

Generating Vectors for Pop-Up Cards from Three-Dimensional Models

by
Or Oppenheimer

S.B., Electrical Engineering and Computer Science
S.B., Mechanical Engineering
Massachusetts Institute of Technology, 2018

Submitted to the
Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© 2019 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature of Author: _____
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by: _____
David R. Wallace, Professor of Mechanical Engineering
May 24, 2019

Accepted by: _____
Katrina LaCurtis, Chair, Master of Engineering Thesis Committee

Generating Vectors for Pop-Up Cards from Three-Dimensional Models

by

Or Oppenheimer

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in Partial Fulfillment of the
Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

Pop-up cards are greeting cards that have a three-dimensional representation of an object that, when the card is opened, stands up on its own. Grid-based pop-up cards are the most difficult to design manually due to their complexity and number of pieces that must fit together accurately. In order to simplify the pop-up design and fabrication process, this paper presents a computational system to slice STL models and create an SVG file that can be used to laser cut the pieces required to make a grid-based pop-up model. The computational system inputs a 3D model, slices it, and layers slits on top of these slices to ensure that the result will fit together, creating an easily assembled pop-up card. Initial testing shows that the computational model will successfully slice and layer slits on models made up of fewer triangles but tends to have issues with larger, more complex models. The main area of improvement for the computational model is with slicing time, which can be improved through the utilization of existing slicing algorithms mainly used for 3D printing.

Thesis Supervisor: David R. Wallace

Title: Professor of Mechanical Engineering

Acknowledgments

I would like to thank David Wallace for an amazing year and giving me the opportunity to build upon skills I've always loved but never had the chance to explore further. From resizing Build Challenge balls with David, to making a tear-away suit with Georgia, to learning to cast with Josh, to making safety glasses holders with Lauren, to choosing pictures with Lenny, to shooting a lightning storm with Ryan, every day was a new adventure and a fun challenge to tackle together. Thank you to my fellow TAs (official and unofficial, also some of you were instructors) for teaching me so much. Working and joking with all of you was the highlight of my year.



Figure 0: Representation of the stickers used to signify a member of CADLab's contribution. Since I cannot actually put a sticker on my thesis, here is a figure instead.

Thank you to my parents and siblings for always supporting me no matter what craziness I had chosen for that semester. Even though you were far away, I always had someone to talk to.

Thanks to my roommates (#HQties #WordNerds) for supporting me despite the fact that I was never around. Coming back at 3am was a lot less painful with a pile of cookies waiting for me.

Table of Contents

| | |
|---|----|
| Abstract..... | 3 |
| Acknowledgements..... | 5 |
| Table of Contents..... | 7 |
| List of Figures..... | 9 |
| 1. Introduction..... | 11 |
| 2. Background..... | 12 |
| 2.1 Types of Pop-Up Cards..... | 12 |
| 2.2 STL Files..... | 15 |
| 2.3 Vector Graphics and SVG Files..... | 15 |
| 2.3.1 Polylines in SVGs..... | 16 |
| 2.3.2 Groups in SVGs..... | 16 |
| 3. How to Make a Grid-Based Pop-Up Card Manually..... | 16 |
| 3.1 Slices..... | 16 |
| 3.2 Connecting the Slices..... | 17 |
| 3.3 Card Attachment..... | 19 |
| 4. Implementation..... | 20 |
| 4.1 User Provided Parameters..... | 20 |
| 4.2 Software Architecture Overview..... | 22 |
| 4.3 Slicing the Model..... | 23 |
| 4.3.1 Triangle Plane Intersection..... | 24 |
| 4.3.2 Generate Maps..... | 26 |
| 4.3.3 Slicing the Model..... | 27 |
| 4.3.4 Create Contours..... | 28 |
| 4.4 Adding Slits..... | 30 |
| 4.4.1 While Creating Contours..... | 31 |
| 4.4.2 Adding to Contours..... | 32 |

| | |
|---|----|
| 4.4.3 SVG Overlay..... | 32 |
| 4.5 Combining Contours and Slits for Exporting | 36 |
| 5. Results and Discussion | 37 |
| 6. Conclusion and Future Work | 39 |
| References..... | 41 |
| A Code | 42 |
| A.1 Triangle Plane Intersection | 42 |
| A.2 RunTranslation PopUp..... | 42 |
| A.3 Accelerated Slicing Parallel to the X- and Y-Axis | 45 |
| A.4 CreateContour | 47 |
| A.5 AddToContour | 47 |
| A.6 Slit | 49 |
| A.7 GenerateMaps | 50 |

List of Figures

| | | |
|-------------------|---|----|
| Figure 0: | Representation of the stickers used to signify a member of CADLab's contribution. | 5 |
| Figure 1: | Example of the Reversed Crease pop-up card. | 13 |
| Figure 2: | Example of the Symmetric Stand Up pop-up card. | 13 |
| Figure 3: | Example of the Grid-Based pop-up card | 14 |
| Figure 4: | Manually designing slices for a Bulbasaur pop-up card. | 17 |
| Figure 5: | Manually designed slices along one axis. | 18 |
| Figure 6: | Design for attaching a pop-up model to a card. | 19 |
| Figure 7: | Process for attaching a pop-up object to a card. | 20 |
| Figure 8: | Coordinate system defined for grid-based pop-up card model. | 21 |
| Figure 9: | Slit line height shown on a single slice as the ratio of the total height. | 22 |
| Figure 10: | Slices along the x - and y -axis of the Stanford Bunny model rendered in MeshLab. | 24 |
| Figure 11: | The intersection between a triangle and a plane. | 25 |
| Figure 12: | The intersection of a plane and one edge of a triangle. | 26 |
| Figure 13: | Placing triangles into buckets to create interval tree. | 27 |
| Figure 14: | A slice of the Stanford Bunny model that results in two contours. | 29 |
| Figure 15: | Slices parallel to the x - and y -axis viewed from both the x and y directions rendered in MeshLab. | 30 |
| Figure 16: | Sorting vertices in a single contour along the horizontal axis. | 31 |
| Figure 17: | Inserting points in a contour to create a slit. | 32 |
| Figure 18: | Multiple slits inserted in the same location on the contour. | 33 |
| Figure 19: | Result of inserting points into contours to create slits using the Stanford Bunny model. | 34 |
| Figure 20: | Varying slit patterns for the same slice of a manually designed Bulbasaur. | 35 |
| Figure 21: | Result of overlaying slits on top of a single slice of the Stanford Bunny model. | 36 |
| Figure 22: | Resulting SVG for slices parallel to the x -axis with slits overlaid on the slices. | 37 |
| Figure 23: | Resulting SVG for slices parallel to the y -axis with slits overlaid on the slices. | 37 |
| Figure 24: | A fully made pop-up card of the Stanford Bunny model as created by the system. | 38 |

