment L for both P_o and P_f to a new pixel line segment coordinates. These pixel coordinates are store as an unsigned integer value to eliminate the decimal point of its original value so that it can be used for the pixel mapping of the projection device.

3.3.4 Algorithm Structure

This section discusses complete implementation of the slicing algorithm. All the fundamentals of the slicing algorithm have been previously explained starting from issues regarding STL models, the fundamentals of slicing, and the line to pixel mapping fundamental. Figure below shows the structure of the algorithm which will be thoroughly discussed in this section.

```
Algorithm 1: Slicing
1:  $L[0...n] \subseteq (Facet[0...j] \cap z_{slice})$          ▷ get intersecting facet
2:  for each $facet \in L$ do
3:      initialize $k, line, P_{slice}[3]$                    ▷ initialize buffers
4:      for each $side \in facet$ do
5:          if $(\hat{n} \cdot side \neq 0)$ then             ▷ dot product
6:              $P_{slice}[k] \leftarrow$ slice$(side, z_{slice})$
7:          else
8:              $P_{slice}[k] \leftarrow \emptyset$
9:          end if
10:         $k \leftarrow k + 1$
11:     end for
12:     $line \leftarrow$ caseHandler$(P_{slice})$
13:     $S[0...m] \leftarrow$ pixelMap$(line)$
14: end for
```

In Step 1, the algorithm first starts by obtaining a list of intersecting facets L from a list of STL Facet Class (as in Section 3.2) where n is the last index in the intersecting facets list L and j is the last index of the STL facets. This procedure filters out other non-intersecting facets to optimize the operation time based on the current slicing height, z_{slice}. As mentioned in Section 3.3.2, the process works by comparing each facet by $z_{min} \leq z_{slice} \leq z_{max}$. If the condition is true, then the list stores the intersecting facet in list L and vice versa. Starting from Step 2 until 14, the algorithm loops for each intersecting facet in the list L. Step 3 initializes working buffers to be used for the next operations. At Step 4, the algorithm performs another loop for each side of a facet since a facet contains 3 sides (u, v, w). Next, the algorithm determines whether the sides intersect with the slicing plane by performing dot product criterion mentioned in Section 3.3.2. If any side intersects, Step 6 is executed. In Step 6, the function $slice(side, \mathrm{z}_{\mathrm{slice}})$ slices the intersecting side of the facet and stores the result in buffer $\mathrm{P}_{\mathrm{slice}}[k]$. The size of the point buffer $\mathrm{P}_{\mathrm{slice}}$ is 3 to represent each sliced point for each side. Result of the slicing operation mainly consists of a line segment made of two points with *double*-precision floating number coordinates of P_{o} and P_{f}. The algorithm executes Step 8 if the side does not intersect and stores *null* in $\mathrm{P}_{\mathrm{slice}}[k]$. Then, Step 10 increments the index k means that the index 0, 1, and 2, represent u, v, and w respectively. Step 4 until Step 11 loops until each side of the facet are checked. The algorithm continues the process by executing Step 12 which handles the errors defined in Table 3.1 and stores the corrected line segment in $line$ variable. Step 13 converts the double-precision coordinates (P_{o} and P_{f}) into pixel (unsigned integer) coordinates and stores it into a list of pixel line segments S that will be used in the contour loop algorithm.

3.4 Contour Loop Algorithm

The contour loop algorithm basically is a head-to-tail search algorithm which connects a set of line segments that belongs to the same contour loop. By assigning the first line segment P_{o} as the initial tail ($\mathrm{P}_{\mathrm{init}}$) and P_{f} as the head ($\mathrm{P}_{\mathrm{find}}$), the head will begin to search for next tail which has the same coordinate but in another line segment. When found, the search algorithm assigns the found line segment as the new head of the search algorithm. This process repeats until the head meets with the first initial tail $\mathrm{P}_{\mathrm{init}}$ that indicates a closed loop is formed. The contour loop algorithm is shown in the algorithm table below.

Algorithm 2: Contour Loop

```
1:  initialize $id \leftarrow 0$, $P_{init} \leftarrow S[0].P_o$
2:  for $i = 0$ to $S.length - 1$ do
3:      $P_{find} \leftarrow S[i].P_f$
4:      if Compare($P_{find}, P_{init}$) then
5:          $id = id + 1$
6:          $P_{init} \leftarrow S[i+1].P_o$
7:      else
8:          $found \leftarrow$ Find($i+1, P_{find}$)
9:          if $found \neq -1$ then
10:             $isInvert \leftarrow$ Compare($P_{find}, S[found].P_f$)
11:             Swap($S[found], S[i+1], isInvert$)
12:         end if
13:     end if
14:     $S[i+1].id \leftarrow id$
15: end for
```

The arbitrary pixel line segments obtained by the slicing algorithm are put into a list which is first sorted by P_o. Y value then P_o. X value. The algorithm then, initializes an unsigned integer variable to identify and isolate each contour loops. Next, the algorithm assigned P_o of the first line segment in the list as initial point P_{init} of the first closed contour loop and its P_f as the search point to locate next neighbouring line segment from the list. In Step 5, the algorithm checks whether a closed loop is found else the algorithm proceeds to find next neighbouring line and hold its position in the list into an unsigned integer variable $found$. The function $Find$ searches for neighbouring line from the list with an offset index starting from next line segment $(i + 1)$ of the iteration until the end of the list. It uses P_{find} as the searching point which can be equal to next line P_o or P_f. This is because all the lines sliced during the slicing algorithm are arbitrary and it is difficult to know whether the point P_o to P_f is in the same direction with the contour loop. When next line is found, the function returns an unsigned integer index of the found line as shown in Step 9. If the line is not found, then the function returns -1. Step 11 checks the inversion of the found line. Should the line inverts, then P_{find} must be equal to the P_f of the found line. Vice versa, the line is in the right orientation. Next in Step 12, the function $Swap$ is to swap the element in the list between the found line segment and the next line of the iteration $(i + 1)$. If the line is inverted as previously checked in Step 11, the function $Swap$ will also flip the found point such as $P_f = P_o$ and $P_o = P_f$ before swapping the two lines. Step 15 is mainly to assign the contour identity to the next line segment in the list since by this point; the next line segment has become a neighbouring line which was previously found in Step 9. During next iteration, new $\mathrm{P_{find}}$ will be assigned as the search parameter and the whole process will be repeated unless the comparison between $\mathrm{P_{find}}$ and $\mathrm{P_{init}}$ is equal to one another. This indicates a closed loop is found and it is necessary to increment the loop identity variable and assign new $\mathrm{P_{init}}$.

3.5 Computational Time Measurement

The proposed Contour Generation Algorithm is implemented in C++11 programming language. The code is written and executed in Qt Creator 4.9.0 (Open-Source) software and compiled with MinGW.

In order to evaluate the performance of the proposed algorithm, computational time is collectively measured and plotted in milliseconds. Qt Creator provides real-time measurement library which is based on system clock with resolution of nanoseconds. It can be achieved by using <QElapsedTimer> object within Qt Creator. An instance *qTimer* of the <QElapsedTimer> needs to be created to start the measurement. The timer is inserted in before the function call for slicing algorithm. Using *start()* initiates the timer. After the function call, the measurement result is obtained by accessing the <QElapsedTimer> class member *nsecsElapsed()* which returns the int64 value of elapsed time since *restart()* is called. The value must be divided by one million to convert to milliseconds. For continuous measurement, the *qTimer* must be reset by calling *restart()* again to reset the timer counter back to zero. Pseudocode below shows the implementation for measuring both slicing and contour algorithm.

```
#include <QElapsedTimer>
..
QElapsedTimer* qTimer;
double SliceResult, ContourResult;

qTimer->start();

while(Slicing){
   qTimer->restart();
   Slice(height);
   SliceResult   = qTimer->nsecsElapsed() / 1e6;
   ContourLoop();
   ContourResult = qTimer->nsecsElapsed() / 1e6 – SliceResult;
   height += delta_height;
}
```

As shown in the pseudocode, the variables use for storing the results is *double* type which is 64-bit precision variable. Summation of both *SliceResult* and *ContourResult* give total computational time to generate the contour layer. These contour results are exported into a comma-separated value (CSV) file format during each iteration of slicing height until the maximum slicing height. Each CSV file contains contour line segments dataset (generated in Algorithm 2 in Section 3.4) according to respective slicing height. Other parameters such as number of intersecting facets at respective slicing height and STL model facet count is also included in the CSV result.

The CSV formatted results are loaded into MATLAB 2016b using the MATLAB Script (attached in Appendix D) to plot the histogram of slicing time, contour loop time, total computational time, number of intersecting facets, and loop count. The MATLAB Script is also programmed to reconstruct a 3D figure based on generated line segments coordinates from the CSV files.

3.6 Test Environment

Proposed algorithm is programmed in VB.NET programming language. The algorithms are implemented on Intel i3-2350M CPU 2.30 GHz with 6 GB RAM workstation. For the validation process, the algorithms are re-written in C++11 and tested on Intel i7-6700 3.40 GHz workstation with 4 GB RAM based on specification mentioned in the referenced journal.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Introduction

Reduction of computational time in Contour Generation Algorithm is important in improving the performance contour layer generation. It is because the process can take up to 60% of the entire process planning time (Gregori et al., 2014; Kirschman & Jara-Almonte, 1992). In this chapter, the performance of both slicing and contour loop algorithm are evaluated. The evaluation of the performance is based on the computational time required to complete each layer respective to their slicing height. Evaluation of the results are based on several STL model with different complexity which were used in the experiment. Each of these geometries/models are thoroughly analyzed based on the time performances of each algorithms including slicing algorithm and contour loop algorithm.

4.2 Sliced Model Output

In this section, the 3D sliced models are presented with colour-map indicating the total computational time at each respective layer slicing height. A sphere model is used to prove that the XZ and YZ planes are in correct ratio with respect to XY plane. Figure 4.1 below shows the results of Sphere slicing.

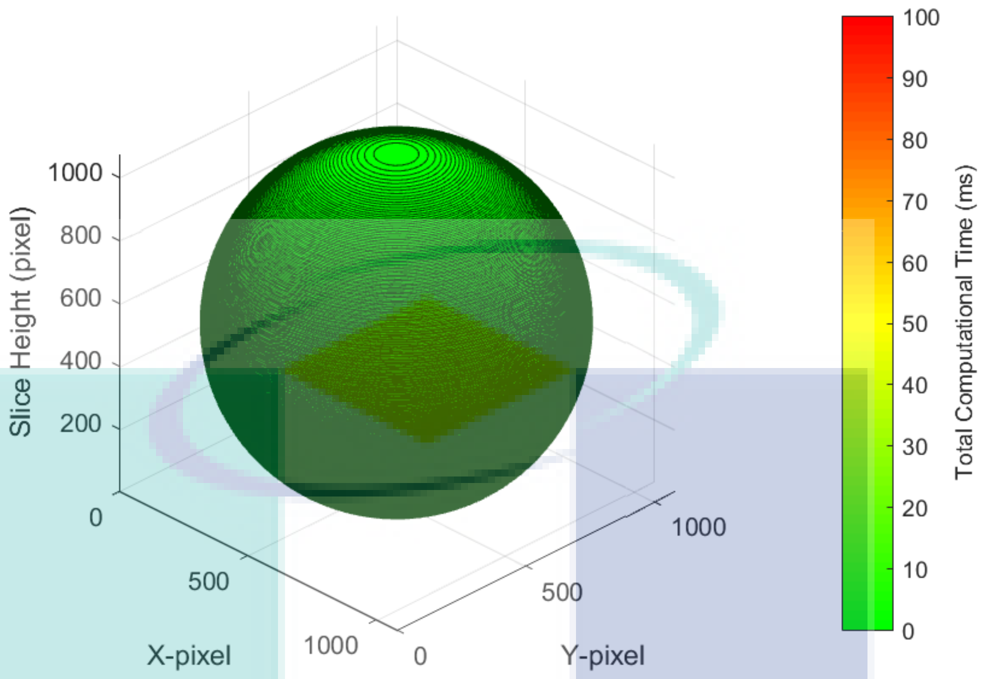


Figure 4.1 Sliced model (Sphere) with colour mapped total computational time

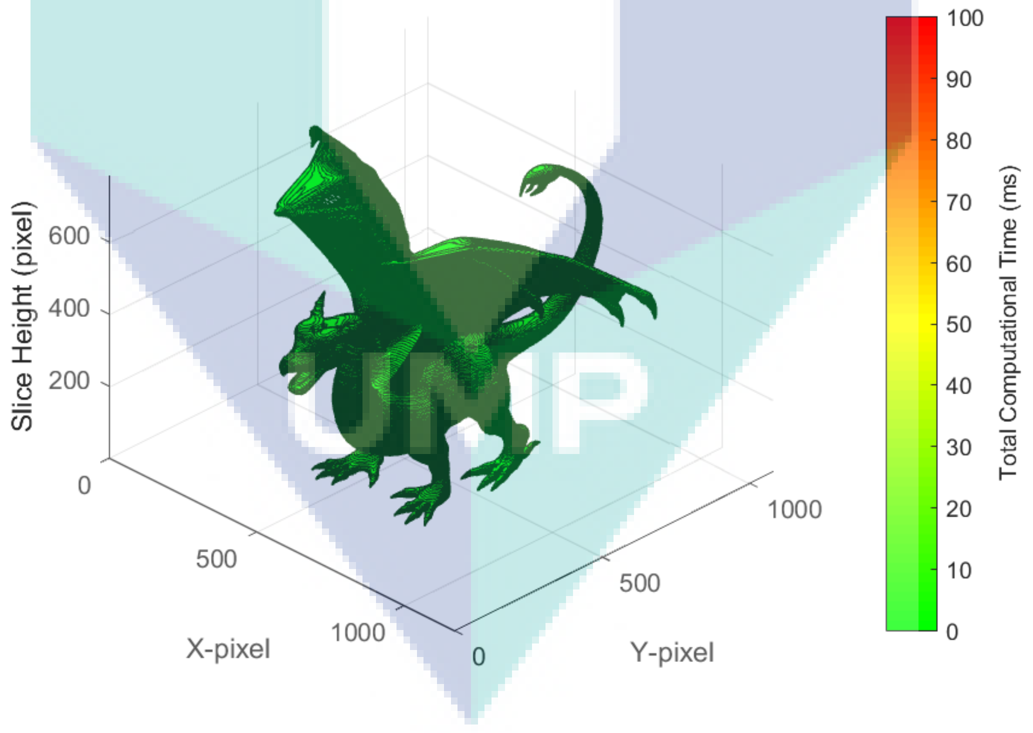


Figure 4.2 Sliced model (Dragon) with colour mapped total computational time

Figure 4.1 shows perfect formation of a Sphere model. Based on the measurement done in MATLAB, the dimension of the Sphere result is 1079 x 1079 x 1079 cubic pixel. This proves that the slicing height is in correct ratio with respect to XY plane. The Dragon

(Figure 4.2) model is properly formed when compared to its original STL file because there are no abnormalities in the layer result. For example, if abnormality occurs during algorithm execution, the layer formation will be disrupted and making the respective layer to show clear malformation.

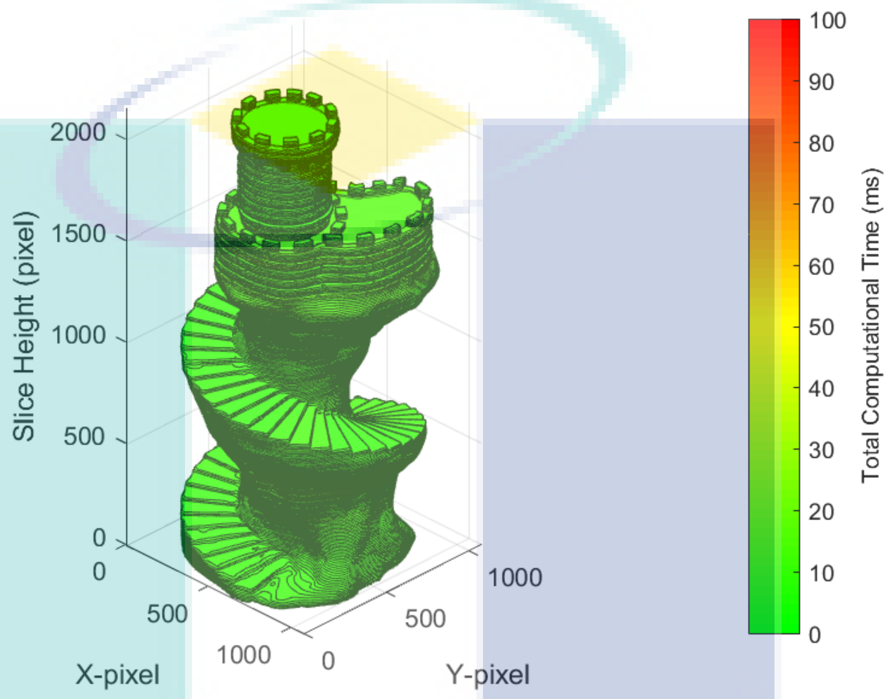


Figure 4.3 Sliced model (Tower) with colour mapped total computational time

Figure 4.2 and Figure 4.3 above show result of the algorithm implementation. Other STL model case studies are included in the Appendix A section. As seen above, the colour indicates the total computational time in milliseconds required to generate each contour to show the feasibility of implementing this algorithm to an actual DLP 3D printer. The lines that appear on the surface of the model are sliced contour lines generated by the algorithms. Colour differences at certain slicing height are due to the complexity of the geometry that differed at each height. This often demands more computational loads to generate the contours. Hence, longer computational time. The computational time at each respective height will be further discussed in the next section.

In previous discussion in Chapter 3, the algorithms are designed to work by referring to the current build plate height of the DLP 3D printer. In other words, the algorithms generate the contour instantly upon receiving the slicing height input value. This method is called instant slicing. The contour generation algorithm consists of two different algorithms: slicing and contour loop, the performance of each algorithms is evaluated in the next sections.

4.3 Slicing Algorithm Performance

In Chapter 3, the slicing algorithm is presented and discussed in detail. In brief, the slicing algorithm works by filtering out other facets which do not intersect with the slicing plane and then slice each of the intersecting facets to form each respective line segments. By algorithms complexity analysis, the worst case for the instant slicing algorithm can be represented as $O(n)$. Thus, it is expected that increasing number of elements will linearly increase the execution time. The performance graph below shows the result of slicing algorithm at each respective slice height. Total facet number of the STL model, the mean average, and the standard deviation (SD) are presented at the top of the graph. Table 4.1 shows the result of execution time measurement for the slicing algorithm at each slicing height.