- Figure 25:** Edits made to a slice of the Stanford Bunny model after the system generated the SVG. 38
- Figure 26:** A fully made pop-up card of the Stanford Bunny model with small edits. 39

1. Introduction

With the rise of maker spaces and the popularity of Do It Yourself (DIY) projects, crafts have become more accessible for a variety of project enthusiasts from children to adults. In fact, in recent years, the arts and crafts industry has grown, as adults want to make more on their own rather than buying things in stores [1]. The popular crafts supply store Michael's has seen an increase in business due to the DIY movement, and claims that sites such as Pinterest and Youtube have helped the business grow, since crafting is now more accessible [1]. Crafting itself has become a business with sites such as Etsy that provide a space where hobbyists can sell their creations as a supplement to their job, making crafting more affordable [1].

There are many different types of crafts that consumers enjoy, but paper crafting and scrapbooking is the most popular category, as stated by the Craft and Hobby Association [2]. In the United States alone, 7 billion greeting cards are purchased every year, and with the popularity of paper crafting and DIY, some crafters like to make their greeting cards by hand to add a personal touch to their cards [3]. Many crafters even choose to sell their handmade greeting cards to allow those with less time or skill the chance to have more unique cards [3].

The rise of maker spaces allows crafters access to more tools without having to purchase them, which allows for more complex crafts. Makerspaces are places that have larger, more expensive tools such as 3D printers and laser cutters. Crafters generally pay a fee to get access to these makerspaces, where they can create more complex personal projects with the use of equipment they would usually not be able to purchase or maintain on their own. Between 2006 and 2016 the number of makerspaces worldwide increased 14 times [4]. These spaces also provide a community where people share ideas and help each other execute their vision [4].

Despite having access to more physical equipment, people still have trouble designing personal projects. Designing grid-based pop-up cards manually requires an understanding of how pieces fit together in three-dimensional space as well as being able to picture this throughout the design process. It is also difficult to manually design grid-based pop-up cards that are not symmetric, as this reduces the number of slices that need to be designed. The tool presented here allows the user to mostly bypass this design process and helps with the fabrication of more complex cards, with the assumption that the user has access to a makerspace with a laser cutter. This project aims to create a tool that would assist paper crafters with access to a laser cutter, to easily design a pop-up card from a three-dimensional model. A computational system was developed to explore converting a 3D geometry to a paper craft pattern. After inputting an STL file of the 3D model to be sliced and a few parameters such as the grid spacing and material thickness, the user then receives an SVG file containing all of the pieces for the pop-up card to scale. The SVG file can then be easily laser cut and assembled to create the pop up card. With this tool, complex pop-up cards can be accessible to any crafters with access to a laser cutter.

2. Background

2.1 Types of Pop-Up Cards

Pop-up cards are greeting cards that when opened have some sort of three-dimensional object inside. For the card to work properly, it must fold flat to fit in an envelope and stand up when opened without additional user input. This paper categorizes pop-up cards into three groups: the Reversed Crease, the Symmetric Stand Up, and the Grid-Based Card.

The first type of card is the Reversed Crease. This card is made of two layers of paper, the external and the internal layer. Both layers are folded to form a card, and aligned along the

centerfold. A shape or word is cut out of the inner layer and the crease is reversed in the area of the design. Finally, the two layers are attached everywhere except where the design is located. Now when the card is folded, the design folds in the opposite direction and stands up when the card is open. This type of card is shown in Figure 1.



Figure 1: Example of the Reversed Crease pop-up card [5]. The internal layer of the card has the word “BOO” cut out of it, and the reversed centerfold along the word, making the word appear to be standing up vertically when the card is opened.

The second type of card works for any design that is symmetric along a single axis. The design is cut out of paper twice to form the two sides. The tops of the cutouts are glued together and the bottoms are glued to the card on either side of the centerfold, as shown in Figure 2.



Figure 2: Example of the Symmetric Stand Up pop-up card [6]. The two sides of the pop-up are symmetric. The tops are glued together and the bottoms are glued to the card on either side of the crease.

The third type of pop-up card relies on a grid system as a way to model some object in 3D. The object is sliced vertically in two perpendicular directions. Slits are then cut into these slices so that they fit together into a three-dimensional representation of the object. As long as the slices are equally spaced along both directions, the model will be able to fold flat and fit into a closed card. When attaching the model to the card, the diagonal line from one corner of grid to the opposite corner must line up with the centerfold of the card. This ensures that when the card is folded, the object will be folded along the same axis. Figure 3 shows an example of this type of card.

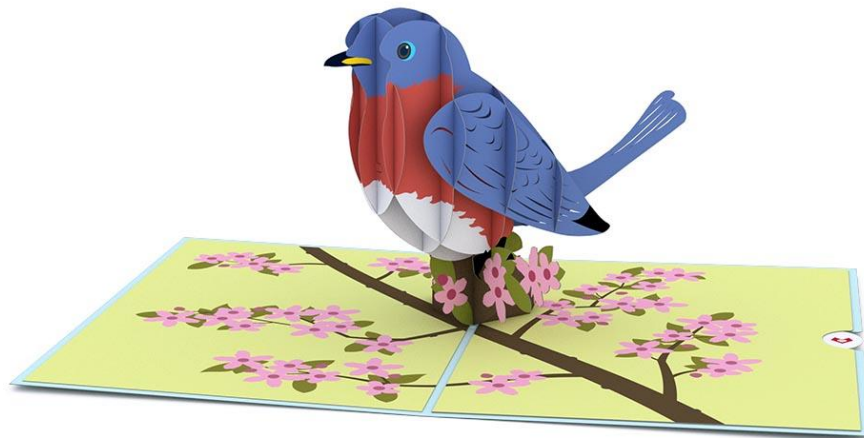


Figure 3: Example of the Grid-Based pop-up card [7]. The bird is made up of a grid of paper. Slits are cut into the paper such that they fit together. The diagonal along of the grid from the front of the bird to the back of the bird is lined up with the centerfold of the card. The bird is sitting on a stand, made to look like part of the branch, that is the attachment point of the bird to the card.

Compared to the Reversed Crease and Symmetric Stand Up cards, the Grid-Based card is more difficult to design manually. It requires the designer to think about how each slice will fit together as well as being able to visualize the final model throughout the design process. The Grid-Based card is also more complex because there are more pieces and they have to fit together in a structural way that can also collapse. For these reasons, the tool described in this paper assists in designing and fabricating grid-based pop-up cards by slicing 3D models and adding slits.

2.2 STL Files

An STL is a representation of the surface of a three-dimensional model that has become an industry standard for rapid prototyping. This file format stores the information for the surface of a model as triangles [8]. Triangles are used as a way to approximate the true shape of the model. The smaller the triangles, the more accurately curved sides will be rendered [8].

2.3 Vector Graphics and SVG Files

Vector graphics are made up of paths that allow them to be scaled without losing any information. These paths are made up of points with lines connecting them that can be either straight or curved[9]. Because the paths store all of the information that makes up the graphic and no specific number of pixels makes up the image, vector graphics can be scaled while maintaining image quality.

The paths of a vector graphic can also be passed into CNC machines such as laser cutters or CNC routers. The machines can interpret the paths and cut along them, allowing the graphic to be transferred into the physical materials.

In this paper, the pieces of the pop-up card are created as a Scalable Vector Graphic (SVG), which allows the pop-up design to be used in a laser cutter. While SVG files are generally used for the web because they allow for easily scalable two-dimensional graphics, they are a generic file type that can also be rendered using vector software such as Adobe Illustrator. SVG images are written as their own coding language, similar to HTML, so they are well suited for use as a way to output the final pop-up slices [10].

2.3.1 Polylines in SVGs

Polylines are one of the ways you can create a shape in an SVG file. They are made up of a list of x - and y -coordinates that are connected by straight lines. Even though polylines are made of straight line segments, if enough points are used to generate the polyline, the final output in the SVG file will appear as a smooth curve [11].

2.3.2 Groups in SVGs

Although each polyline is a single object in an SVG, it is sometimes helpful to group multiple polylines together so that their location and size remain constant relative to each other. To do this, SVGs have a group attribute. If multiple polylines are created inside of a group, then when the SVG is opened in an editor such as Illustrator, all of the polylines will be kept constant relative to each other. This is used to ensure that the relative size and shape of all of the slices are constant so that the pop-up object will always fit together properly, even if the user makes changes after the SVG has been generated. This allows the user to adjust the output of the computational system if additional changes are needed for a particular design.

3. How to Make a Grid-Based Pop-Up Card Manually

There are three main steps to creating a grid-based pop-up card. The first is creating the slices that make up the object, the second is adding in the slits that will allow the slices to be connected, and the third is to attach the object to the card in a way that allows the object to stand up properly when the card is open, but also fold when the card is closed.

3.1 Slices

The slices are cut in two perpendicular directions with equal spacing. The spacing between the slices has to be equal in both directions for the model to successfully collapse, which is

required for the object to fit into a card. The slices are each shaped differently according to the profile defined by the cross section of the model at that slice, and when the slices are all lined up, the object becomes recognizable. As more slices are used, the pop-up will represent the object more clearly and with higher fidelity.

To create the slices by hand, a front, side, and back view of the object is required. It is easier to work with an object that is symmetric; only one side view is needed to fully understand the object. The first step is determining where the slices will be located, making sure they are evenly spaced along both axes. Next, the image is used to design the shape of the slice, ensuring that the slices along the different axes meet up at the correct locations, as shown in Figure 4.

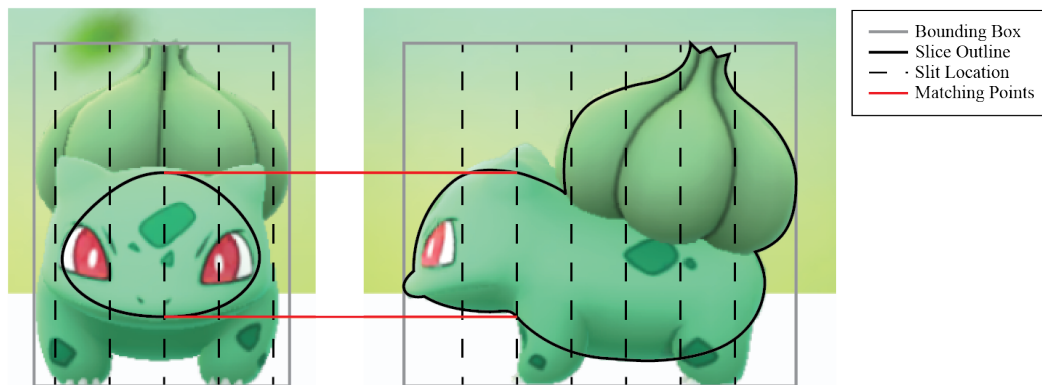


Figure 4: Manually designing slices for a Bulbasaur pop-up card. The slice from the front view has to intersect the side view slice at the points where they meet to ensure that the final model will align correctly along both axes.

3.2 *Connecting the Slices*

Now that there are two sets of slices that are equally spaced, connection points need to be designed. This is done by adding vertical slits to each slice. Slits are thin cuts in each slice that are the width of the material thickness. For 80lb cardstock, the type of paper used for the examples in this paper, slits need to be 0.01 inches wide.

Slits are added to the slices at the locations where the slices intersect. A slit height is chosen based on the shape of the model. Improper slit height can lead to parts of the model being cut off,

so it is important to choose a slit height that will allow for all of the important details of the object to be included. Some sections with unnecessary detail may need to be removed to ensure more important features are realized in other slices.

Finally, the slits are cut into each slice. The starting location of each slit alternates between the top and bottom of that particular slice. In addition to alternating the starting location of the slits within a slice, the first slit from left to right will alternate starting on the top or bottom of the slice depending on where the first slit of the previous slice was made. Alternating slit placement is shown in Figure 5. This alternation between slices ensures that the final model is structural and will not fall apart. Figure 5 shows the slices and slits along one axis of a Bulbasaur pop-up.

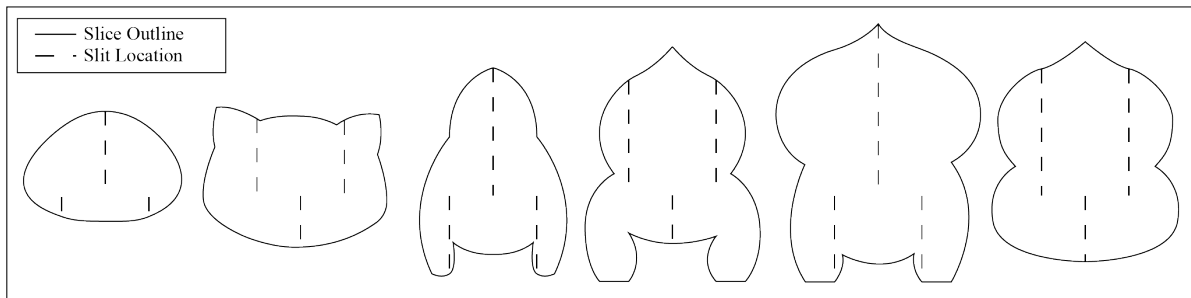


Figure 5: Manually designed slices along one axis. Note that the slit patterns alternate between each slice. The first slice has two slits starting from the bottom, the next has two starting from the top, and so on. This is also done for slices along the other axis to ensure that the entire grid pattern will alternate, keeping the model from falling apart.