Table 4.1 Time measurement for slicing algorithm

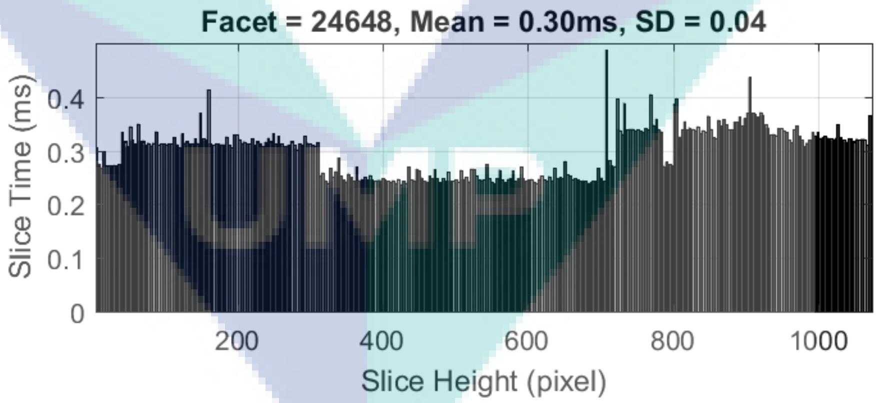
Model	Performance Graph
Sphere	

Table 4.1 Continued

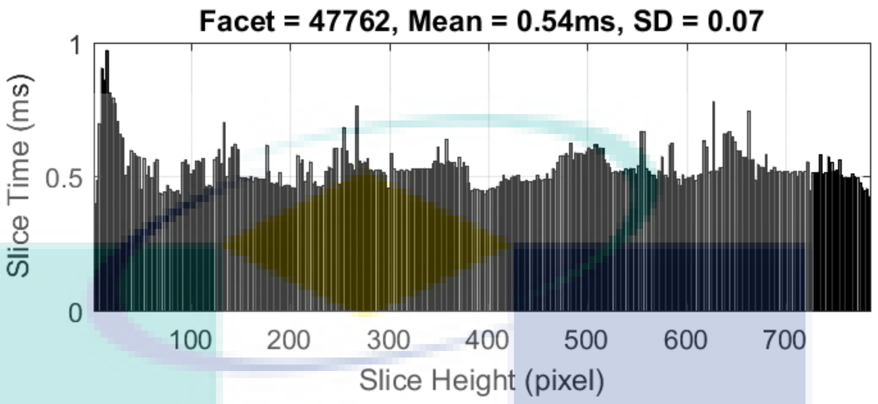
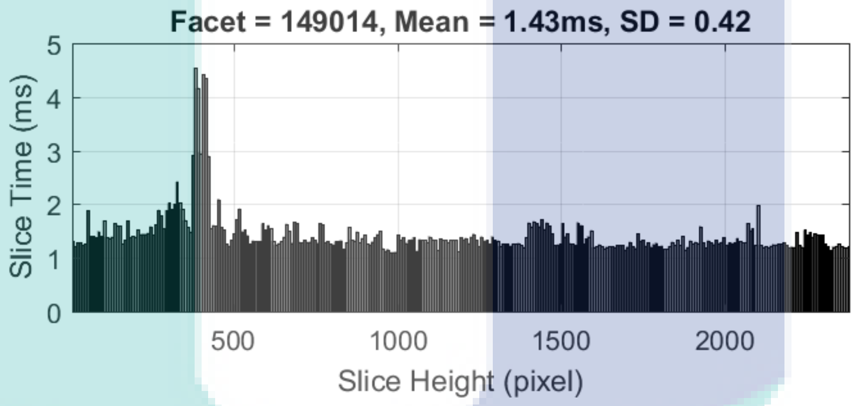
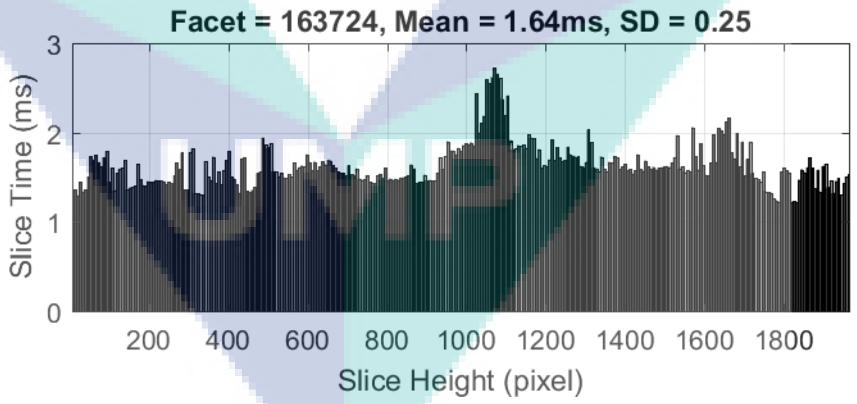
Model	Performance Graph
Dragon	
Eiffel Tower	
Gundam	

Table 4.1 Continued

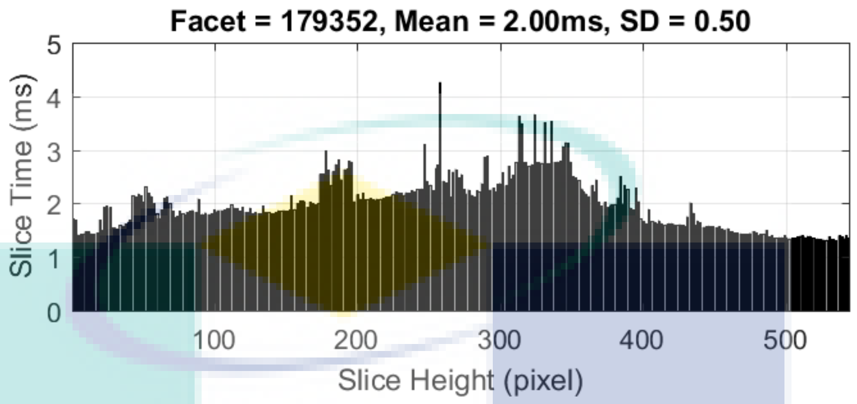
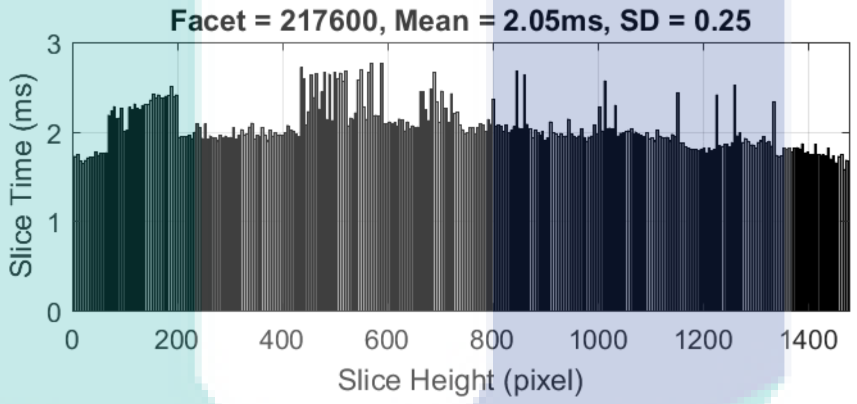
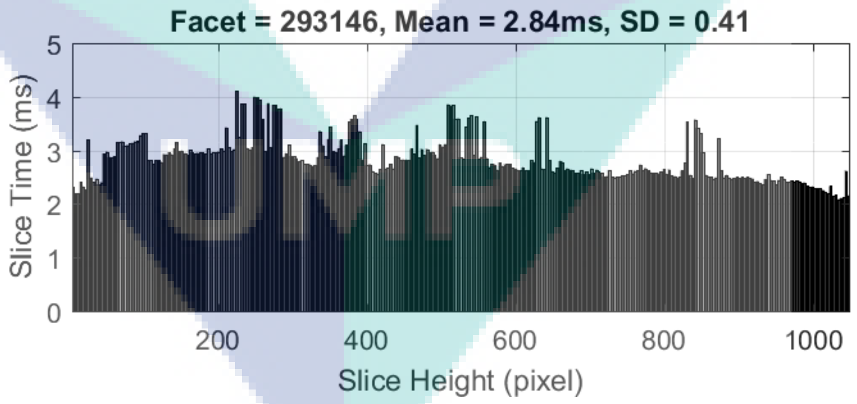
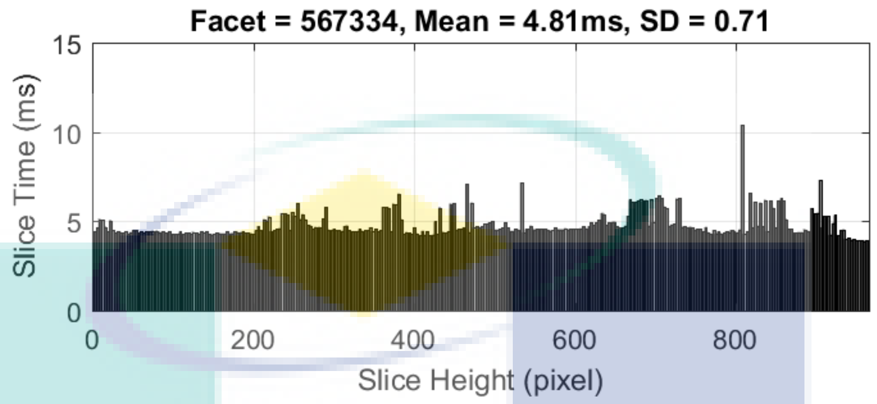
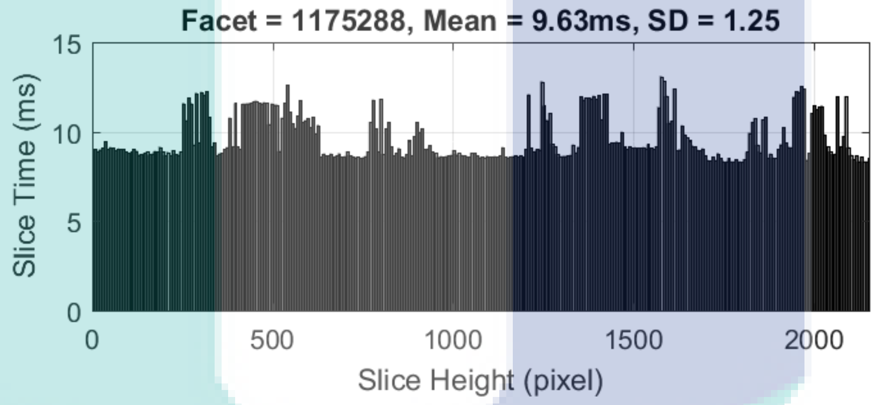
Model	Performance Graph
Speedster	
Heart	
Dreadnaught	

Table 4.1 Continued

Model	Performance Graph
Worm	
Spiral Tower	

By analyzing the pattern of each graph, all the graph above shows consistent slicing time regardless the slicing height when comparing the value of standard deviation (SD). There are some spikes caused by the operating system background processes which occupied the processor at the time. Each model above is sorted in ascending order of their total facet number (low polygon model to high polygon model). It is noticeable that increasing total facet number also increases the mean slicing time average.

4.4 Contour Loop Performance

The contour loop algorithm works by connecting all the arbitrary line segments generated by the slicing algorithm into one or more contour loops. Time measurement for the execution time is taken in milliseconds to measure how fast the algorithm is executed.

The results obtained are shown in Table 4.2 below which represents contour loop computational time at each slicing height for each STL model.

Table 4.2 Time measurement for contour loop algorithm

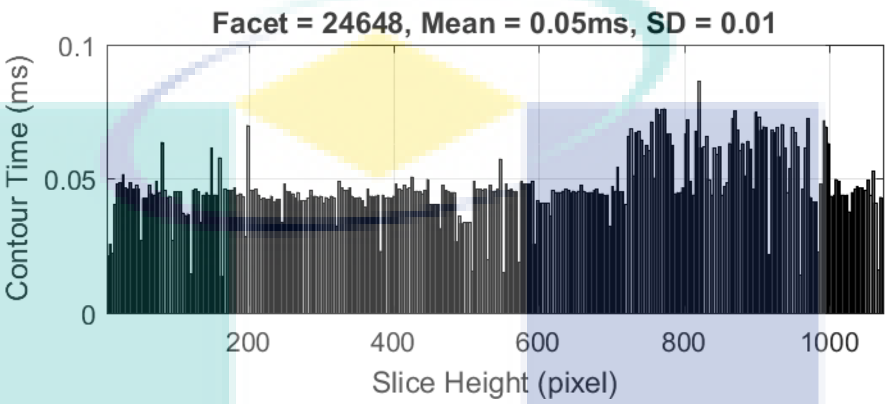
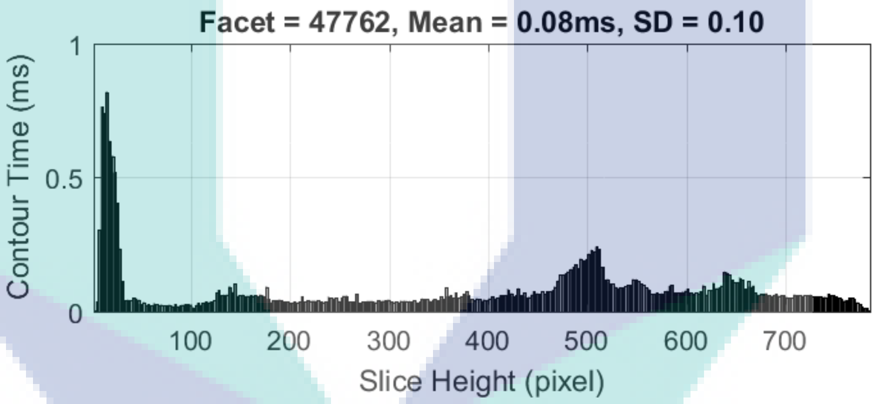
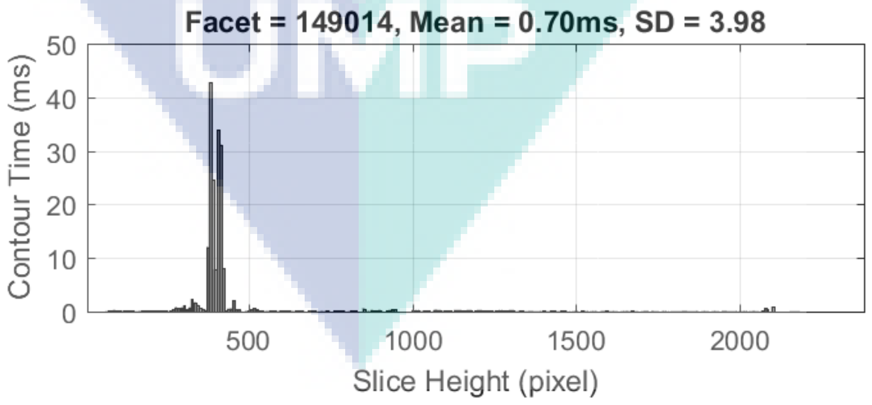
Model	Performance Graph
Sphere	
Dragon	
Eiffel Tower	

Table 4.2 Continued

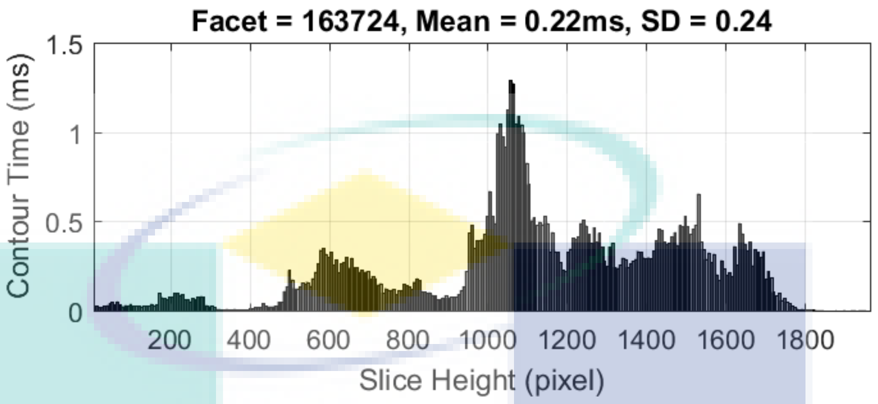
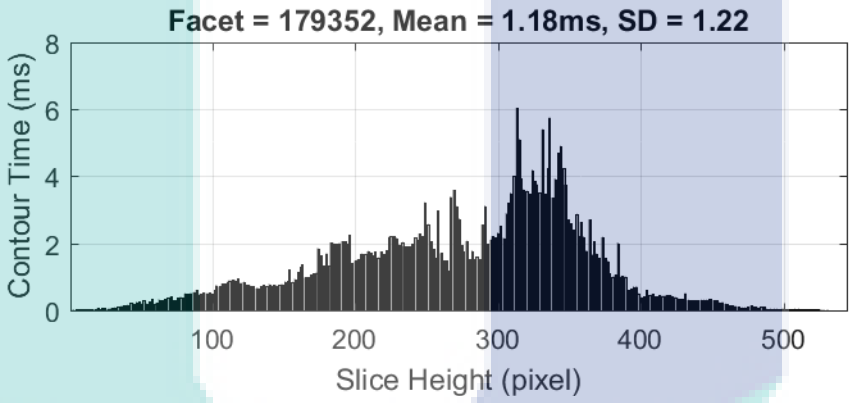
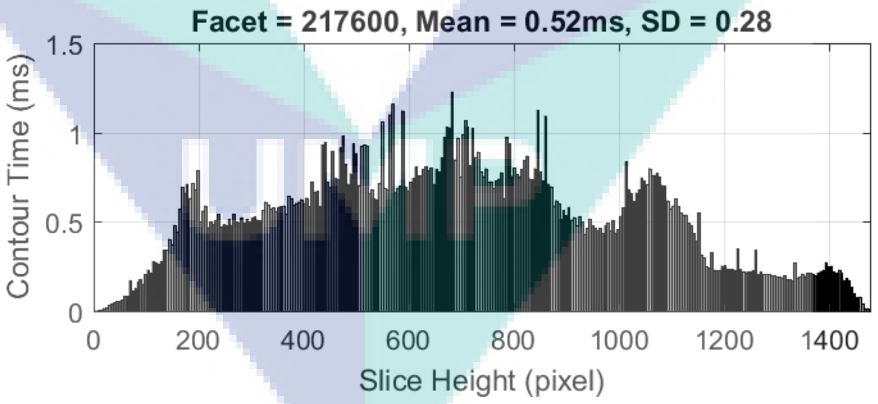
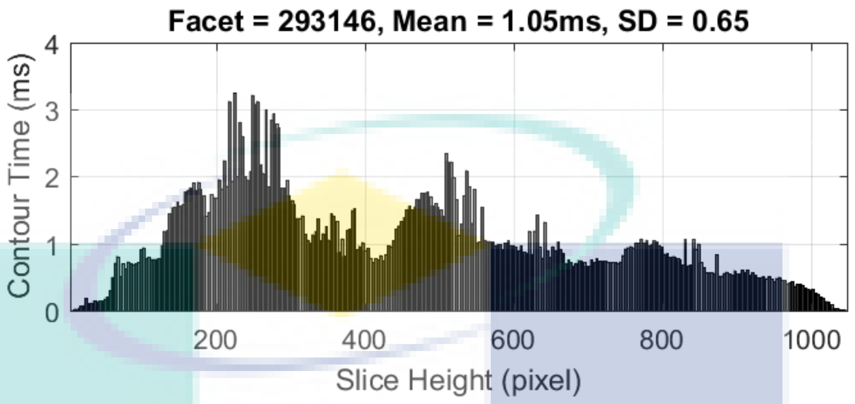
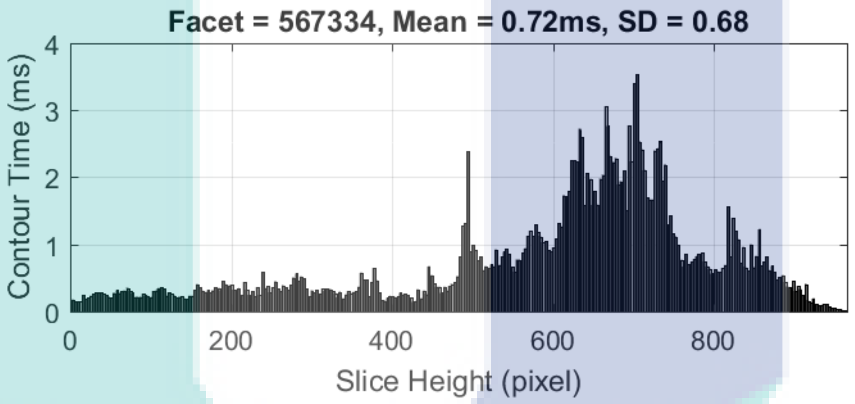
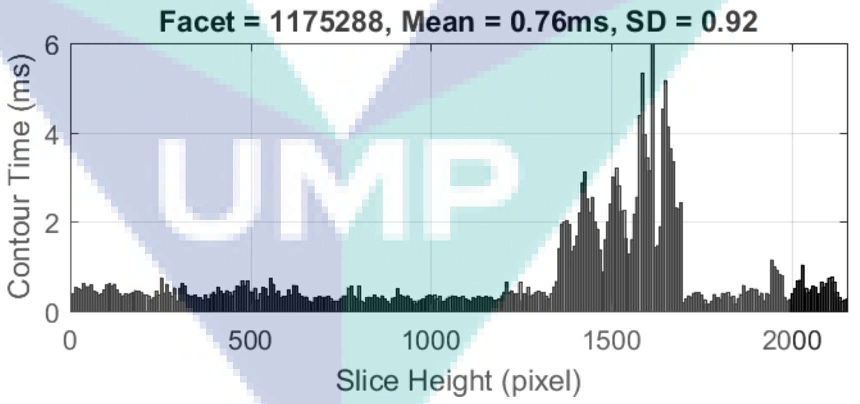
Model	Performance Graph
Gundam	
Speedster	
Heart	

Table 4.2 Continued

Model	Performance Graph
Dreadnaught	
Worm	
Spiral Tower	

As seen in the Table 4.2 above, the contour time results show unique pattern for each model. The Sphere model has consistent contour loop execution time. But, for the other models, there are inconsistencies in the contour loop time at certain slicing height. This is due to complex features of the models at certain height. The complexity of the model can be represented as the number of intersecting facets at each slicing height. More complex layer will have more facets number. Thus, it requires more computational time due to high intersecting facet counts.