After the slits have been cut into the slices, the slices are woven together. This requires some twisting and maneuvering of the slices because the slit pattern creates a woven pattern for structural stability. However, as long as the slices are carefully curved to fit into place as opposed to folded, they will go back to their original shape without showing any evidence of the process.

3.3 Card Attachment

To attach the model to the card, two small circles are cut out of paper and each folded each in half. A slit is cut halfway through the circle along the crease, and a quarter of the circle is folded back as shown in the Figure 7.

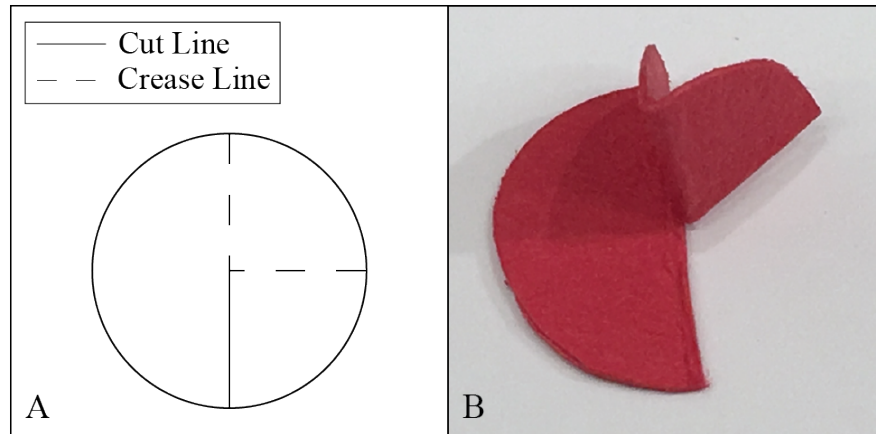


Figure 6: Design for attaching a pop-up model to a card.

To attach the pop-up model to the rest of the card, the folded quarter of each circle is glued to opposite corners of the object. The card is opened so that it lays flat and the pop-up model is placed in the center. The diagonal of the grid is aligned with the centerfold of the card. One of the circles is glued to the card while the model is in the open position. The model is collapsed down so it is lying flat on the side of the card that it is already connected to. Glue is placed on the other circle, and the other half of the card is folded down on top of the model. Following this process will make certain that the diagonal of the model stays aligned with the centerfold of the card, ensuring that it will open and close properly. The process of attaching a grid-based object to a card is demonstrated in Figure 7.

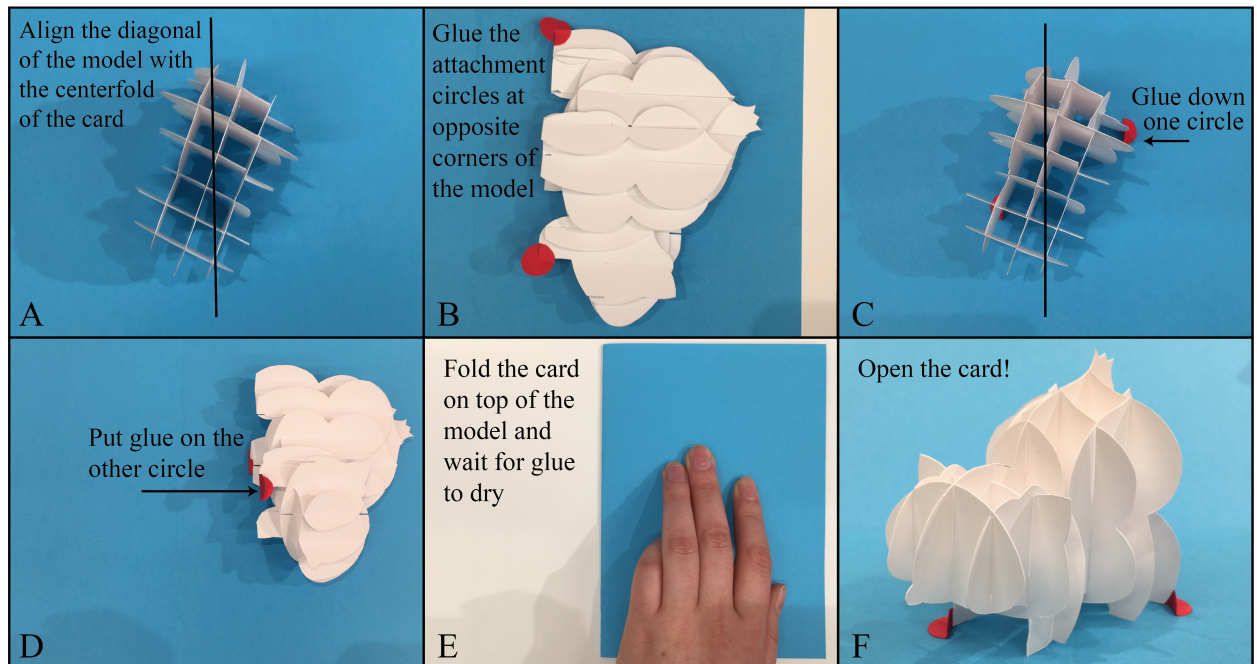


Figure 7: Design for attaching a pop-up object to a card. A) Align the diagonal of the grid-based object with the centerfold of the card. B) Glue the attachment circles to the object, making sure that only one quarter of the circle is attached to the object. C) Glue down one of the circles to the card. D) Fold the object onto the side that is glued down. Put glue on the other circle. E) Fold the card down on top of the object and let the glue on the second circle dry. F) Once the glue has dried, the card can be opened and the grid-based object will stand up and collapse correctly.

4. Implementation

The computational system for aiding in the design of grid-based pop-up cards can be broken down into three parts: slicing the 3D model, inserting the slits into the slices, and exporting the slices as SVGs that can be cut on a laser cutter.