4.4.1 Number of Intersecting Facet at Different Slicing Height

Table 4.3 below show the number of intersecting facets which intersect with the slicing plane at different heights. Unique feature of the Sphere model can be seen in the results below. It is found that the Sphere model has the same number of intersecting facets regardless the slicing height. This relates to the consistencies in its contour loop algorithm results in previous section. The number of intersecting facet pattern shows by the Dragon model also closely resembles the pattern in its contour loop time. These similarities can also be observed in other STL models.

Table 4.3 Number of intersecting facet at each slicing height

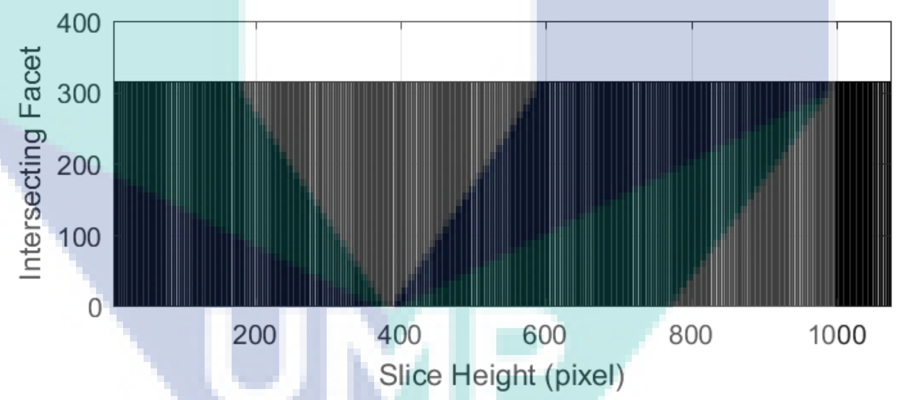
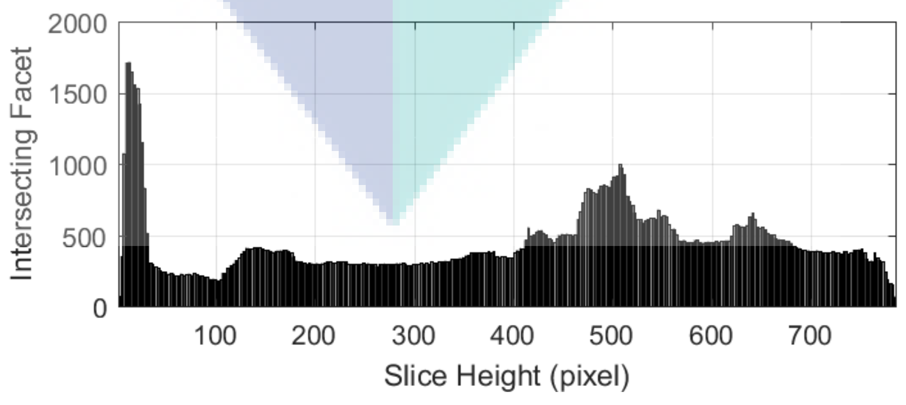
Model	Performance Graph
Sphere	
Dragon	

Table 4.3 Continued

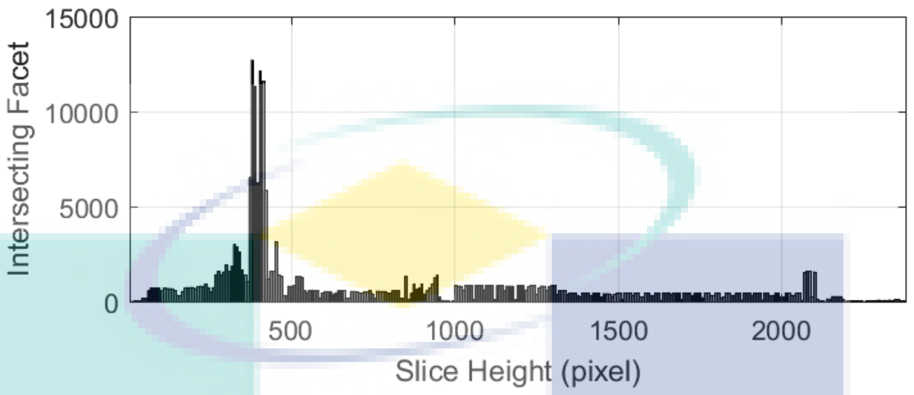
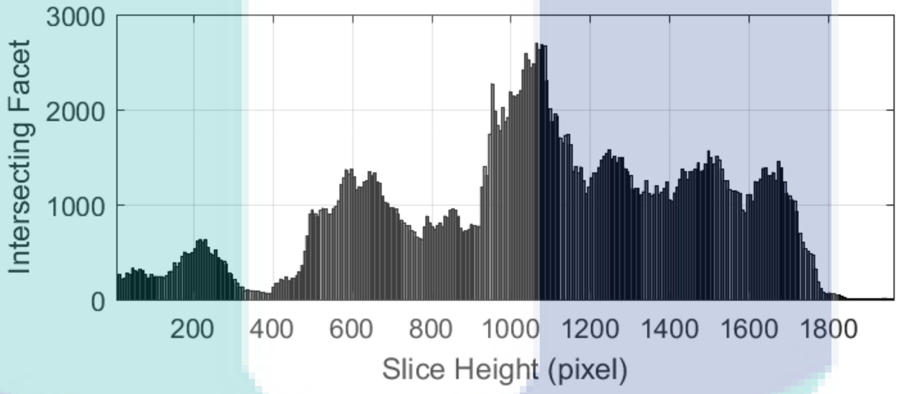
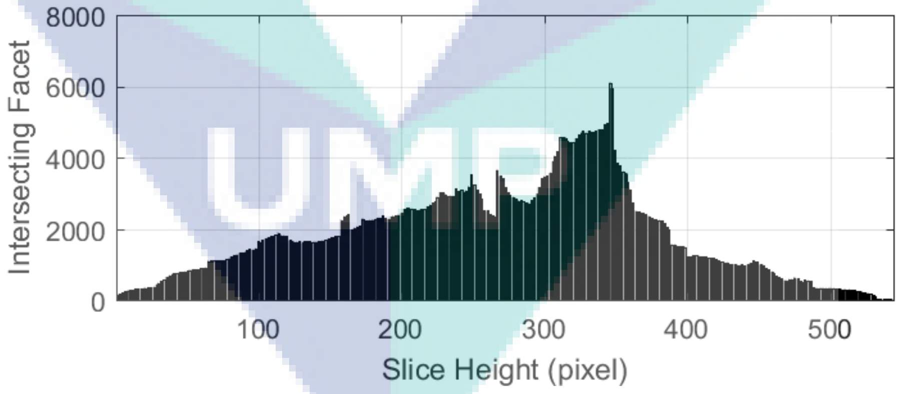
Model	**Performance Graph**
Eiffel Tower	
Gundam	
Speedster	

Table 4.3 Continued

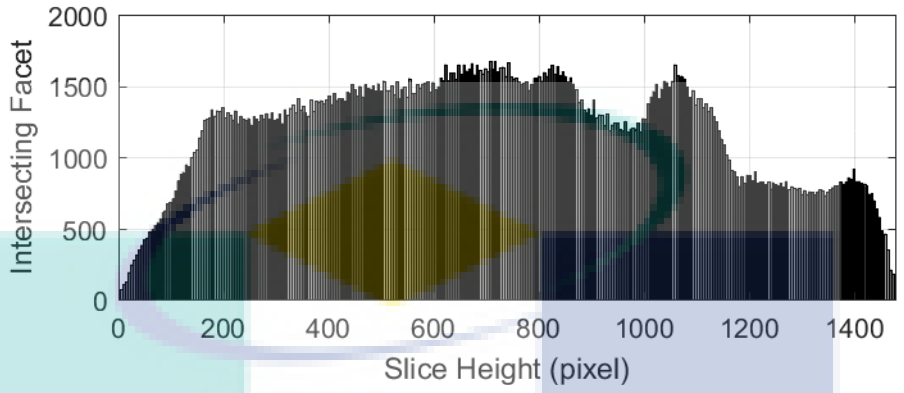
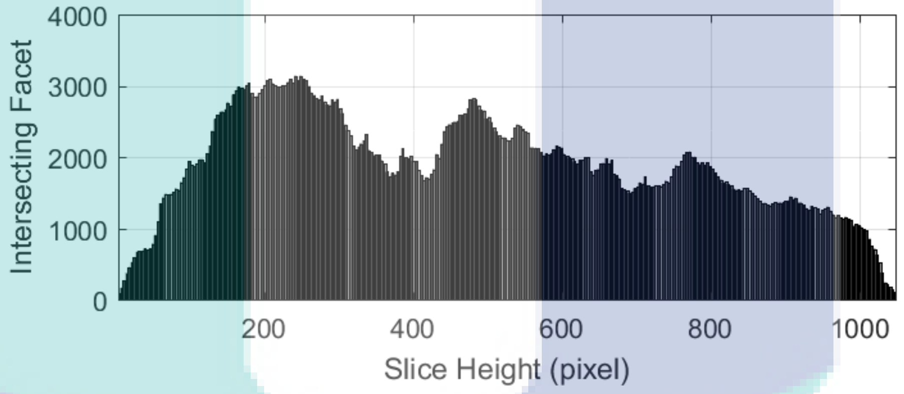
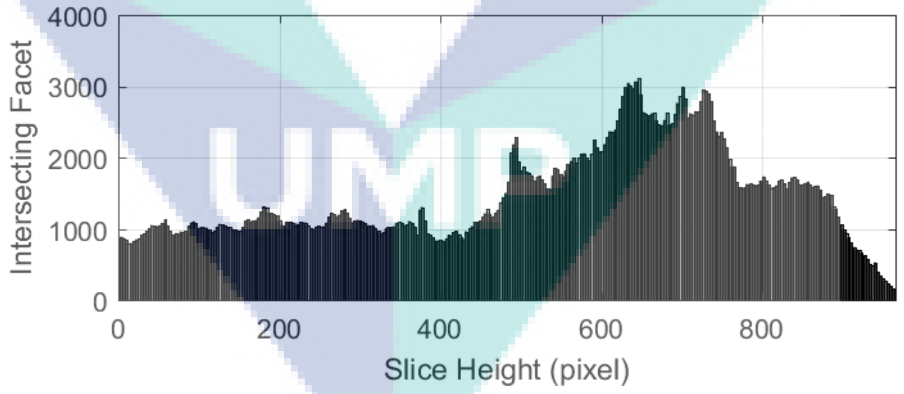
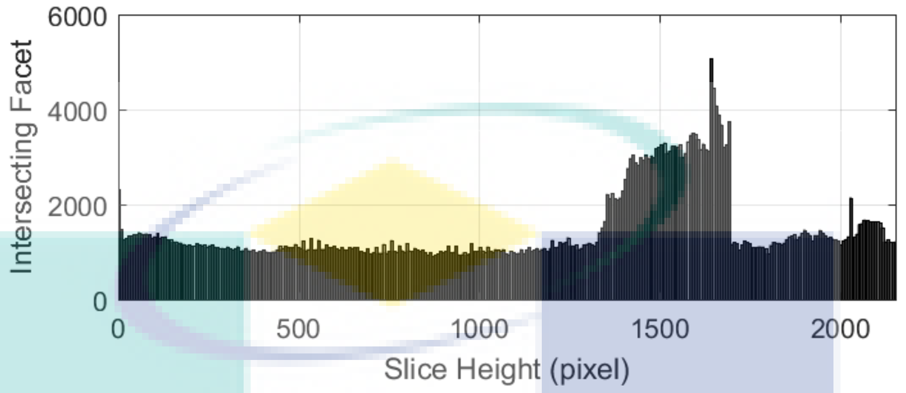
Model	Performance Graph
Heart	
Dreadnaught	
Worm	

Table 4.3 Continued

Model	Performance Graph
Spiral Tower	

4.4.2 Contour Loop Counts

After running the test, it is found that at each slicing height, there are different numbers of contour loops can be observed. Based on the proposed contour loop algorithm in Chapter 3, the contour loop counts are programmed to re-iterate to connect another closed loop contour. These re-iterations depend on the contour loop counts. As a result, this process requires more computational time compares to a single closed loop contour. The measurement of the contour loop counts at each respective slicing height are tabulated in Table 4.4 below.

Table 4.4 Number of loop counts at each slicing height

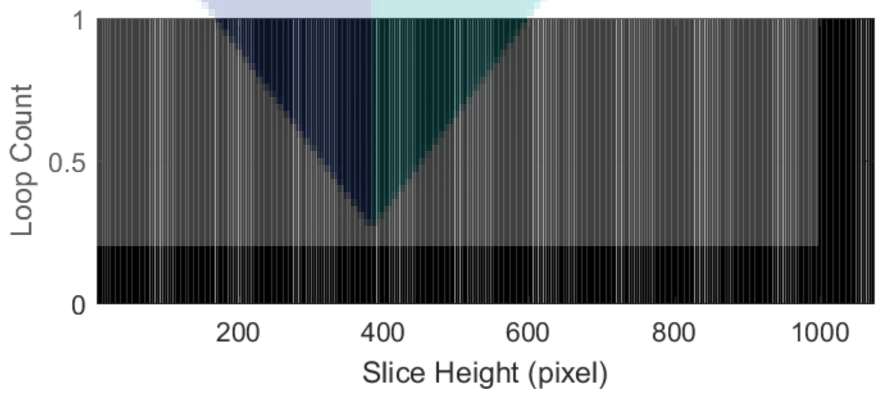
Model	Performance Graph
Sphere	

Table 4.4 Continued

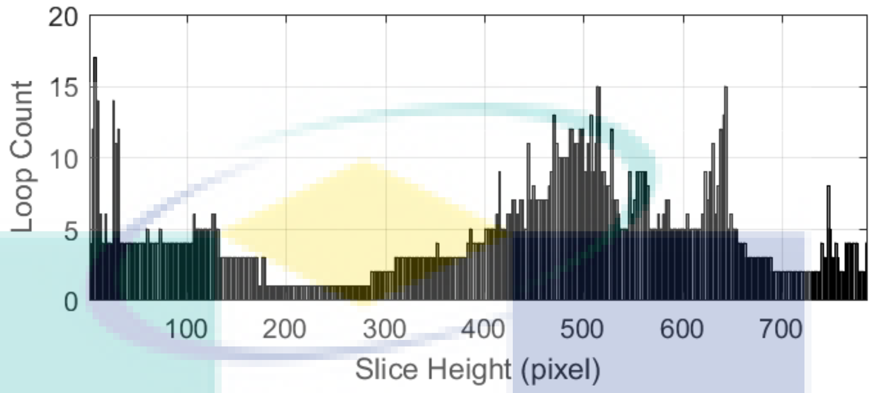
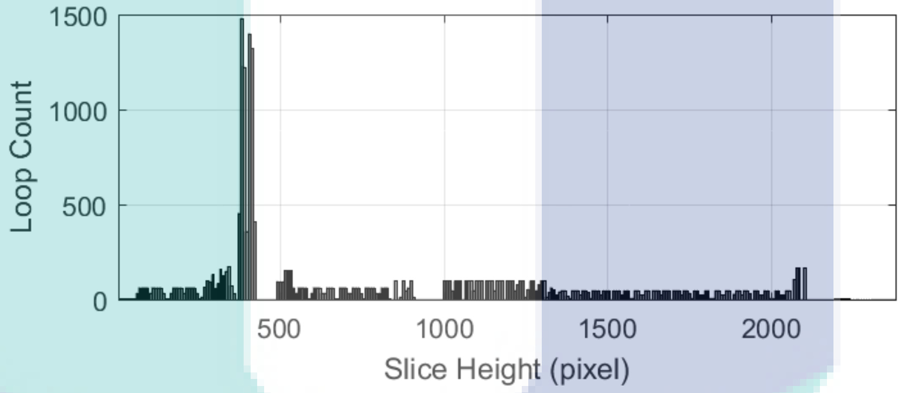
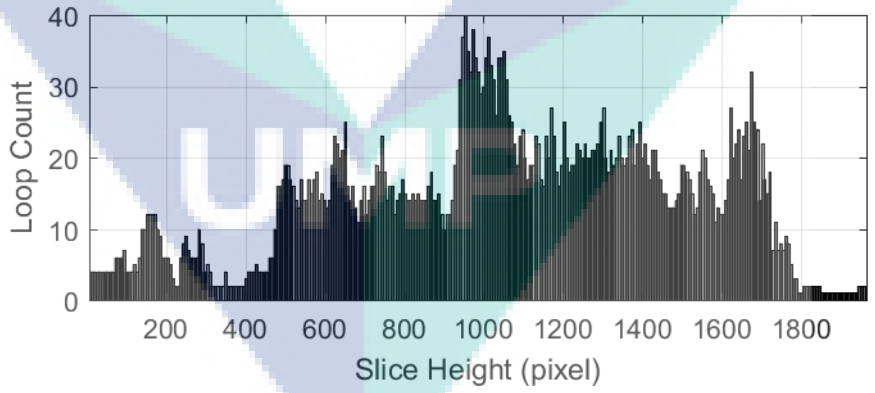
Model	Performance Graph
Dragon	
Eiffel Tower	
Gundam	

Table 4.4 Continued

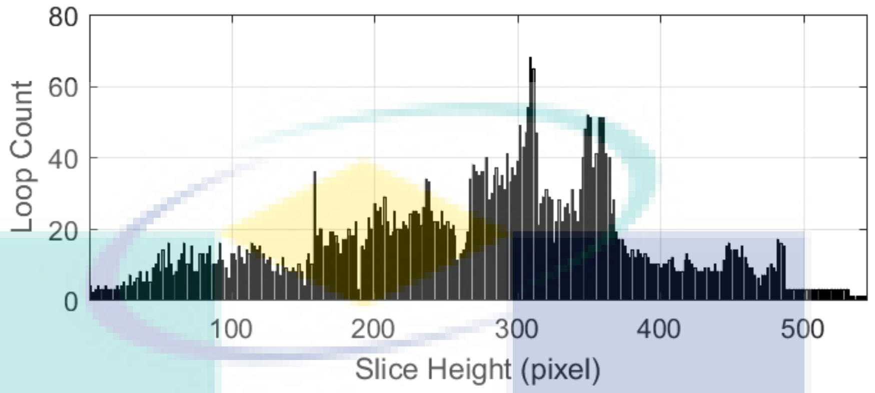
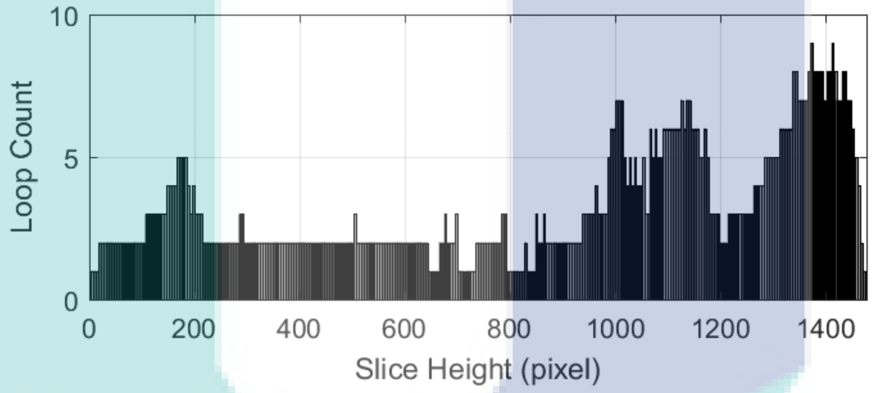
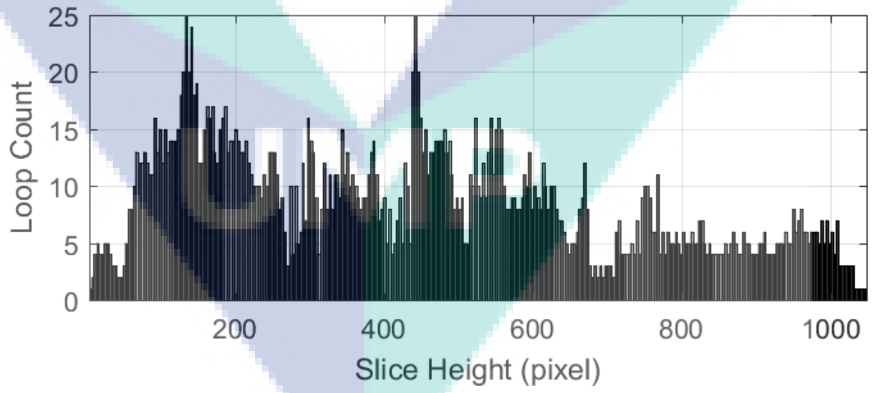
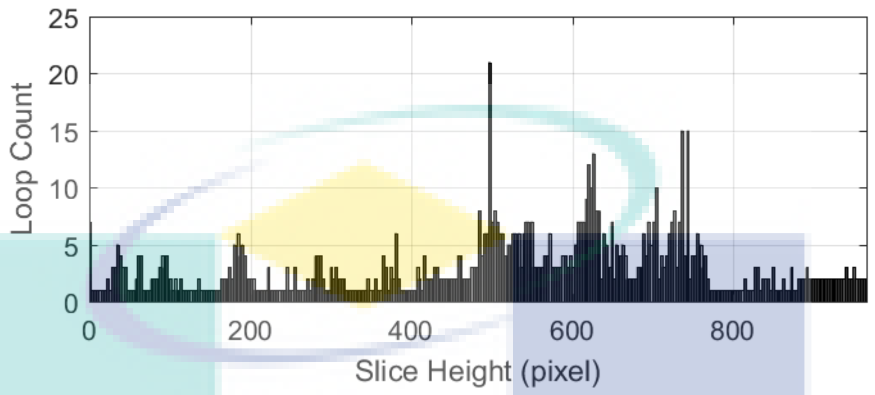
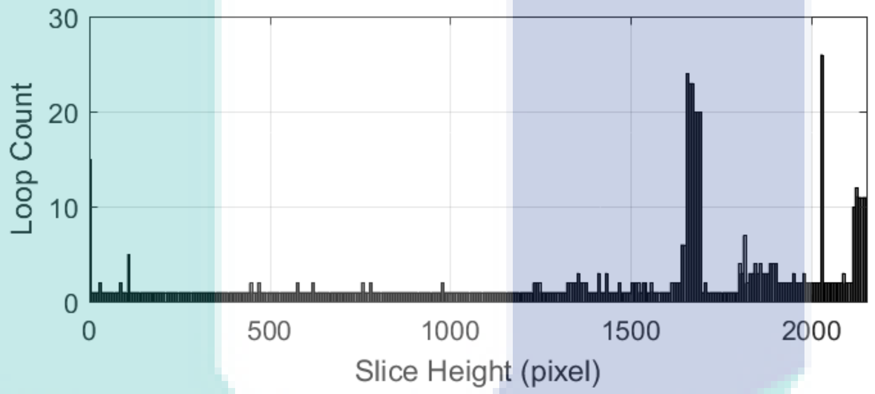
Model	Performance Graph
Speedster	
Heart	
Dreadnaught	

Table 4.4 Continued

Model	Performance Graph
Worm	
Spiral Tower	

In earlier section, it could be hypothesized that the computational time for contour loop algorithm has a similarity with the number of intersecting facets. Thus, to further support this statement, a normalized correlation method is used to measure the similarities between these two results. The Equation 4.1 is the equation of normalized correlation which is written as:

$$\mathrm{NC} = \frac{\sum x_n y_n}{\sqrt{\sum {x_n}^2 \sum {y_n}^2}} \qquad 4.1$$

where the x is the data of contour loop time, y is the intersecting facet number, n is the number of elements, and NC is the normalized correlation. The normalized correlation is also tested for the relations between contour loop time and contour loop counts by using the contour loop time as variable x and contour loop counts as y according to the previous Equation 4.1. By using this equation for each model, the results of normalized correlation are tabulated in Table 4.5 where CT is the contour loop time, LC is the contour loop counts and IF is the number of intersecting facet.

Table 4.5 Calculated normalized correlation of each STL model

STL Model	**Normalized Correlation**	
	CT vs LC	**CT vs IF**
Sphere	0.93	0.93
Dragon	0.77	0.93
Eiffel Tower	0.92	0.96
Gundam	0.89	0.96
Speedster	0.87	0.96
Heart	0.64	0.97
Dreadnaught	0.86	0.94
Worm	0.72	0.91
Spiral Tower	0.49	0.89

The results of normalized correlation for contour loop time against contour loop counts show that the strong correlation only implied to certain model such as Sphere, Eiffel Tower, Gundam, and Speedster. But the rest of the models show weak correlations. Thus, it can be concluded that the number of contour loop counts do not significantly affect the contour loop time. On the other hand, the number of intersecting facets for every model has strong correlations with the contour loop execution time. Hence, earlier hypothesis that states increasing number of intersecting facets also increase the contour loop algorithm computational time.

4.5 Total Computational Time

Overall, total computational time is measured by adding both slicing time and contour loop time to give the total time required (in milliseconds) to generate the contour at each slicing height for DLP 3D printing contour projection. The result is tabulated in Table 4.6 below based on different STL model.

Table 4.6 Total computational time required for each slicing height

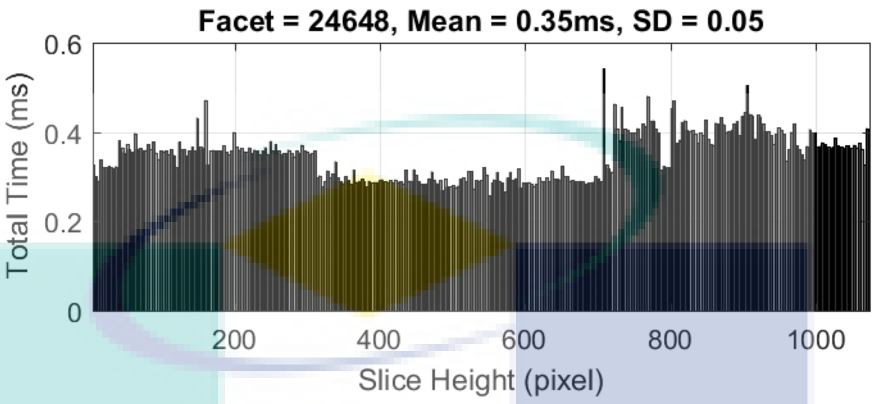
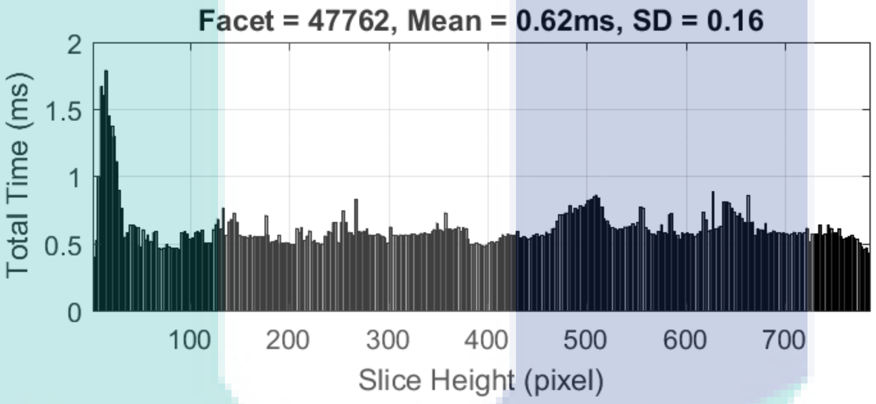
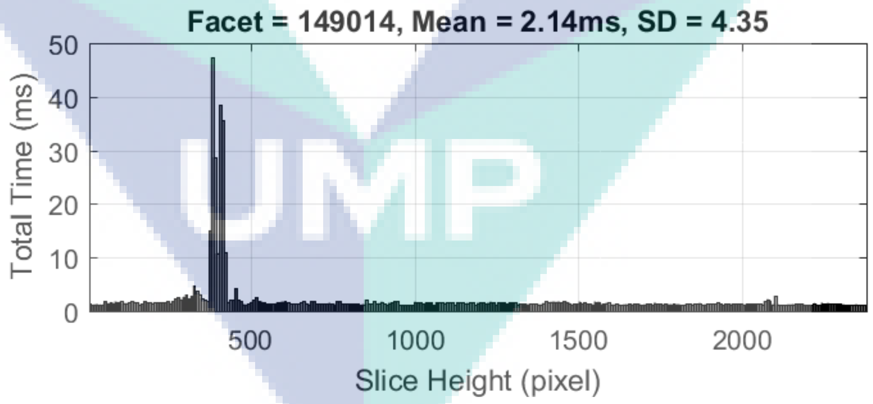
Model	Performance Graph
Sphere	
Dragon	
Eiffel Tower	

Table 4.6 Continued

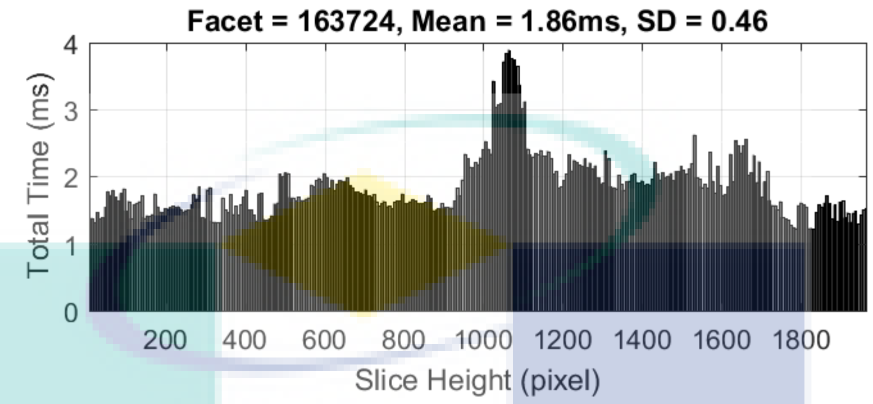
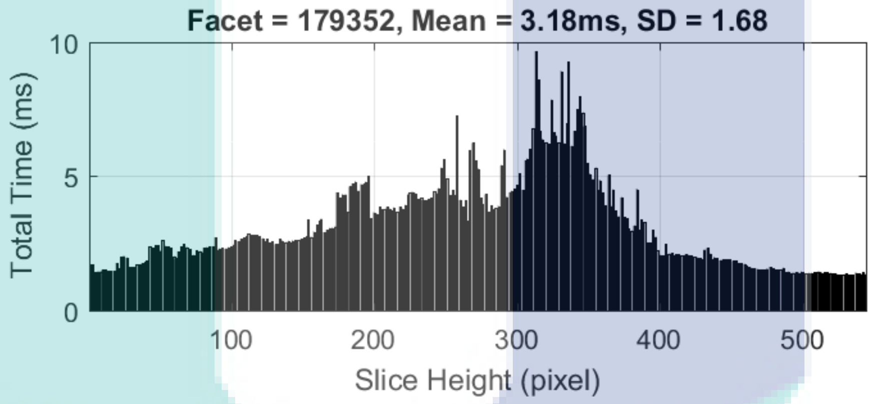
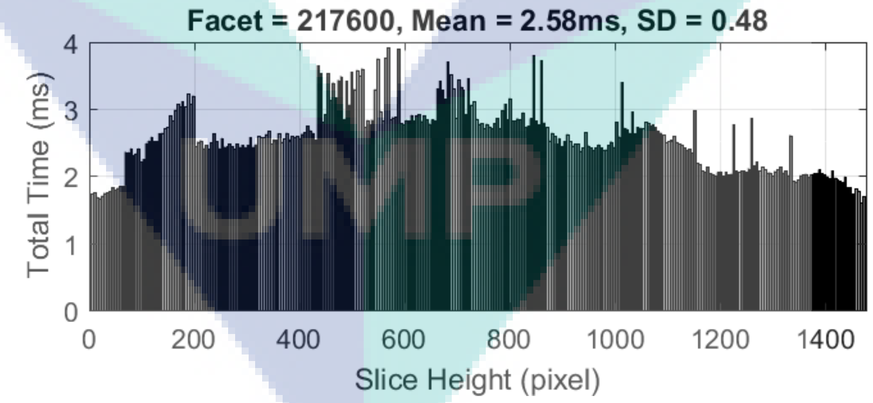
Model	Performance Graph
Gundam	
Speedster	
Heart	

Table 4.6 Continued

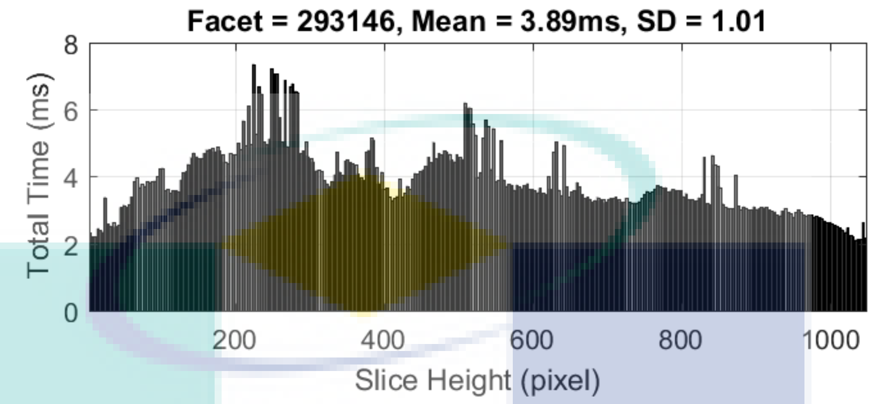
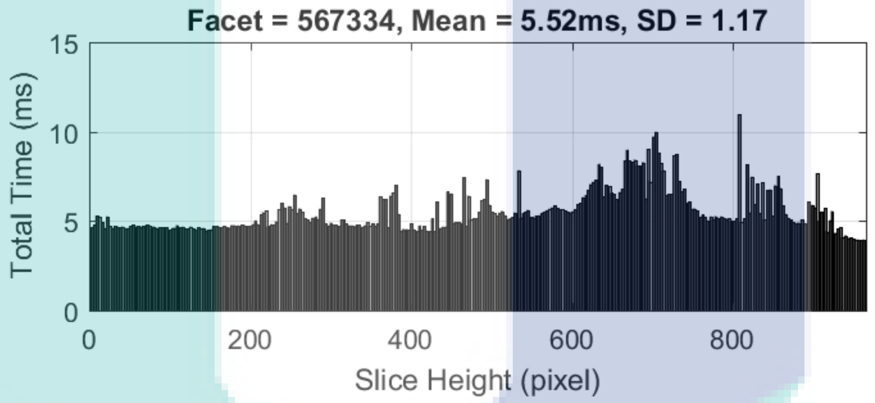
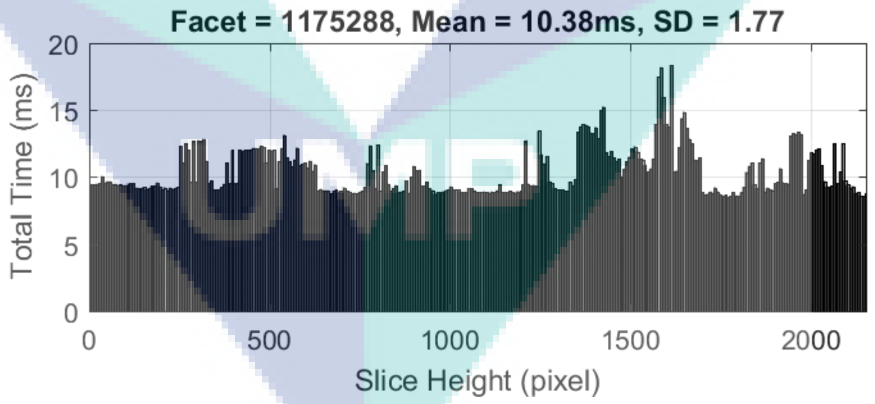
Model	Performance Graph
Dreadnaught	
Worm	
Spiral Tower	

Table 4.6 shows the total computational time required with respect to each slicing height. It can be observed that, increasing facet number also increases its mean computational time. It natural since more complex model will have more facet counts and requires more computational time. At certain slicing height, several peaks can be seen. This indicates that around that particular slicing height has a greater number of intersecting facets compared to the other slicing heights.