The code was written in C++, an efficient and convenient language for doing calculations involving vectors and other structures often used in computer graphics. All methods discussed in this section can be found in the Appendix.

4.1 User Provided Parameters

The user begins by passing in the STL file that will be turned into the pop-up card. The user also determines how many slices they would like along the x -axis of the model. Since the spacing of the slices along the x - and y -axis must be the same for the card to fold, providing one

input is enough to define the pop-up model. The coordinate system defined for grid-based pop-up cards is shown in Figure 8.

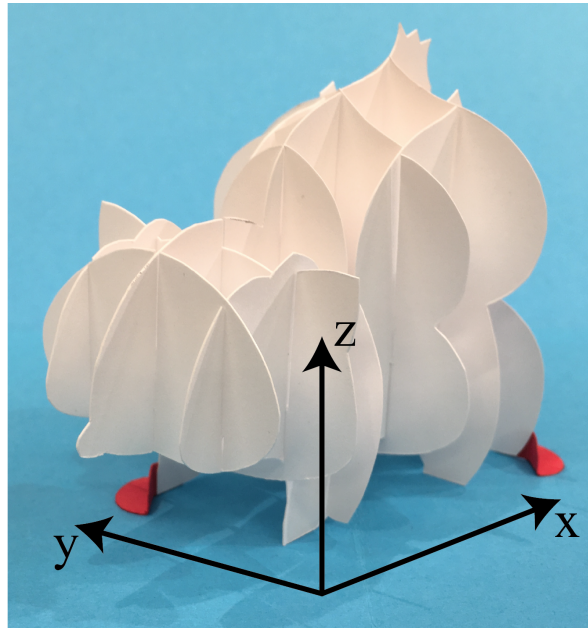


Figure 8: Coordinate system defined for grid-base pop-up card model.

The user also provides the thickness of the material in order for the system to determine the proper slit width. The user also chooses a value between 0 and 1 that represents the fraction of the total height of the model that the slits should go up to. For example, if the model is 5 inches tall, and the user chooses a `slit_line_height` of 0.2, then the slits starting from the bottom would only have a length of 1 inch, and those coming from the top would have a length of 4 inches, as shown in Figure 9. Adjusting the slit height can help mitigate slit placement errors that might cut off sections of the model, discussed later on in this paper.

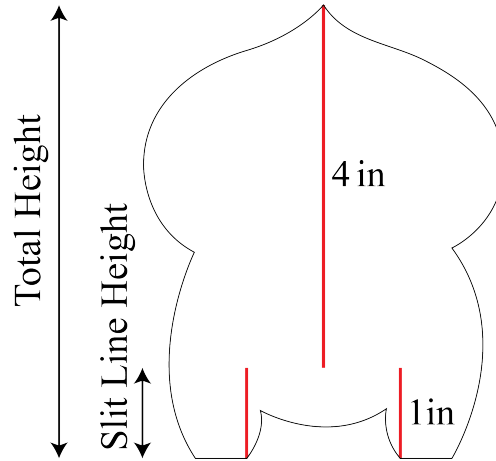


Figure 9: Slit line height shown on a single slice as the ratio of the total height.

4.2 Software Architecture Overview

The `main` method starts by loading in the data from the user-provided STL file. The STL file is then converted into a triangle mesh, which stores the entire 3D object as a series of triangles. This structure allows for easier access to the data for future use; however, it runs in $O(n)$ time and can therefore take a long time if the STL is large. The next step is to find the maximum and minimum values of all the points in the triangle mesh to be used to create the bounding box of the model. The bounding box is used throughout the system to optimize algorithms as well as to correctly place slits. Then the `GenerateMaps` function creates interval trees along the x - and y -axis to make the overall calculations run faster.

The `main` method then calls `RunTranslation_PopUp`, which runs the remaining computations. It starts by finding the intersections between all of the slice planes and the model. This gives a list of edges along each plane that is passed into `CreateContour`, which stitches the edges together to form closed loops, referred to as contours.

Lastly, an SVG file is generated and each contour is translated into the SVG. The slits that line up with the slices going in the perpendicular direction to the contour are also added to the contour so that the final SVG will have each contour and its corresponding slits.

4.3 Slicing the Model

A set of slices parallel to the x -axis and a set of slices parallel to the y -axis need to be generated. The spacing of the slices is a user given input determined by the number of slices they desire along the x -axis. Using the minimum and maximum values of all the points in the STL file, the distance dx between the slices can be calculated using Equation 1.

$$dx = (\max_x - \min_x) / (\text{num_across_x} + 1) \quad (1)$$

Where \max_x is the maximum x value of any point in the model, \min_x is the minimum x value of any point in the model, and num_across_x is the number of slices along the x -axis that the user specified. Note that $\text{num_across_x} + 1$ is the number of portions along the x -axis that the model is divided into so that the final number of slices will be num_across_x .

In order for the pop-up model to successfully collapse, the spacing between the slices along the x -axis must be the same as the spacing along the y -axis, therefore

$$dy = dx = (\max_x - \min_x) / (\text{num_across_x} + 1) \quad (2)$$

Using Equations 1 and 2, the exact locations along the x - and y -axis where the slices should be generated can be found. Each of these slices is a plane, and the points of intersection between the 3D model and each plane must be found next. Figure 10 shows the resulting slices along the x - and y -axis for the Stanford Bunny model.

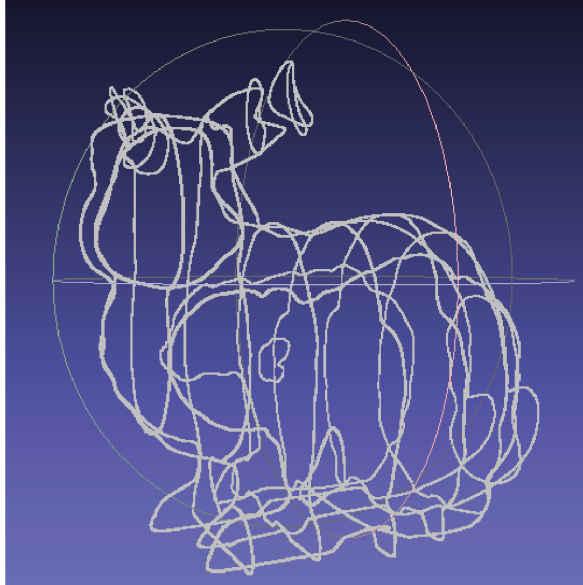


Figure 10: Slices along the x - and y -axis of the Stanford Bunny model rendered in MeshLab [12].