4.6 Visualization of Contour Generation Algorithm

In this section, the top view of stacked generated contour for some STL models are shown to visualize the 3D model the DLP 3D printing process.

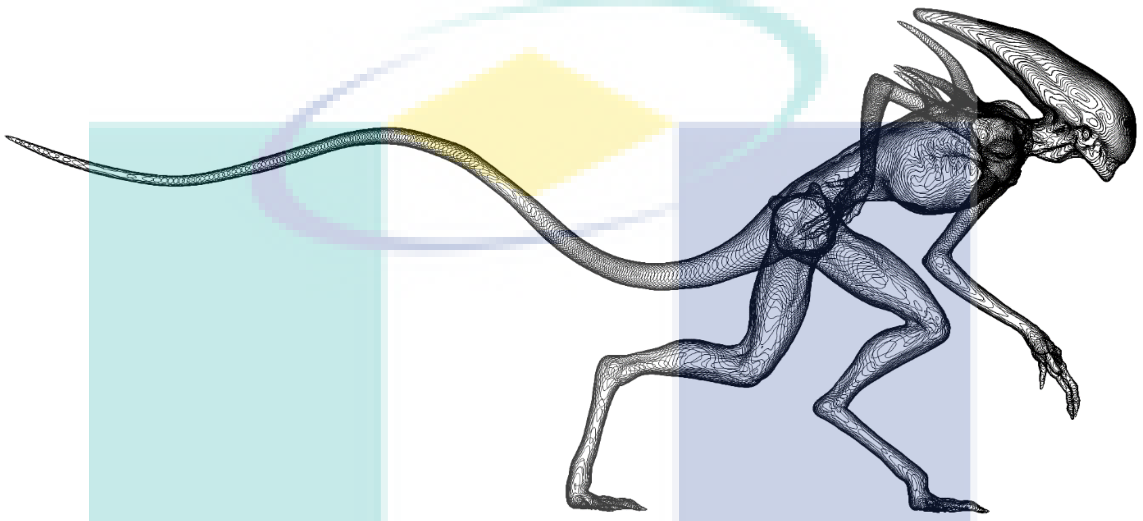


Figure 4.4 Stacked contours Alien model (side slicing)

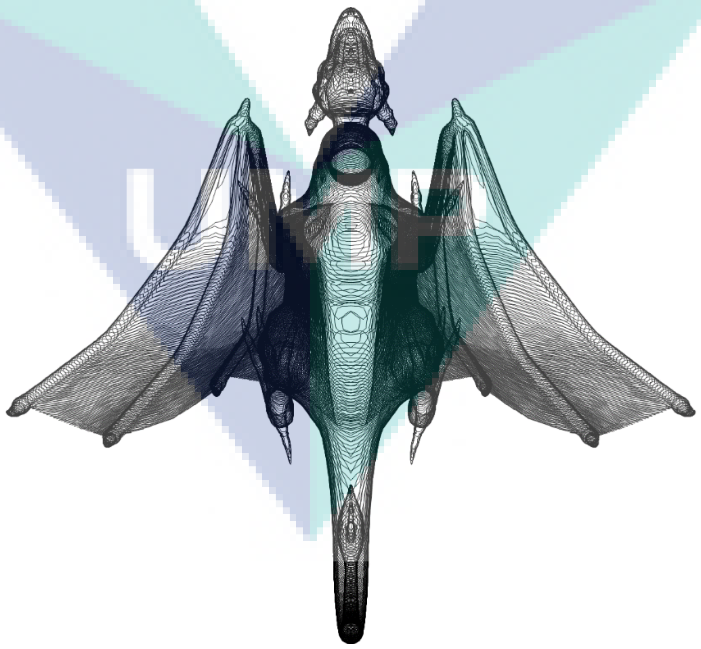


Figure 4.5 Stacked contours Dragon model (bottom-up slicing)

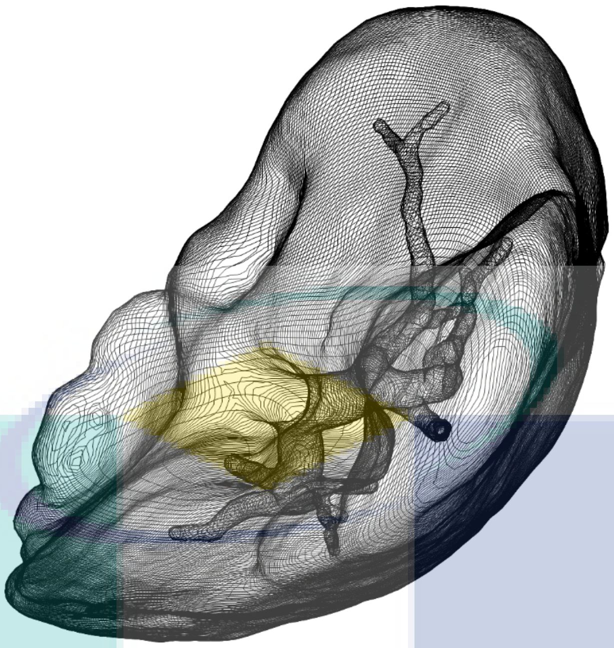


Figure 4.6 Stacked contours Liver model (bottom-up slicing)

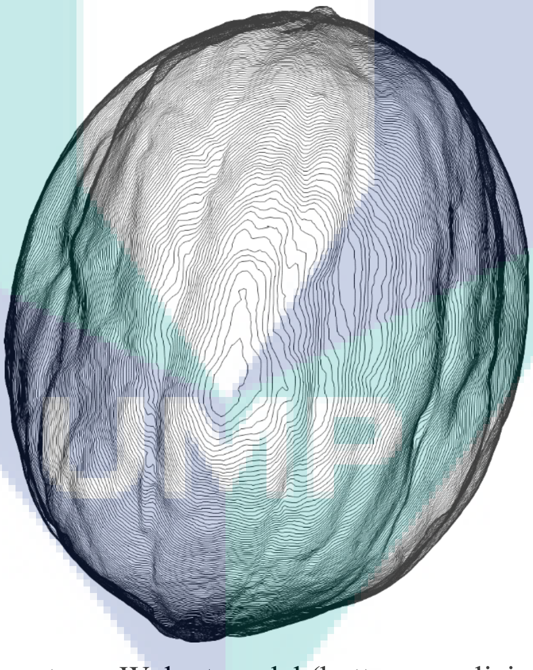


Figure 4.7 Stacked contours Walnut model (bottom-up slicing)

4.7 Comparison of Slicing and Contour Loop algorithms

Both of the slicing and contour algorithms are re-written in C++11 and tested on Intel i7-6700 3.40 GHz CPU with 4 GB RAM workstation for benchmarking with the results obtained by the work from literature. The reason is to evaluate the performances of both algorithms compared to the algorithms result obtained by other researcher in their work. All the parameters such as the STL model, its facet count, and the number of slicing planes is exactly the same as the one used in their research paper. The proposed algorithms are measured using built-in <QElapsedTimer> library provided by the Qt Creator software to obtain the execution time in milliseconds.

In the work of Minetto on Contour Generation algorithm, the author reimplemented other researcher algorithm (Park) written in C++ and executed on Intel i7 3.4 GHz workstation. The results are measured based on minimum execution time for each algorithm in seconds. The results are also compared to commercial 3D printing software Slic3r (Minetto et al., 2017; Park, 2003). Using the data obtained by the author, our proposed algorithm results are validated in Table 4.7, Table 4.8, and Table 4.9 below.

Table 4.7 Time measurement and comparison for slicing algorithm

Model	**Facet Count**	**Layer Count**	**Slicing Algorithms (s)**		
			Park	**Slic3r**	**Proposed**
Liver	38142	6242	1.28	**0.32**	1.48
Femur	42150	3155	0.53	**0.16**	2.53
Bunny	270021	1547	2.70	**0.29**	2.87
Demon	935236	3126	20.12	**1.28**	20.15
Rider	1281950	849	6.37	**0.54**	7.49

Bolded value in Table 4.7 above shows the best runtime among the test results. Our proposed slicing algorithm is the slowest among other two algorithms with on average 22.72% slower than Park and 89.68% slower than Slic3r. This is because the proposed algorithm utilized vectors and 3D points computation instead of the commonly used extrapolation method which used more simplified mathematical equation to compute. However, since the proposed slicing algorithm uses vector coordinate computation, manipulating the slicing direction will be much easier compared to extrapolation. The extrapolation method works best for one direction slicing, but in order to modify the slicing angle, the algorithm has to change every single point that exist in the STL model. This heavy task will demand more computational time to be performed for each time the user wanted to change the slicing angle. Another reason is that the proposed slicing algorithm is the slowest because it includes point conversion (Line to

Pixel Mapping Algorithm) that changes the data type from *Float* to *Unsigned Short* that gives advantages in the proposed Contour Loop algorithm.

Table 4.8 Time measurement and comparison for contour loop algorithm

Model	Facet Count	Layer Count	Contour Loop Algorithms (seconds)		
			Park	Slic3r	Proposed
Liver	38142	6242	35.97	3.57	**0.003**
Femur	42150	3155	16.59	2.00	**0.002**
Bunny	270021	1547	22.00	8.51	**0.004**
Demon	935236	3126	140.77	69.49	**0.022**
Rider	1281950	849	27.82	25.02	**0.001**

As shown in Table 4.8 above, our proposed contour loop algorithm is the fastest compared to the rest of the algorithm with on average 1,199,972.73% faster than Park and 649,822.73% faster than Slic3r. Our proposed algorithm uses simple head-to-tail contour algorithm. The key to the fast execution time of the proposed algorithm lies within the data type of the Line Segment. Normally, in the field of computational geometry programming, *Float* data type is often used by the programmers to reduce truncation errors and improve execution time. The *Float* data type has data size of 32-bit (4 Bytes) which capable of storing number ranging between -3.40282e+38 until +3.40282e+38. This large data type demands more processing time of the CPU compared to smaller size data type. Line-to-Pixel map algorithm which was discussed in Chapter 3 converted the floating-point data type into *Unsigned Short* data type. The *Unsigned Short* is a 16-bit (2 Bytes) data type which is smaller than *Float* data type and it does not has decimal points. This data type able to store numbers ranging from 0 until 65535. The Line-to-Pixel map algorithm scales the floating points data to be within the range of *Unsigned Short* data. The main idea of the proposed contour loop algorithm is rejection of the use of *Float* data type. It is because in DLP 3D printing technology, the end device is always a projection device which are constrained by the number of pixels in each row. Current display/projection technology still has not exceeded 65535 pixels in each row. Hence, it is still within the range of *Unsigned Short* data type. Furthermore, the operation using *Unsigned Short* are much faster and more accurate compared to *Float* data type. Using this ideology, the proposed contour loop scales the contours depending on the display resolutions of the DLP 3D printer projection device. The algorithm is proven to be more than 100 times faster as shown in Table 4.8 above.

Table 4.9 Time measurement and comparison for total computational time

Model	Facet Count	Layer Count	Total Time for both algorithms (seconds)		
			Park	**Slic3r**	**Proposed**
Liver	38142	6242	37.24	3.89	**1.483**
Femur	42150	3155	17.12	2.16	**2.532**
Bunny	270021	1547	24.70	8.80	**2.874**
Demon	935236	3126	160.89	70.77	**20.174**
Rider	1281950	849	34.19	25.56	**7.489**

Table 4.9 above shows the total execution time for both slicing and contour loop algorithm. The results are obtained by summing both results from slicing and contour loop algorithm. As seen in the Table 4.9, the proposed algorithms are the fastest algorithm by comparison with average 960.15% faster than Park and 169.18% faster Slic3r. Most of the computational time is consumed by the proposed slicing algorithm. However, the proposed slicing algorithm has its own merits as discussed earlier.

CHAPTER 5

CONCLUSION

5.1 Conclusion

Mask projection stereolithography is a recent discovery in 3D printing industry. It harnesses the power of UV light to cure the photocurable resin to form the solid 3D model. Each layer is projected through transparent glass into the resin vat and built layer-by-layer until the process completes. STL CAD format is considered as de facto in 3D printing. This format is generated from multiple triangular meshes which are generated by tessellation process. The STL model undergoes contour generation algorithm to generate the necessary contour to be projected to the photocurable resin. In this study, a real-time contour generation algorithm is presented which involves series of algorithms. The algorithm consists of slicing algorithm, pixel-mapping algorithm, and contour loop algorithm. Each of these algorithms have been thoroughly studied, developed, and evaluated.

The developed slicing algorithm is based on line-plane intersection model which is computationally efficient and simple. The slicing algorithm generates multiple arbitrary line segments that act as the bones of the contour. But the line segments are not digitally connected to each other. Thus, a contour loop algorithm is required to connect each of these line segments into one or multiple closed-loop contour.

The line segments generated from the slicing algorithm are mapped referring the resolution of the projection device using the proposed pixel-mapping algorithm. The pixel-mapping algorithm remapped the line segments which use floating point coordinates into unsigned int pixel coordinate of the projection device. Then, these mapped line segments are connected using contour loop algorithm.

The contour loop algorithm is based on head-to-tail search algorithm. By assigning the first point from the list of line segments, the algorithm recursively searches and compares the remaining line segments and eventually form one or more closed-loop contour. The results of contour loop algorithm show that the algorithm is very fast and efficient regardless the facet number of the STL model. But the algorithm performs a bit slower when the layer has multiple closed-loop contours.

The algorithms are executed on Intel i3-2350M CPU 2.30 GHz with 6 GB RAM workstation and written in VB.NET programming language. For peer result comparison with the algorithm obtained from the journal, the algorithms are re-written in C++11 and tested on Intel i7-6700 3.40 GHz workstation with 4 GB RAM similar to the referenced literature. The result finds that the proposed slicing algorithm is slower compares to the result from literature with on average 22.72% slower than Park and 89.68% slower than Slic3r software. For contour loop algorithm, the results are significantly faster than the one from the literature with on average 1,199,972.73% faster than Park and 649,822.73% faster than Slic3r. Thus, the total required time for contour generation has improved by 960.15% compared to Park, and 169.18% compared to Slic3r software.

Overall, the contour generation algorithm proposed in this study shows promising results. According to the measured computational time, the algorithm can operate in real-time due to fast computational time required to generate 2D contour at any slicing height. This allows the algorithm to solve the memory storage issue whilst achieving the highest printing resolution and mechanical properties.

5.2 Future Work

The proposed algorithm only covers the contour generation process of the mask projection stereolithography 3D printing process. It does not cover the support generation process which is crucial for stereolithography printing process. In order to fulfill the pre-processing stage of the stereolithography, a support generation algorithm is required.

Another improvement that can be made to the algorithm is the parallel computation. Current multi-core technology in modern CPU allows multi-tasking operation. Thus, distributing the processes among cores can rapidly improve the computational time for the algorithm.

One of most important features in 3D printing process is the dimensional accuracy of the printed product. It is important for the printer to deliver exact dimension as given by the STL model so that the printed product does not need to be reworked. The implementation of the proposed algorithm on real hardware has not yet been studied. Hence, its dimensional accuracy is also important topic for further improvement of the algorithm.

REFERENCES

Barequet, G., & Sharir, M. (1995). Filling gaps in the boundary of a polyhedron. *Computer Aided Geometric Design*, *12*(2), 207–229. https://doi.org/10.1016/0167-8396(94)00011-G

Bloomenthal, J. (1988). Polygonization of implicit surfaces. *Computer Aided Geometric Design*, *5*(4), 341–355. https://doi.org/10.1016/0167-8396(88)90013-1

Boddapati, A. (2010). *Modeling Cure Depth During Photopolymerization of Multifunctional Acrylates*. Georgia Institute of Technology.

Cătălin IANCU, P., Daniela IANCU, E., & Alin STĂNCIOIU, D. (2010). From Cad Model to 3D Print Via STL Format. *Academica Brâncuşi Târgu Jiu*, *1*(1), 1844–640.

Choi, S. H., & Kwok, K. T. (1999). A Memory Efficient Slicing Algorithm for Large STL Files. In *Proceedings of the 31st International Conference on Computers and Industrial Engineering* (pp. 155–162).

Dendukuri, D., Panda, P., Haghgooie, R., Kim, J. M., Hatton, T. A., & Doyle, P. S. (2008). Modeling of oxygen-inhibited free radical photopolymerization in a PDMS microfluidic device. *Macromolecules*, *41*(22), 8547–8556. https://doi.org/10.1021/ma801219w

Dizon, J. R. C., Espera, A. H., Chen, Q., & Advincula, R. C. (2018). Mechanical characterization of 3D-printed polymers. *Additive Manufacturing*, *20*, 44–67. https://doi.org/10.1016/j.addma.2017.12.002

Gao, F., Yang, Y., & Li, L. (1999). Visible light photopolymerization of Methylmethacrylate coninitiated with titanocene and ketocoumarin dye. *Chinese Journal of Polymer Science*, *17*(5), 465–470.

Gregori, R. M. M. H., Volpato, N., Minetto, R., & Silva, M. V. G. Da. (2014). Slicing Triangle Meshes: An Asymptotically Optimal Algorithm. *2014 14th International Conference on Computational Science and Its Applications*, 252–255. https://doi.org/10.1109/ICCSA.2014.58

Hayasi, M. T., & Asiabanpour, B. (2009). Machine path generation using direct slicing from design-by-feature solid model for rapid prototyping. *International Journal of Advanced Manufacturing Technology*, *45*(1–2), 170–180. https://doi.org/10.1007/s00170-009-1944-8

Hemant, P., Kulkarni, P., & Thokale, M. (2015). 3D Printing Technology. *International Journal of Multidisciplinary Research and Development*, *2*(3), 351–358. Retrieved from http://3dprintingindustry.com/3d-printing-basics-free-beginners-guide/technology/

Huang, S. H., Zhang, L. C., & Han, M. (2002). An effective error-tolerance slicing algorithm for STL files. *International Journal of Advanced Manufacturing Technology*, *20*(5), 363–367. https://doi.org/10.1007/s001700200164

Huang, X., Yao, Y., & Hu, Q. (2012). Research on the rapid slicing algorithm for NC milling based on STL model. *Communications in Computer and Information Science*, *325 CCIS*(PART 3), 263–271. https://doi.org/10.1007/978-3-642-34387-2_30

Jacob, G. G. K., Kai, C. C., & Mei, T. (1999). Development of a new rapid prototyping interface. *Computers in Industry*, *39*(1), 61–70. https://doi.org/10.1016/S0166-3615(98)00124-9

Jacobs, P. F. (1992). Fundamentals of Stereolithography. *Society of Manufacturing Engineers*, (July), 196–211. https://doi.org/10.1017/CBO9781107415324.004

Jing Hu. (2017). Study On STL-Based Slicing Process For 3D Printing. *Solid Freeform Fabrication*, 885–895.

Kang, H. W., Park, J. H., & Cho, D. W. (2012). A pixel based solidification model for projection based stereolithography technology. *Sensors and Actuators, A: Physical*, *178*, 223–229. https://doi.org/10.1016/j.sna.2012.01.016

Kirschman, C., & Jara-Almonte, C. (1992). A parallel slicing algorithm for solid freeform fabrication processes. *Solid Freeform Fabrication Symposium*.

Kitano, H. (2012). Advances In light-induced polymerizations : I . Shadow cure in free radical photopolymerizations , II . Experimental and modeling studies of photoinitiator systems for effective polymerizations with LEDs. *PhD Dissertation*, 195.

Koc, B., Ma, Y., & Lee, Y. S. (2000). Smoothing STL files by Max-Fit biarc curves for rapid prototyping. *Rapid Prototyping Journal*, *6*(3), 186–203. https://doi.org/10.1108/13552540010337065

Kodama, H. (1981). Automatic method for fabricating a three-dimensional plastic model with photo-hardening polymer. *Review of Scientific Instruments*, *52*(11), 1770–1773. https://doi.org/10.1063/1.1136492

Królikowski, M., & Grzesiak, D. (2014). Technological Restrictions of Lightweight Lattice Structures Manufactured by Selective Laser Melting of Metals. *Advances in Manufacturing Science and Technology*, *38*(2). https://doi.org/10.2478/amst-2014-0012

Kulkarni, P., Marsan, A., & Dutta, D. (2000). Review of process planning techniques in layered manufacturing. *Rapid Prototyping Journal*. https://doi.org/10.1108/13552540010309859

Kumar, V., & Dutta, D. (1997). An assessment of data formats for layered manufacturing.

Advances in Engineering Software, *28*(3), 151–164. https://doi.org/10.1016/S0965-9978(96)00050-6

Lederle, F., Meyer, F., Brunotte, G.-P., Kaldun, C., & Hübner, E. G. (2016). Improved mechanical properties of 3D-printed parts by fused deposition modeling processed under the exclusion of oxygen. *Progress in Additive Manufacturing*, *1*(1–2), 3–7. https://doi.org/10.1007/s40964-016-0010-y

Lee, J. H., Prud'homme, R. K., & Aksay, I. a. (2001). Cure depth in photopolymerization: Experiments and theory. *Journal of Materials Research*, *16*(12), 3536–3544. https://doi.org/10.1557/JMR.2001.0485

Leong, K. F., Chua, C. K., & Ng, Y. M. (1996). A study of stereolithography file errors and repair. Part 1. Generic solution. *International Journal of Advanced Manufacturing Technology*, *12*(6), 407–414. https://doi.org/10.1007/BF01186929

Manmadhachary, A., Ravi Kumar, Y., & Krishnanand, L. (2016). Improve the accuracy, surface smoothing and material adaption in STL file for RP medical models. *Journal of Manufacturing Processes*, *21*, 46–55. https://doi.org/10.1016/j.jmapro.2015.11.006

Minetto, R., Volpato, N., Stolfi, J., Gregori, R. M. M. H., & da Silva, M. V. G. (2017). An optimal algorithm for 3D triangle mesh slicing. *CAD Computer Aided Design*, *92*, 1–10. https://doi.org/10.1016/j.cad.2017.07.001

Mu, Q., Wang, L., Dunn, C. K., Kuang, X., Duan, F., Zhang, Z., … Wang, T. (2017). Digital light processing 3D printing of conductive complex structures. *Additive Manufacturing*, *18*, 74–83. https://doi.org/10.1016/j.addma.2017.08.011

Pan, X., Chen, K., & Chen, D. (2014). Development of rapid prototyping slicing software based on STL model. *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2014*, (51175395), 191–195. https://doi.org/10.1109/CSCWD.2014.6846840

Pan, Y., Zhou, C., & Chen, Y. (2012). A Fast Mask Projection Stereolithography Process for Fabricating Digital Models in Minutes. *Journal of Manufacturing Science and Engineering*, *134*(5), 051011. https://doi.org/10.1115/1.4007465

Pandey, P. M., Reddy, N. V., & Dhande, S. G. (2003). Real time adaptive slicing for fused deposition modelling. *International Journal of Machine Tools and Manufacture*, *43*(1), 61–71. https://doi.org/10.1016/S0890-6955(02)00164-5

Pandey, R. (2014). *Photopolymers in 3D printing applications*.

Park, S. C. (2003). Tool-path generation for Z-constant contour machining. *CAD Computer Aided Design*, *35*(1), 27–36. https://doi.org/10.1016/S0010-4485(01)00173-7

Piegl, L. A., & Richard, A. M. (1995). Tessellating trimmed nurbs surfaces. *Computer-Aided Design*, *27*(1), 16–26. https://doi.org/10.1016/0010-4485(95)90749-6

Ranellucci, A., & Lenox, J. (2011). Slic3r - G-code generator for 3D printers. Retrieved from http://www.slic3r.org/

Szilvśi-Nagy, M., & Mátyási, G. (2003). Analysis of STL files. *Mathematical and Computer Modelling*, *38*(7–9), 945–960. https://doi.org/10.1016/S0895-7177(03)90079-3

Tang, Y. (2005). *Stereolithography Cure Process Modeling*. Georgia Institute of Technology.

Tang, Y., Henderson, C. L., Muzzy, J., & Rosen, D. W. (2004). Stereolithography Cure Process Modeling Using Acrylate Resin. *Fifteenth Solid Freeform Fabrication (SFF) Symposium*, 612–623. https://doi.org/10.1017/CBO9781107415324.004

Tian, R., Liu, S., & Zhang, Y. (2018). Research on fast grouping slice algorithm for STL model in rapid prototyping. *Journal of Physics: Conference Series*, *1074*, 012165. https://doi.org/10.1088/1742-6596/1074/1/012165

Topçu, O., Taşcıoğlu, Y., & Ünver, H. Ö. (2011). A Method for Slicing CAD Models in Binary STL Format. *6th International Advanced Technologies Symposium (IATS'11)*, (May), 141–148. Retrieved from http://web.firat.edu.tr/iats/cd/subjects/Manufacturing/MTE-31.pdf

Tumbleston, J. R., Shirvanyants, D., Ermoshkin, N., Janusziewicz, R., Johnson, A. R., Kelly, D., … Desimone, J. M. (2015). Continuous liquid interface production of 3D objects. *Science*, *347*(6228), 1349–1352.

Tyvaert, I., Fadel, G., & Rouhaud, E. (1999). A methodology to create STL files from data point clouds generated with a coordinate measuring machine. *Annual Interantional Solid Freeform Fabrication Symposium*, 47–58.

Vatani, M., Rahimi, A. R., Brazandeh, F., & Sanati Nezhad, A. (2009). An enhanced slicing algorithm using nearest distance analysis for layer manufacturing. *Proceedings of World Academy of Science, Engineering and Technology*, *37*(1), 721–726. Retrieved from http://www.waset.ac.nz/journals/waset/v49/v49-130.pdf

Wang, D. X., Guo, D. M., Jia, Z. Y., & Leng, H. W. (2006). Slicing of CAD models in color STL format. *Computers in Industry*, *57*(1), 3–10. https://doi.org/10.1016/j.compind.2005.03.007

Wong, H.-T. T., Huang, Y., Tsang, S.-C., & Lam, M.-L. (2017). Real-time model slicing in arbitrary direction using octree. *ACM SIGGRAPH 2017 Posters on - SIGGRAPH '17*, 1–2. https://doi.org/10.1145/3102163.3102185

Wong, K. V., & Hernandez, A. (2012). A Review of Additive Manufacturing. *ISRN Mechanical Engineering*, *2012*, 1–10. https://doi.org/10.5402/2012/208760

Wu, T., & Cheung, E. H. M. (2006). Enhanced STL. *International Journal of Advanced Manufacturing Technology*, *29*(11–12), 1143–1150. https://doi.org/10.1007/s00170-005-0001-5

Xu, H., Weihua, J., Li, M., & Li, W. (2017). A slicing model algorithm based on STL model for additive manufacturing processes. *Proceedings of 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference, IMCEC 2016*, 1607–1610. https://doi.org/10.1109/IMCEC.2016.7867489

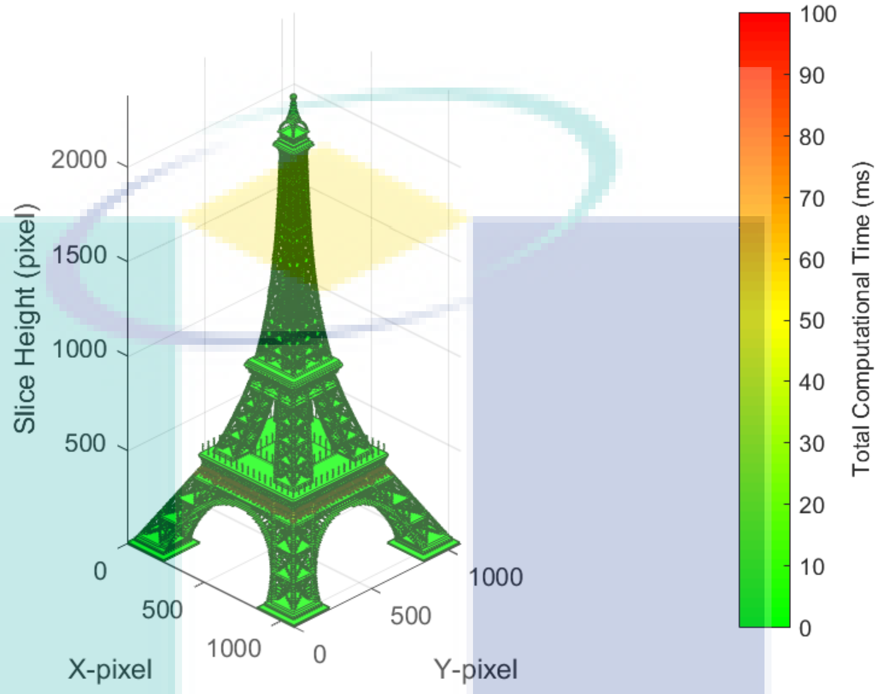
Ye, H., Zhou, C., & Xu, W. (2017). Image-Based Slicing and Tool Path Planning for Hybrid Stereolithography Additive Manufacturing. *Journal of Manufacturing Science and Engineering*, *139*(7), 071006. https://doi.org/10.1115/1.4035795

Zhang, Z., & Joshi, S. (2015). An improved slicing algorithm with efficient contour construction using STL files. *International Journal of Advanced Manufacturing Technology*, *80*(5–8), 1347–1362. https://doi.org/10.1007/s00170-015-7071-9

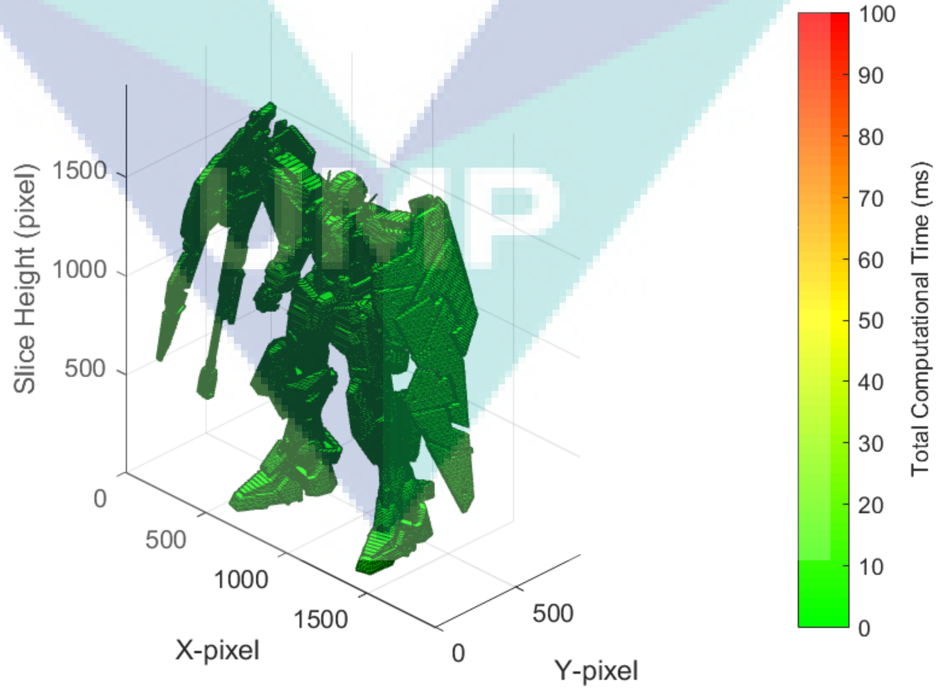
Zheng, X., Cheng, K., Zhou, X., Lin, J., & Jing, X. (2018). An adaptive direct slicing method based on tilted voxel of two-photon polymerization. *International Journal of Advanced Manufacturing Technology*, *96*(1–4), 521–530. https://doi.org/10.1007/s00170-017-1507-3

Zhou, M. Y., Xi, J. T., & Yan, J. Q. (2004). Adaptive direct slicing with non-uniform cusp heights for rapid prototyping. *International Journal of Advanced Manufacturing Technology*, *23*(1–2), 20–27. https://doi.org/10.1007/s00170-002-1523-8

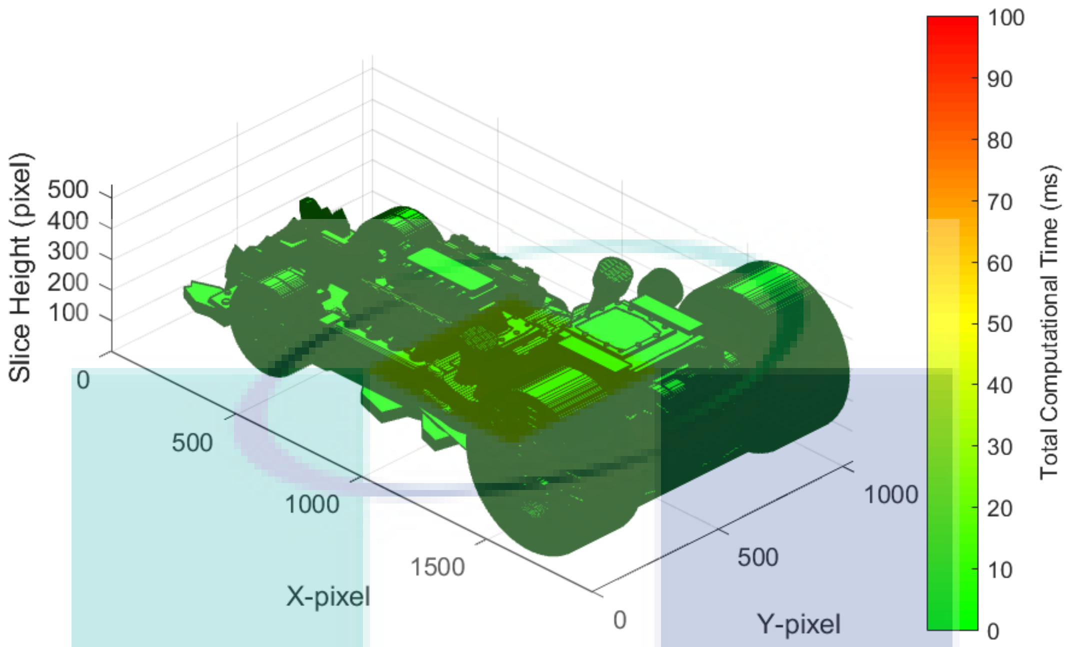
APPENDIX A
ADDITIONAL SLICING RESULT



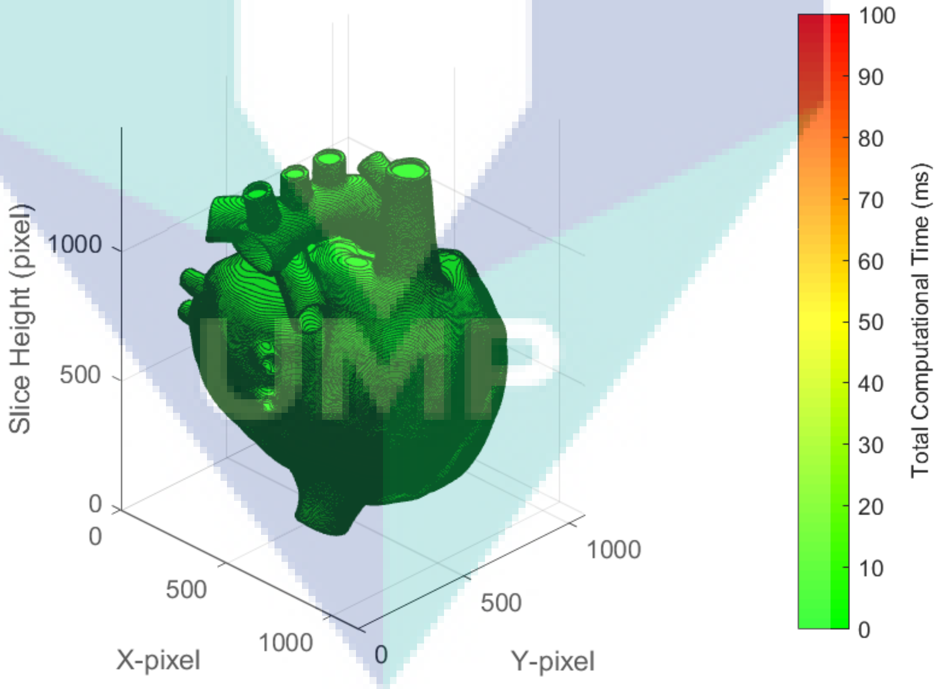
Eiffel Tower (Facet: 149014)



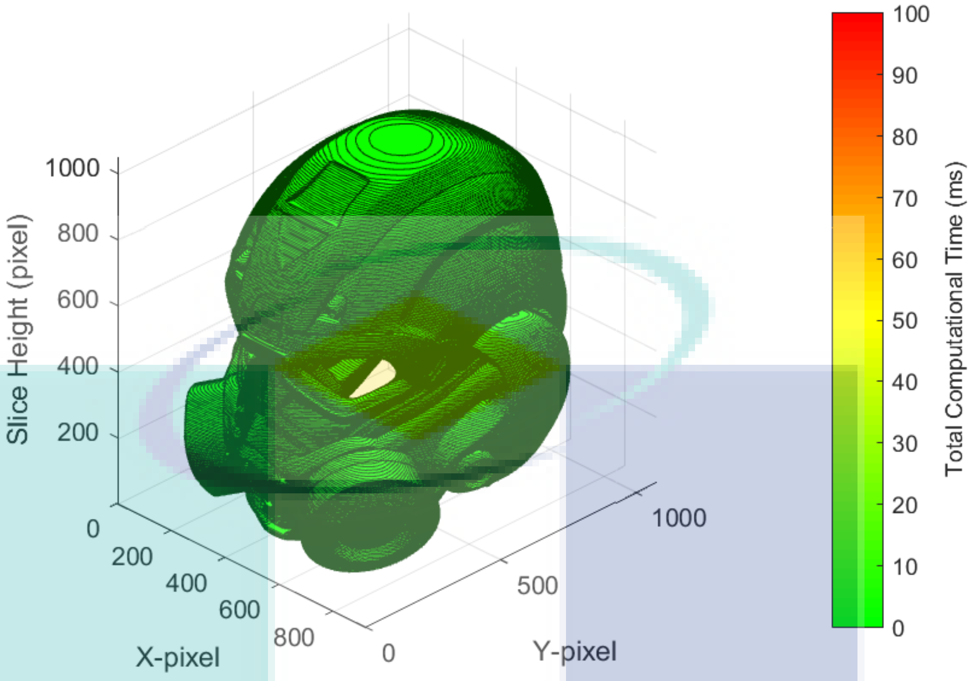
Gundam (Facet: 163724)