4.3.1 Triangle Plane Intersection

The initial slicing reduced the model to a series of triangles, each defined by three points in three-dimensional space. The points that make up the contour of the slice of the pop-up card are the locations where the plane of the slice and the triangles intersect. To find how a plane intersects with the overall model, the first step is to calculate the intersection points of a plane with a single triangle.

The intersection between a triangle and a plane is a line segment with a start and an end point, referred to as an edge. If the triangle does intersect the plane, then an edge is stored in the system represented by the two points along the outside of the triangle that define the line where the triangle and plane intersect. This edge is the part of the 3D model that lies on the plane and will be part of the slice, as shown in Figure 11.

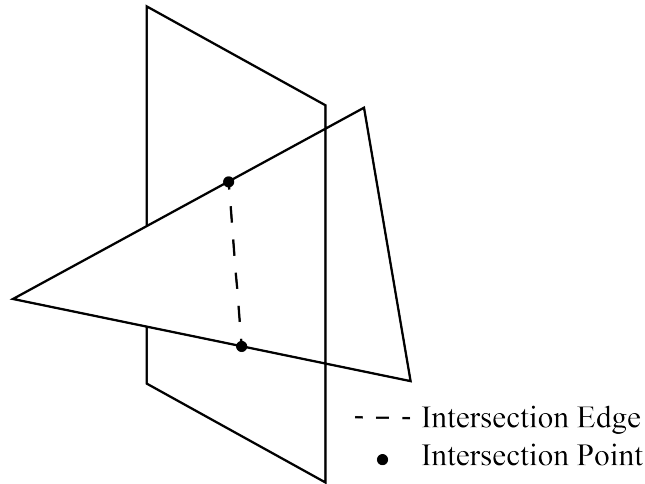


Figure 11: The intersection between a triangle and a plane. The intersection is a line segment where the endpoints lie on the edges of the triangle.

To find if a triangle intersects with a plane, first check if any vertices of the triangle lie on the plane. If any of them do, then add those vertices to the edge because they intersect with the plane, and since they are the endpoints of the triangle, they must also be the endpoint of the edge that lies on the triangle.

Next, each pair of vertices needs to be checked to see if they lie on opposite sides of the plane. If they do, then there is a point between them that lies on the plane. Because the line between the two vertices is one of the edges of the triangle, the point that lies on the plane and this line must be the endpoint of the intersection edge between the triangle and the plane, and can therefore be added to the intersecting edge. Given a triangle with vertices A , B , and C , the point Q that lies along AB and intersects with the plane P can be found using Equation 3.

$$Q = A + AB\left(\frac{Qa}{Qa + Qb}\right) \quad (3)$$

In the equation above, Qa and Qb are the distance from the point Q that lies on the plane to projection of vertices A and B to be perpendicular to the plane P . Equation 3 is shown geometrically in Figure 12.

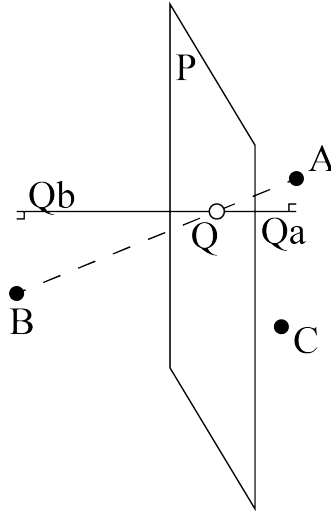


Figure 12: The intersection of a plane and one edge of a triangle. The point Q lies on the plane P and the edge AB . The lengths Qa and Qb are the perpendicular distances between point Q and points A and B , respectively.

4.3.2 Generate Maps

One way to find the intersecting edges on every slice is to intersect each triangle that makes up the 3D model with each plane that defines a slice. However, this is time consuming and unnecessary because most triangles will not intersect with all of the planes. **GenerateMaps** creates an interval tree along the x - and y -axis to organize the triangles.

An interval tree is a way of bucketing the triangles into groups that have the possibility of intersecting the different planes. The system starts by creating buckets along the x - and y -axis. The total number of buckets along each direction should be equal to the number of segments that the slices split the 3D model into. Therefore,

$$\text{num_buckets_x} = \text{num_across_x} + 1 \quad (4)$$

Each of these buckets represents a segment of the x -axis that is dx wide. For each triangle, the maximum and minimum x value out of the three vertices that make up the triangle need to be found. For each bucket that the triangle aligns with, the triangle is added to that bucket, as shown in Figure 13.

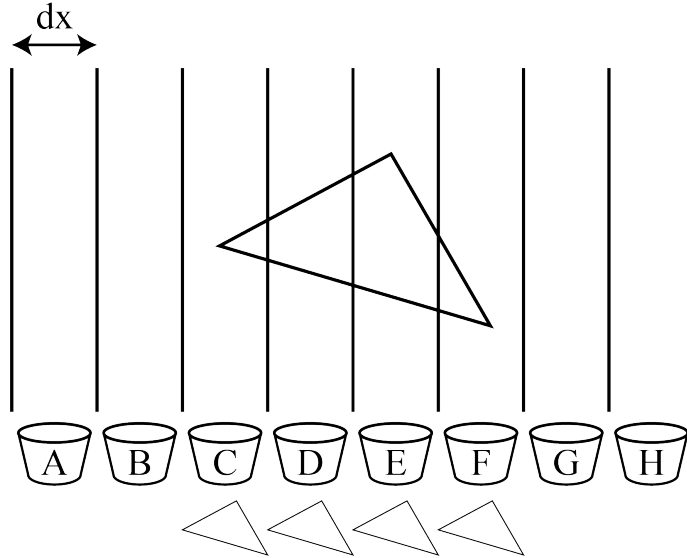


Figure 13: Placing triangles into buckets to create interval trees. The triangle is added to all intervals that it overlaps with.

This process is repeated along the y -axis, though the number of buckets varies depending on the dimensions of the 3D model, and is determined using Equation 5.

$$\text{num_buckets_y} = (\text{max_y} - \text{min_y})/\text{dx} \quad (5)$$

4.3.3 Slicing the Model

Now that the interval trees have been generated and the triangle plane intersection method has been implemented, slicing the model is straightforward. This slicing is done in `Slicing_Accelerated_X` and `Slicing_Accelerated_Y`. The two methods are very similar, so only `Slicing_Accelerated_X` will be described in this section.

For each slice along the x -axis, the system creates a plane object defined by a point and a normal. For each of these planes, the system loops through all of the triangles from the interval tree that could intersect with the plane. A list of all of the intersection edges is compiled, generating a list containing lists of intersection edges for each plane as the output of this method.