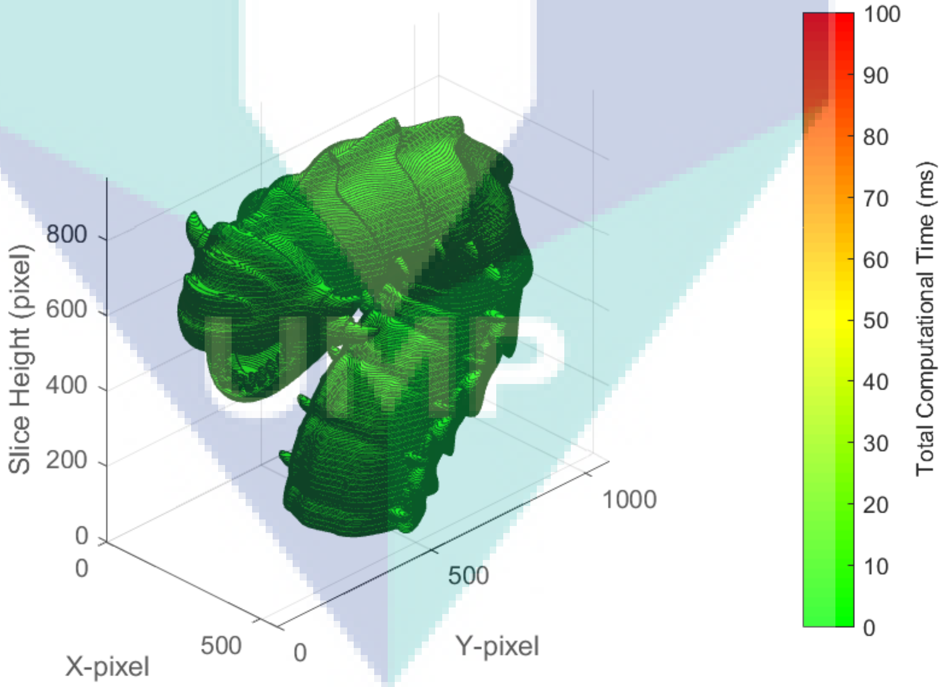
Speedster (Facet: 179352)



Heart (Facet: 217600)



Dreadnaught (Facet: 293146)



Worm (Facet: 567334)

APPENDIX B
PSEUDOCODE (VB.NET)

Facet Class

```
Public Class Facet
    Structure Point3D
        Dim X As Double
        Dim Y As Double
        Dim Z As Double

        'Constructor Point3D
        Public Sub New(ByVal x As Double, ByVal y As Double, ByVal z As Double)
            Me.X = x
            Me.Y = y
            Me.Z = z
        End Sub

        Public Shared Function Dot(ByRef p1 As Point3D, ByRef p2 As Point3D) As Double
            Return (p1.X * p2.X) + (p1.Y * p2.Y) + (p1.Z * p2.Z)
        End Function

        Public Shared Function Cross(ByRef p1 As Point3D, ByRef p2 As Point3D) As Point3D
            Return New Point3D(p1.Y * p2.Z - p1.Z * p2.Y, p1.X * p2.Z - p1.Z * p2.X, p1.X * p2.Y - p1.Y * p2.X)
        End Function

        Public Shared Function LengthBetween(ByRef p1 As Point3D, ByRef p2 As Point3D) As Double
            Return Math.Sqrt(Math.Pow((p1.X - p2.X), 2) + Math.Pow((p1.Y - p2.Y), 2) + Math.Pow((p1.Z - p2.Z),
2))
        End Function

        Public Shared Function LengthSq(ByRef p1 As Point3D, ByRef p2 As Point3D) As Double
            Return Math.Pow((p1.X - p2.X), 2) + Math.Pow((p1.Y - p2.Y), 2) + Math.Pow((p1.Z - p2.Z), 2)
        End Function

        Public Shared Operator +(ByVal p1 As Point3D, ByVal p2 As Point3D) As Point3D
            Return New Point3D(p1.X + p2.X, p1.Y + p2.Y, p1.Z + p2.Z)
        End Operator

        Public Shared Operator -(ByVal p1 As Point3D, ByVal p2 As Point3D) As Point3D
            Return New Point3D(p1.X - p2.X, p1.Y - p2.Y, p1.Z - p2.Z)
        End Operator

        Public Shared Operator *(ByVal multiplier As Double, ByVal p1 As Point3D) As Point3D
            Return New Point3D(p1.X * multiplier, p1.Y * multiplier, p1.Z * multiplier)
        End Operator

        Public Shared Operator *(ByVal p1 As Point3D, ByVal multiplier As Double) As Point3D
            Return New Point3D(p1.X * multiplier, p1.Y * multiplier, p1.Z * multiplier)
        End Operator

        Public Overrides Function ToString() As String
            Return String.Format("{0},{1},{2}", X, Y, Z)
        End Function
    End Structure
    Public ZMax As Double
    Public ZMin As Double
    Public Normal As Point3D
    Public P1, P2, P3 As Point3D

    Public Sub New(ByRef norm As Point3D, ByRef Point1 As Point3D, ByRef Point2 As Point3D, ByRef
Point3 As Point3D)
        Me.Normal = norm
        Me.P1 = Point1
        Me.P2 = Point2
        Me.P3 = Point3
        Me.ZMax = Math.Max(Point1.Z, Point2.Z)
        Me.ZMax = Math.Max(Me.ZMax, Point3.Z)
        Me.ZMin = Math.Min(Point1.Z, Point2.Z)
        Me.ZMin = Math.Min(Me.ZMin, Point3.Z)
    End Sub

    Public Overrides Function ToString() As String
        Return String.Format("[{0}] [{1}] [{2}]", P1, P2, P3)
    End Function

End Class
```

Pixel Line Class

```
Public Class PixelClass
   Structure VectorPixel
      Dim X, Y As UInteger

      Public Sub New(ByRef x As UInteger, ByRef y As UInteger)
         Me.X = x
         Me.Y = y
      End Sub

      Public Overrides Function ToString() As String
         Return String.Format("[{0}, {1}]", X, Y)
      End Function
   End Structure

   Public Po, Pf As VectorPixel
   Public Group As UInteger

   Public Sub New(ByRef P1 As VectorPixel, ByRef P2 As VectorPixel, ByRef id As UInteger)
      Me.Po = P1
      Me.Pf = P2
      Me.Group = id
   End Sub

   Public Shared Function Compare(ByRef P1 As VectorPixel, ByRef P2 As VectorPixel) As Boolean
      If P1.X = P2.X Then
         If P1.Y = P2.Y Then
            Return True
         Else
            Return False
         End If
      Else
         Return False
      End If
   End Function

   Public Overrides Function ToString() As String
      Return String.Format("[{0}, {1}] [{2}, {3}] [{4}]", Po.X, Po.Y, Pf.X, Pf.Y, Group)
   End Function

End Class
```

Main Code

```
Imports System
Imports System.IO
Imports System.ComponentModel
Imports System.Text
Imports DLP_3D_Printer.PixelClass
Imports DLP_3D_Printer.Facet

Public Class mainForm
    'STL Facet Read variables
    Dim FacetCount As UInteger = 0
    Dim groupID As UInteger = 0
    Dim exCount As UInteger = 0
    Dim zSlice As Double = 0
    Dim xMax, xMin, yMax, yMin, zMax, zMin As Double
    Dim STL_list As New List(Of Facet)
    Dim STL_intersect As New List(Of Facet)
    Dim facetBuffer(4) As Byte
    Dim header(80) As Byte
    Dim nx(4), ny(4), nz(4) As Byte
    Dim p1x(4), p1y(4), p1z(4) As Byte
    Dim p2x(4), p2y(4), p2z(4) As Byte
    Dim p3x(4), p3y(4), p3z(4) As Byte
    Dim atb(2) As Byte

    'Pixel Mapping variables
    Dim resW As UInteger = 1920     'Temp Screen Width X
    Dim resH As UInteger = 1080     'Temp Screen Height Y
    Dim ARxy, ARwh, ARz As Double
    Dim pixelList As List(Of PixelClass)
    Dim zOut As UInteger = 0

    Private Sub readBinary(ByVal fileSTL As String)
        Dim result As UInteger = 0
        Dim normal As New Point3D
        Dim p1 As New Point3D
        Dim p2 As New Point3D
        Dim p3 As New Point3D
        STL_list = New List(Of Facet)
        FacetCount = 0
        xMax = Double.MinValue
        yMax = Double.MinValue
        zMax = Double.MinValue
        xMin = Double.MaxValue
        yMin = Double.MaxValue
        zMin = Double.MaxValue

        Using myReader As New FileStream(fileSTL, FileMode.Open)
            myReader.Seek(0, SeekOrigin.Begin)
            Dim remains As Integer = CType(myReader.Length, Integer)
            Dim i As UInteger = 0

            If remains > 0 Then
                myReader.Read(header, 0, 80)
                myReader.Read(facetBuffer, 0, 4)
                exCount = BitConverter.ToInt32(facetBuffer, 0)
```

```
FacetCount = exCount

For k As UInteger = 0 To exCount - 1
  myReader.Read(nx, 0, 4)
  myReader.Read(ny, 0, 4)
  myReader.Read(nz, 0, 4)
  myReader.Read(p1x, 0, 4)
  myReader.Read(p1y, 0, 4)
  myReader.Read(p1z, 0, 4)
  myReader.Read(p2x, 0, 4)
  myReader.Read(p2y, 0, 4)
  myReader.Read(p2z, 0, 4)
  myReader.Read(p3x, 0, 4)
  myReader.Read(p3y, 0, 4)
  myReader.Read(p3z, 0, 4)
  myReader.Read(atb, 0, 2)

  normal.X = BitConverter.ToSingle(nx, 0)
  normal.Y = BitConverter.ToSingle(ny, 0)
  normal.Z = BitConverter.ToSingle(nz, 0)
  p1.X = BitConverter.ToSingle(p1x, 0)
  p1.Y = BitConverter.ToSingle(p1y, 0)
  p1.Z = BitConverter.ToSingle(p1z, 0)
  p2.X = BitConverter.ToSingle(p2x, 0)
  p2.Y = BitConverter.ToSingle(p2y, 0)
  p2.Z = BitConverter.ToSingle(p2z, 0)
  p3.X = BitConverter.ToSingle(p3x, 0)
  p3.Y = BitConverter.ToSingle(p3y, 0)
  p3.Z = BitConverter.ToSingle(p3z, 0)

  'Object X max/min
  xMax = Math.Max(xMax, p1.X)
  xMax = Math.Max(xMax, p2.X)
  xMax = Math.Max(xMax, p3.X)
  xMin = Math.Min(xMin, p1.X)
  xMin = Math.Min(xMin, p2.X)
  xMin = Math.Min(xMin, p3.X)

  'Object Y max/min
  yMax = Math.Max(yMax, p1.Y)
  yMax = Math.Max(yMax, p2.Y)
  yMax = Math.Max(yMax, p3.Y)
  yMin = Math.Min(yMin, p1.Y)
  yMin = Math.Min(yMin, p2.Y)
  yMin = Math.Min(yMin, p3.Y)

  'Object Z max/min
  zMax = Math.Max(zMax, p1.Z)
  zMax = Math.Max(zMax, p2.Z)
  zMax = Math.Max(zMax, p3.Z)
  zMin = Math.Min(zMin, p1.Z)
  zMin = Math.Min(zMin, p2.Z)
  zMin = Math.Min(zMin, p3.Z)

  STL_list.Add(New Facet(normal, p1, p2, p3))

Next
```

```
        End If
    End Using

    STL_list = STL_list.OrderBy(Function(x) x.ZMin).ToList()

End Sub

Private Sub Slice(ByVal sliceZ As Double)
    Dim si As Double
    Dim n, u, Po, Pf, Vo As Point3D

    initializeMatrix()
    STL_intersect = New List(Of Facet)
    STL_intersect = STL_list.FindAll(Function(x) x.ZMin < sliceZ And x.ZMax > sliceZ)

    n = New Point3D(0, 0, 1)
    Vo = New Point3D(0, 0, sliceZ)

    For Each facet_tri In STL_intersect
        Dim pFlag As Boolean() = {False, False, False}
        Dim pBuffer(3) As Point3D
        Dim vLength(3) As Double
        Dim pointCount As Byte = 0

        For k As Byte = 0 To 2
            Select Case k
                Case 0
                    Po = facet_tri.P1
                    Pf = facet_tri.P2
                Case 1
                    Po = facet_tri.P2
                    Pf = facet_tri.P3
                Case 2
                    Po = facet_tri.P3
                    Pf = facet_tri.P1
            End Select

            u = Pf - Po
            If Point3D.Dot(n, u) <> 0 Then  'If there is intersection
                si = Point3D.Dot(n, Vo - Po) / Point3D.Dot(n, u)
                If si >= 0 And si <= 1 Then
                    pBuffer(k) = Po + si * u
                    pFlag(k) = True
                    pointCount += 1
                End If
            End If
        Next

        'Case Handler
        Select Case pointCount
            Case 2
                If pFlag(0) And pFlag(1) Then
                    HashConvert(pBuffer(0), pBuffer(1))
                End If
                If pFlag(1) And pFlag(2) Then
                    HashConvert(pBuffer(1), pBuffer(2))
                End If
            If pFlag(2) And pFlag(0) Then
                HashConvert(pBuffer(2), pBuffer(0))
            End If
            Continue For
        Case 3
            'Find which pairs will produce longest vector
            vLength(0) = Point3D.LengthSq(pBuffer(0), pBuffer(1))
            vLength(1) = Point3D.LengthSq(pBuffer(1), pBuffer(2))
            vLength(2) = Point3D.LengthSq(pBuffer(2), pBuffer(0))

            If vLength(0) > vLength(1) Then
                If vLength(0) >= vLength(2) Then
                    HashConvert(pBuffer(0), pBuffer(1))
                Else
                    HashConvert(pBuffer(2), pBuffer(0))
                End If
            Else
                If vLength(1) >= vLength(2) Then
                    HashConvert(pBuffer(1), pBuffer(2))
                Else
                    HashConvert(pBuffer(2), pBuffer(0))
                End If
            End If
            Continue For
        Case Else
            Continue For
    End Select
Next

If pixelList.Count > 0 Then
    generateContour(pixelList)
End If
End Sub

Private Sub initializeMatrix()
    ARwh = resW / resH
    ARxy = (xMax - xMin) / (yMax - yMin)

    If ARxy >= ARwh Then
        ARz = (zMax - zMin) / (xMax - xMin)
        zOut = (zSlice - zMin) / (zMax - zMin) * (resW - 1) * ARz
    Else
        ARz = (zMax - zMin) / (yMax - yMin)
        zOut = (zSlice - zMin) / (zMax - zMin) * (resH - 1) * ARz
    End If
    pixelList = New List(Of PixelClass)
End Sub

Private Sub HashConvert(ByRef Point1 As Point3D, ByRef Point2 As Point3D)
    Dim po, pf As VectorPixel

    If ARxy >= ARwh Then
        'Fit to Width (X)
        po.X = (Point1.X - xMin) / (xMax - xMin) * (resW - 1)
        po.Y = (Point1.Y - yMin) / (yMax - yMin) * ((resW - 1) / ARxy)
        pf.X = (Point2.X - xMin) / (xMax - xMin) * (resW - 1)
        pf.Y = (Point2.Y - yMin) / (yMax - yMin) * ((resW - 1) / ARxy)
        Else
            'Fit to Height (Y)
            po.X = (Point1.X - xMin) / (xMax - xMin) * ((resH - 1) * ARxy)
            po.Y = (Point1.Y - yMin) / (yMax - yMin) * (resH - 1)
            pf.X = (Point2.X - xMin) / (xMax - xMin) * ((resH - 1) * ARxy)
            pf.Y = (Point2.Y - yMin) / (yMax - yMin) * (resH - 1)
        End If

        If Not Compare(po, pf) Then
            pixelList.Add(New PixelClass(po, pf, 0))
        End If
    End Sub


    Private Sub generateContour(ByRef list As List(Of PixelClass))
        Dim isInverse As Boolean = False
        Dim findInt As Integer = 0
        Dim initPoint As VectorPixel
        Dim searchPoint As VectorPixel

        groupID = 0
        list = list.OrderBy(Function(X) X.Po.Y).ToList
        list = list.OrderBy(Function(X) X.Po.X).ToList
        initPoint = list.Item(0).Po
        For i As Integer = 0 To list.Count - 2
            'Assign search point
            searchPoint = list.Item(i).Pf

            'Closed Loop check
            If Compare(searchPoint, initPoint) Then
                groupID += 1
                initPoint = list.Item(i + 1).Po
            Else
                'Find next pair
                findInt = FindPair(i, searchPoint, list)
                If findInt <> -1 Then
                    'Check if the point is inverted
                    isInverse = Compare(searchPoint, list.Item(findInt).Pf)
                    SwapPoint(findInt, i + 1, isInverse, list)
                End If
            End If
            list.Item(i + 1).Group = groupID
        Next

    End Sub

    Private Function FindPair(ByRef offset As UInteger, ByRef point As VectorPixel, ByRef list As List(Of
PixelClass)) As Integer
        For i As Integer = offset + 1 To list.Count - 1
            If Compare(point, list.Item(i).Po) Or Compare(point, list.Item(i).Pf) Then
                Return i
            End If
        Next
        Return -1
    End Function
    Private Sub SwapPoint(ByRef foundPoint As UInteger, ByVal nextPoint As UInteger, ByRef Inverse As
Boolean, ByRef list As List(Of PixelClass))
        Dim buffer As PixelClass = list.Item(foundPoint)
        list.Item(foundPoint) = list.Item(nextPoint)

        If Inverse Then
            list.Item(nextPoint) = New PixelClass(buffer.Pf, buffer.Po, 0)
        Else
            list.Item(nextPoint) = New PixelClass(buffer.Po, buffer.Pf, 0)
        End If
    End Sub

End Class
```

APPENDIX C
PSEUDOCODE (C++)

Point3D Class

```
#include "point3d.h"
#include <string>

using namespace std;

point3d::point3d(){}

point3d::point3d(float x, float y, float z):X(x),Y(y),Z(z){}

point3d::point3d(char* input){
    char x[4] = {input[0],input[1],input[2],input[3]};
    char y[4] = {input[4],input[5],input[6],input[7]};
    char z[4] = {input[8],input[9],input[10],input[11]};

    this->X = *((float*)x);
    this->Y = *((float*)y);
    this->Z = *((float*)z);
}

float point3d::dot(point3d Pa){
    return ((this->X * Pa.X) + (this->Y * Pa.Y) + (this->Z * Pa.Z));
}

point3d point3d::operator +(const point3d &Pa){
    return point3d(this->X+Pa.X, this->Y+Pa.Y, this->Z+Pa.Z);
}

point3d point3d::operator -(const point3d &Pa){
    return point3d(this->X-Pa.X, this->Y-Pa.Y, this->Z-Pa.Z);
}

point3d point3d::operator *(const float &mult){
    return point3d((this->X * mult), (this->Y * mult), (this->Z * mult));
}

string point3d::toString(){
    string buffer = "";
    buffer+=to_string(this->X); buffer += " ";
    buffer+=to_string(this->Y); buffer += " ";
    buffer+=to_string(this->Z); buffer += " ";
    return buffer;
}

point3d::~point3d(){}
```

Facet Class

```
#include "facet.h"
#include <math.h>

using namespace std;

facet::facet(){}

facet::facet(point3d p1, point3d p2, point3d p3, point3d norm):P1(p1),P2(p2), P3(p3), Norm(norm)
{
    this->Xmax = max(p1.X, p2.X); this->Xmax = max(this->Xmax, p3.X);
    this->Xmin = min(p1.X, p2.X); this->Xmin = min(this->Xmin, p3.X);
    this->Ymax = max(p1.Y, p2.Y); this->Ymax = max(this->Ymax, p3.Y);
    this->Ymin = min(p1.Y, p2.Y); this->Ymin = min(this->Ymin, p3.Y);
    this->Zmax = max(p1.Z, p2.Z); this->Zmax = max(this->Zmax, p3.Z);
    this->Zmin = min(p1.Z, p2.Z); this->Zmin = min(this->Zmin, p3.Z);
}

facet::facet(char *input){
    point3d gNorm(input);
    point3d gP1(input+12);
    point3d gP2(input+24);
    point3d gP3(input+36);

    this->Norm = gNorm;
    this->P1 = gP1;
    this->P2 = gP2;
    this->P3 = gP3;

    this->Xmax = max(gP1.X, gP2.X); this->Xmax = max(this->Xmax, gP3.X);
    this->Xmin = min(gP1.X, gP2.X); this->Xmin = min(this->Xmin, gP3.X);
    this->Ymax = max(gP1.Y, gP2.Y); this->Ymax = max(this->Ymax, gP3.Y);
    this->Ymin = min(gP1.Y, gP2.Y); this->Ymin = min(this->Ymin, gP3.Y);
    this->Zmax = max(gP1.Z, gP2.Z); this->Zmax = max(this->Zmax, gP3.Z);
    this->Zmin = min(gP1.Z, gP2.Z); this->Zmin = min(this->Zmin, gP3.Z);
}

string facet::toString(){
    string buffer="Facet\n";
    buffer+= this->P1.toString() + "\n";
    buffer+= this->P2.toString() + "\n";
    buffer+= this->P3.toString() + "\n";
    //buffer+= this->Norm.toString() + " ";
    buffer+= to_string(this->Zmax) + " ";
    buffer+= to_string(this->Zmin) + "\n";
    return buffer;
}

bool facet::operator<(const facet &other){
    return this->Zmin < other.Zmin;
}

bool facet::isIntersect(float &height){
    if((height < this->Zmax) && (height > this->Zmin)){
        return true;
    } else{
        return false;
    }
}

facet::~facet(){}
```

Point2D Class

```
#include "point2d.h"
#include "math.h"

point2d::point2d(){}

point2d::~point2d(){}

point2d::point2d(unsigned short &xx, unsigned short &yy):X(xx), Y(yy){}

string point2d::toString(){
    return "(" + to_string(this->X) + "," + to_string(this->Y) + ")";
}

float point2d::length(point2d &other){
    return (powf(this->X - other.X, 2) + powf(this->Y - other.Y, 2));
}
```

Pixel Line Class

```
#include "pixelline.h"

pixelLine::pixelLine(){}

pixelLine::pixelLine(const point2d a, const point2d b, const unsigned int id){
    this->Po = a;
    this->Pf = b;
    this->Id = id;
}

string pixelLine::toString(){
    string buffer="";
    buffer += this->Po.toString() + " ";
    buffer += this->Pf.toString() + " ";
    buffer += "[" + to_string(this->Id) + "]";
    return buffer;
}

pixelLine::~pixelLine(){}
```

Slicer Class

```
#include "slicer.h"
#include <iostream>
#include <fstream>
#include <algorithm>
#include <limits>

using namespace std;

Slicer::Slicer(){}

void Slicer::Initialize(){
   this->facetList.clear();
   this->facetCount = 0;
}

void Slicer::ReadSTL(string filename){
   ifstream stlFile;
   char fCount[4];
   char inputFacet[50];

   Initialize();

   stlFile.open(filename, ios::binary);

   if(stlFile.is_open()){
      stlFile.seekg(0);
      stlFile.ignore(80);
      stlFile.read(fCount, 4);
      this->facetCount = *((unsigned long*)fCount);

      for(unsigned long i = 0; i < this->facetCount; i++){
         stlFile.read(inputFacet, 50);
         facet ex(inputFacet);
         this->facetList.push_back(ex);
      };
   }

   stlFile.close();
   this->facetList.shrink_to_fit();
   sort(facetList.begin(),facetList.end());
   DefineBoundary();

}

void Slicer::Slice(float &height){
   this->LineList.clear();
   vector<facet*> intersectList;
   point2d *aPo, *aPf, pConvert[3];
   point3d n(0,0,1), u, *Po, *Pf, Vo(0,0,height);
   float si = 0;
   GenerateList(height, intersectList);

   for(unsigned int i = 0; i < intersectList.size(); i++){
      bool pFlag[3] = {false,false,false};
      point3d pBuffer[3];
```

```
float vLength[3];
unsigned short pointCount = 0;

for(unsigned short k = 0; k < 3; k++){
    switch(k){
    case 0:
        Po = &intersectList[i]->P1;
        Pf = &intersectList[i]->P2;
        break;
    case 1:
        Po = &intersectList[i]->P2;
        Pf = &intersectList[i]->P3;
        break;
    case 2:
        Po = &intersectList[i]->P3;
        Pf = &intersectList[i]->P1;
        break;
    };

    u = (*Pf) - (*Po);
    if(n.dot(u) != 0){
        si = n.dot(Vo - (*Po)) / n.dot(u);
        if((si >= 0) && (si <= 1)){
            pBuffer[k] = (*Po) + (u * si);
            pConvert[k] = Convert(pBuffer[k]);
            pFlag[k] = true;
            pointCount++;
        }
    };
};

switch(pointCount){
case 0:
case 1:
    break;
case 2:
    if(pFlag[0] && pFlag[1]){
        aPo = &pConvert[0]; aPf = &pConvert[1];
    }
    if(pFlag[1] && pFlag[2]){
        aPo = &pConvert[1]; aPf = &pConvert[2];
    }
    if(pFlag[2] && pFlag[0]){
        aPo = &pConvert[2]; aPf = &pConvert[0];
    }

    if(!Compare(*aPo, *aPf)){
        this->LineList.push_back(pixelLine(*aPo, *aPf, 0));
    }
    break;
case 3:
    vLength[0] = pConvert[0].length(pConvert[1]);
    vLength[1] = pConvert[1].length(pConvert[2]);
    vLength[2] = pConvert[2].length(pConvert[0]);

    if(vLength[0] > vLength[1]){
        if(vLength[0] >= vLength[2]){
                aPo = &pConvert[0]; aPf = &pConvert[1];
            }else {
                aPo = &pConvert[2]; aPf = &pConvert[0];
            }
        } else {
            if(vLength[1] >= vLength[2]){
                aPo = &pConvert[1]; aPf = &pConvert[2];
            } else {
                aPo = &pConvert[2]; aPf = &pConvert[0];
            }
        }

        if(!Compare(*aPo, *aPf)){
            this->LineList.push_back(pixelLine(*aPo, *aPf, 0));
        }
        break;
    };
};

if(LineList.size() != 0){
    Contour();
}
}

void Slicer::GenerateList(float &height, vector<facet*> &objectList){
    objectList.clear();
    for(unsigned int i = 0; i < this->facetList.size(); i++){
        if(this->facetList[i].isIntersect(height)){
            objectList.push_back(&facetList[i]);
        }
    }
}

void Slicer::DefineBoundary(){
    upperX = upperY = upperZ = numeric_limits<float>::lowest();
    lowerX = lowerY = lowerZ = numeric_limits<float>::max();

    for(unsigned int i = 0; i < this->facetList.size(); i++){
        upperX = max(upperX, this->facetList[i].Xmax);
        lowerX = min(lowerX, this->facetList[i].Xmin);
        upperY = max(upperY, this->facetList[i].Ymax);
        lowerY = min(lowerY, this->facetList[i].Ymin);
        upperZ = max(upperZ, this->facetList[i].Zmax);
        lowerZ = min(lowerZ, this->facetList[i].Zmin);
    }

    this->ARxy = (upperX - lowerX) / (upperY - lowerY);
}

void Slicer::SetResolution(const unsigned short width, const unsigned short height){
    this->resW = width;
    this->resH = height;
    this->ARwh = (float)width / (float)height;
}

point2d Slicer::Convert(point3d &Pa){
    unsigned short px, py;
    if(ARxy >= ARwh){
        px = (Pa.X - lowerX) / (upperX - lowerX) * (resW - 1);
        py = (Pa.Y - lowerY) / (upperY - lowerY) * ((resW - 1) / ARxy);
    } else {
        px = (Pa.X - lowerX) / (upperX - lowerX) * ((resH - 1) * ARxy);
        py = (Pa.Y - lowerY) / (upperY - lowerY) * (resH - 1);
    }

    return point2d(px,py);
}

bool Slicer::Compare(point2d &a, point2d &b){
    if(a.X == b.X){
        if(a.Y == b.Y){
            return true;
        }
    }
    return false;
}

void Slicer::Contour(){
    unsigned short id = 0;
    point2d *searchPoint, *initPoint;

    initPoint = &this->LineList[0].Po;
    for(unsigned int i = 0; i < this->LineList.size() - 1; i++){
        searchPoint = &this->LineList[i].Pf;

        if(Compare(*searchPoint, *initPoint)){
            id++;
            initPoint = &this->LineList[i + 1].Po;
        } else {
            int findInt = FindPair(i, *searchPoint);
            if(findInt != -1){
                bool isInverse = Compare(*searchPoint, this->LineList[findInt].Pf);
                SwapPoint(findInt, i+1, isInverse, this->LineList);
            } else {
                cout << "Point Not Found at: " << i << endl;
            }
        }
        this->LineList[i+1].Id = id;
    }
}

int Slicer::FindPair(unsigned int &startIndex, point2d &searchPoint){
    for(unsigned int i = startIndex + 1; i < this->LineList.size(); i++){
        if((Compare(searchPoint, this->LineList[i].Po)) || (Compare(searchPoint, this->LineList[i].Pf))){
            return i;
        }
    }
    return -1;
}

void Slicer::SwapPoint(const unsigned int foundPoint, const unsigned int nextPoint, bool inverse,
vector<pixelLine> &list){
    swap(list[foundPoint], list[nextPoint]);
    if(inverse){
        swap(list[nextPoint].Po, list[nextPoint].Pf);
    }
}

Slicer::~Slicer(){}
```

APPENDIX D
PSEUDOCODE (MATLAB)

MATLAB Script

```
%% CONTOUR PLOT
clear;
cd 'C:\Users\DELL\Documents\DLP Folder\Vector';
filename = 'Dragon';
myvars   = dir(sprintf('%s*.csv', filename));       %Get list of CSV file
disp(filename);
figure(1);
set(1, 'Name', sprintf('%s', filename),...          %Create new figure
   'Color', [1 1 1],'pos', [350 200 600 400]);

% Initialize Slice Height, Slice Time, and Contour Time variables
STime   = zeros(length(myvars), 1);
CTime   = zeros(length(myvars), 1);
HSlice  = zeros(length(myvars), 1);
IFacet  = zeros(length(myvars), 1);
GNumber = zeros(length(myvars), 1);
TTime   = zeros(length(myvars), 1);

for i = 1 : length(myvars)                         %Iterate for each file
   CurrentFile = csvread(myvars(i,1).name);        %Load working file
   Gmax = max(CurrentFile(:, 4));                  %Get Max Group number
   G = 0;                                          %Initialize G

   % Acquire Slice Height, Slice Time, and Contour Time
   HSlice(i)  = CurrentFile(1, 3);
   NFacet     = CurrentFile(1, 5);
   IFacet(i)  = CurrentFile(1, 6);
   STime(i)   = CurrentFile(1, 7);
   CTime(i)   = CurrentFile(1, 8);
   GNumber(i) = Gmax + 1;
   TTime(i) = STime(i) + CTime(i);

   while G <= Gmax
      % Filter Array based on Group Number at Column 4
      t = find(CurrentFile(:, 4) == G);

      % Get XYZ (Column 1 2 3)
      X = CurrentFile(t, 1);
      Y = CurrentFile(t, 2);
      Z = CurrentFile(t, 3);

      % Plot Contour for each layer with color mapping
      Ubound = 100;
      Lbound = Ubound / 2;
      if (TTime(i) > Ubound)
         yr = 255;
         yg = 0;
      end
      if (TTime(i) >= Lbound) && (TTime(i) <= Ubound)
         m = -255 / (Ubound - Lbound);
         c = 255 - Lbound * m;
```

```
            yr = 255;
            yg = round(m * TTime(i) + c);
        end
        if (TTime(i) < Lbound)
            yg = 255;
            yr = round(255 / Lbound * TTime(i));
        end
        colors_p = [yr, yg, 0] / 255;
        fill3(X, Y, Z, colors_p);
        G = G + 1;
        hold on;                              %Stack plotting
    end
end
a = [linspace(0, 1, 32); ones(1, 32); zeros(1, 32)]';
b = [ones(1, 32); linspace(1, 0, 32); zeros(1, 32)]';
c = [a; b];
colormap(c);
val = linspace(0, Ubound, 11);
colorbar('YTickLabel', val);
axis equal;
set(gca, 'Color', [0.9 0.9 0.9]);                  %Set grid BG color
set(gca, 'View', [45 30]);                        %Set rotation axis
grid on;                                      %Enable grid
hold off;                                     %Disable stack plot
saveas(gcf,sprintf('%s_1-Fig.png',filename));      %Save figure

%% RESULT GRAPH PLOTS
% Plot Slice Time
figure(2);
set(2,'Name','Slice Time vs Slice Height','pos',[350 200 500 200]); %edited for single plot
bar(HSlice, STime, 1, 'FaceColor', barColor, 'EdgeColor', 'k');
xlabel('Slice Height');
ylabel('Slice Time (ms)');
xlim([min(HSlice) max(HSlice)]);
title(sprintf('Facet = %d, Mean = %0.2fms, SD = %0.2f', NFacet, mean(STime), std(STime))); %edited for
single plot
grid on;
saveas(gcf,sprintf('%s_2-ST.png',filename));      %Save figure

% Plot Contour Time
figure(3);
set(3,'Name','Contour Time vs Slice Height','pos',[350 200 500 200]); %edited for single plot
bar(HSlice, CTime, 1, 'FaceColor', barColor, 'EdgeColor', 'k');
xlabel('Slice Height');
ylabel('Contour Time (ms)');
xlim([min(HSlice) max(HSlice)]);
title(sprintf('Facet = %d, Mean = %0.2fms, SD = %0.2f', NFacet, mean(CTime), std(CTime)))
grid on;
saveas(gcf,sprintf('%s_3-CT.png',filename));      %Save figure

% Plot Total Time
figure(4);
set(4,'Name','Total Time vs Slice Height','pos',[350 200 500 200]); %edited for single plot
bar(HSlice, TTime, 1, 'FaceColor', barColor, 'EdgeColor', 'k');
xlabel('Slice Height');
ylabel('Total Time (ms)');
xlim([min(HSlice) max(HSlice)]);
```

```
title(sprintf('Facet = %d, Mean = %0.2fms, SD = %0.2f', NFacet, mean(TTime), std(TTime)))
grid on;
saveas(gcf,sprintf('%s_4-TT.png',filename));      %Save figure

% Plot Intersecting Facet
figure(5);
set(5,'Name','Intersecting Facet vs Slice Height','pos',[350 200 500 200]); %edited for single plot
bar(HSlice, IFacet, 1, 'FaceColor', barColor, 'EdgeColor', 'k');
xlabel('Slice Height');
ylabel('Intersecting Facet');
xlim([min(HSlice) max(HSlice)]);
grid on;
saveas(gcf,sprintf('%s_5-IF.png',filename));      %Save figure

% Plot Loop Count
figure(6);
set(6,'Name','Loop Number vs Slice Height','pos',[350 200 500 200]); %edited for single plot
bar(HSlice, GNumber, 1, 'FaceColor', barColor, 'EdgeColor', 'k');
xlabel('Slice Height');
ylabel('Loop Count');
xlim([min(HSlice) max(HSlice)]);
grid on;
saveas(gcf,sprintf('%s_6-LC.png',filename));      %Save figure

%% CALL FUNCTION
[it, rw] = max(CTime);
zs = HSlice(rw, 1);
funcContour(zs, filename);

disp('Slice Time VS Loop Count');
disp(NCorr(STime,GNumber));
disp('Slice Time VS Intersecting Facet');
disp(NCorr(STime,IFacet));
disp('Contour Time VS Loop Count');
disp(NCorr(CTime,GNumber));
disp('Contour Time VS Intersecting Facet');
disp(NCorr(CTime,IFacet));
```

```
function funcContour(zSlice, filename)
   myvars = dir(sprintf('%s*.csv', filename));         %Get list of CSV file
   figure(8);
   set(8, 'Name', sprintf('Contour Time (%s)',...      %Create new figure
      filename), 'Color', [1 1 1],'pos', [350 200 600 400]);

   for i = 1 : length(myvars)                          %Iterate for each file
      CurrentFile = csvread(myvars(i,1).name);         %Load working file
      Gmax = max(CurrentFile(:, 4));                   %Get Max Group number
      G = 0;                                           %Initialize G

      if zSlice == CurrentFile(1, 3)
         CTprev   = CurrentFile(1, 8);
         CTheight = CurrentFile(1, 3);
         CIfacet  = CurrentFile(1, 6);
         GHigh    = Gmax + 1;
         hold off;
         while G <= Gmax
            % Filter Array based on Group Number at Column 4
            t = find(CurrentFile(:, 4) == G);

            % Get XYZ (Column 1 2 3)
            X = CurrentFile(t, 1);
            Y = CurrentFile(t, 2);

            % Plot Contour
            plot(X, Y, 'k', 'LineWidth', 1);
            hold on;
            G = G + 1;
         end
         break;
      end
   end
   title(sprintf('C.Time = %0.2fms,  Z = %d,  Loop = %d,  Facet = %d', CTprev, CTheight,...
      GHigh, CIfacet));
   axis equal;
   grid on;
   xlabel('X-axis');
   ylabel('Y-axis');
   saveas(gcf,sprintf('%s_8-HCT.png',filename));      %Save figure
end
```