The interval tree greatly reduces the number of triangles that need to be checked for each plane, and allows the overall system to run faster. For instance, running on the same Stanford Bunny model, it took 22 seconds to slice the model while checking every triangle for each slice, and took only 0.8 seconds to slice the model using the interval tree.

4.3.4 Create Contours

After slicing the model, there is a list of edges that lie on each plane. However, they are in no particular order. To create the closed loop contours for laser cutting, the edges need to be in order. Additionally, the system must be robust to multiple contours that can exist on the same slice plane.

To create the contours, the system loops through the list of edges that intersect each plane, which is implemented in `CreateContour`. The first vertex of the first edge is chosen to be the starting point of the contour. The next closest vertex will be the vertex at the opposite end of the edge, so that will be the second vertex of the contour. `AddToContour` then continues the computation by looping through all of the remaining edges to find the vertex that is closest to the second vertex. Continue this process, checking both points on each edge each time. A distance between the first and most recent vertex of less than 10^{-6} cm determines where the contour should be closed, as such a small distance indicates that those two points are essentially the same, but some floating point error in the intersection calculations lead to them being slightly different. The contour is then closed by adding the first vertex to the end of the contour, thereby completing the loop. If at this point there are still more edges that have not been used, then the process for creating a contour is repeated because there are multiple contours on the same plane. There could be multiple contours if there is a hole or indent in the model or if the slice passes

through different sections of the model that, at this slice, are not connected, as shown in Figure 14.

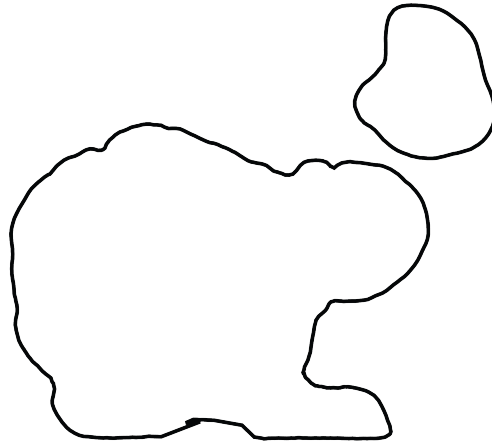
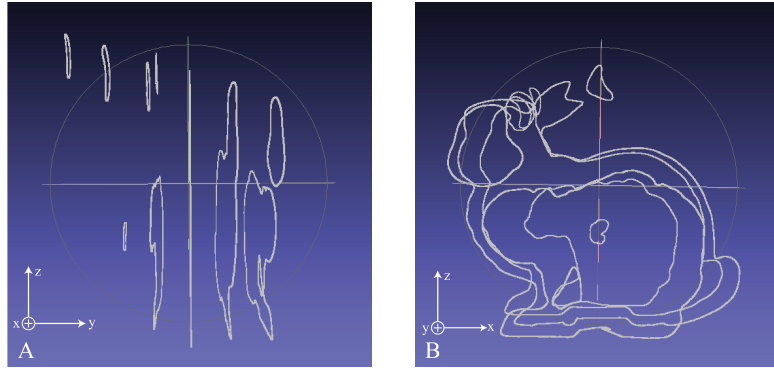


Figure 14: A slice of the Stanford Bunny model that results in two contours. The larger contour is part of the body and the smaller contour is part of the head. These two sections are connected in the model but are separate on this slice.

This contour closing process repeats for each slice plane. Figure 15 shows all of the slices for the Stanford Bunny model.

Slices parallel to the x-axis



Slices parallel to the y-axis

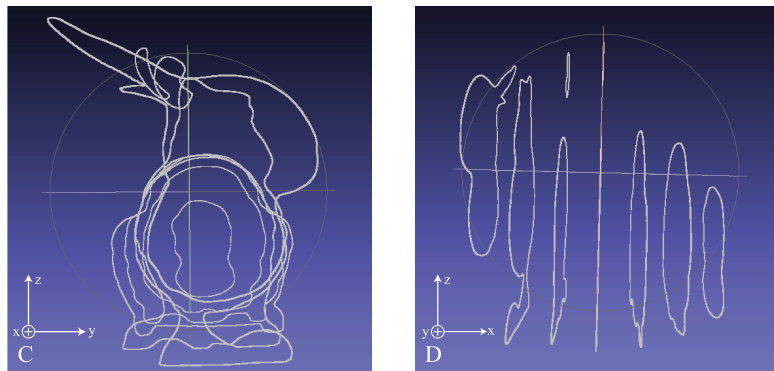


Figure 15: Slices parallel to the x- and y-axis viewed from both the x and y directions rendered in MeshLab [12].

4.4 Adding Slits

Adding the slits to the contours is the most challenging step. Three different slit generation approaches were explored. The first approach explored adding points as the contour was being created, the second approach explored adding points to the contours after they had been fully generated, and the third approach explored the creation of a separate list of vertices for the slits that could be put into the SVG on top of the contours. The third approach proved to be the most successful.

4.4.1 Approach 1: While Creating Contours

The first approach explored for adding the slits to the contours was to add them as the contours were created. The following explanation is for slices parallel to the x -axis, but the same approach can be used for those parallel to the y -axis.

First, two data structures are created; `slit_locations` is a list of coordinates along the x -axis where the slits are located. `from_top` is a list of booleans that if true mean the slit comes from the top and if false means the slit comes from the bottom. `slit_location` and `from_top` are lists that should always have the same length.

For a single layer of `intersection_edges`, the vertices start as a list of vertices in a random order. The vertices need to be sorted by their x -coordinate. This was a challenging step because the edges are pairs, so there is no easy way to sort the vertices. To create a starting point, the first vertex to be added to the contour will be the one with the smallest x -coordinate. Figure 16 shows the sorting of the vertices along the x -axis.

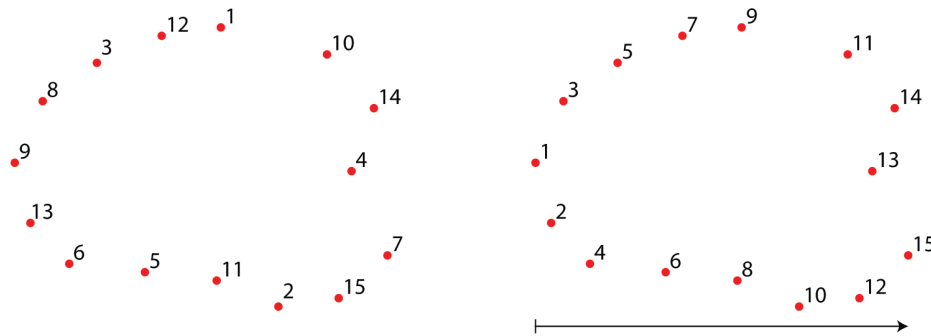


Figure 16: Sorting vertices in a single contour along the horizontal axis.

When the closest point to the first point is found, the direction that the contour is being created in can be determined. Knowing whether it is going clockwise or counterclockwise will describe whether the current point is on the top of the contour or the bottom.

The user provided the material thickness, which is used as the slit thickness, as well as the number of slices along a single axis. With this information, the location of all of the slits can be determined. For each point that is added to the contour, check if it is inside of a slit. This check should take into account the slit width, which is determined by the `material_thickness`. If the point is in a slit and on the correct side of the contour for that slit (checking `from_top`), then a point is added to the contour to create the slit. A total of 4 points are required to make a slit, as shown in Figure 17.

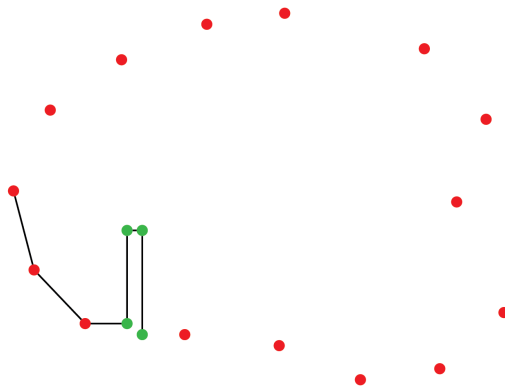


Figure 17: Inserting points in a contour to create a slit.

This process is continued along the entire contour. However, this implementation was not successful because sorting pairs of vertices could not give a meaningful result without doing cumbersome data transformations. Because of the difficulty of ordering the vertices, other approaches were explored.

4.4.2 Approach 2: Adding to Contours

The second approach of adding slits to contours is to add them after the contours are created but before the SVG is generated. The general approach for this method is to loop through all of the points in a contour. If a point falls inside of a slit, replace that point with a point at the same x -coordinate but with a z -coordinate at the slit line height.

This method could not take into account whether the slit is coming from the top or the bottom of the contour. Also, some contours cross a slit line more than twice, which leads to issues with inserted slits. Figure 18 shows an example of when slits are added from both the top and bottom of the contour due to the contours crossing the slit line multiple times.

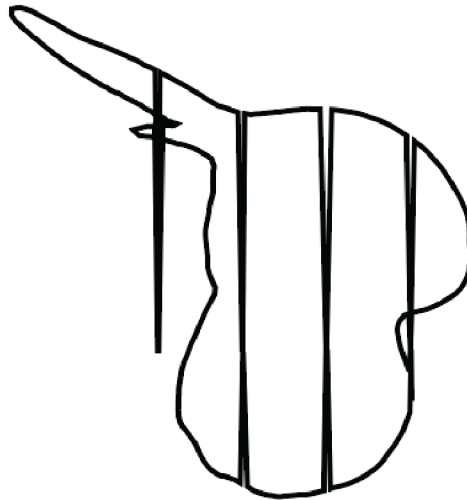


Figure 18: Multiple slits inserted in the same location on the contour.

As seen in Figure 19, the slits created with this method come to a point, and sometimes entire slits are missed. Slits can be missed if there are not enough points along a contour. If there is not a point that lies in the slit, the slit will be missed. Also, there is usually only one point in a slit because of the number of triangles making up the model, so the slit comes down to a point, creating a triangular shaped slit instead of a rectangular one.

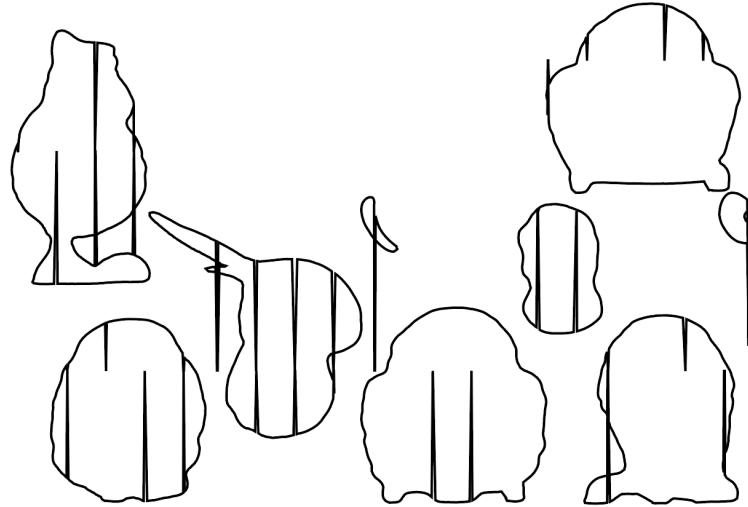


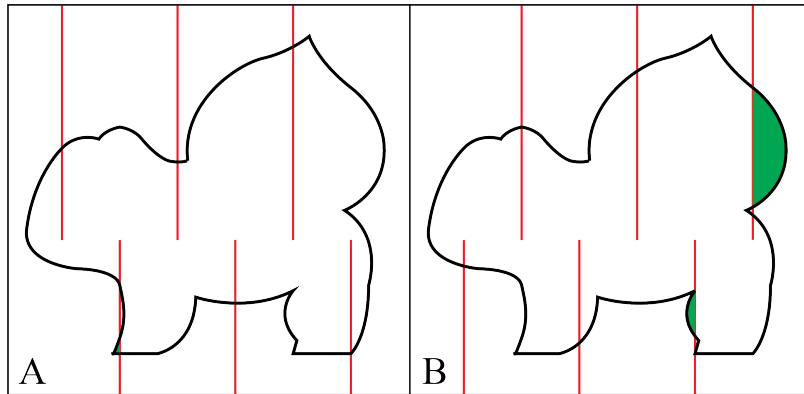
Figure 19: Result of inserting points into contours to create slits using the Stanford Bunny model.

4.4.3 Approach 3: SVG Overlay

The final method for adding the slits to the contours is to overlay the slits on top of the contour when the SVG file is generated. This proved to be the most robust system.

SVG files are text-based files that describe shapes. An SVG file is created for all the slices along each axis. Each contour is added to the SVG file as a polyline. This allows each individual contour to be moved around separately when the SVG is opened in an editing tool so that it can be placed wherever the user wants, allowing users to adjust the location of the shapes to best fit their laser cutter. Each Polyline contour is grouped with the slits that are overlaid. Grouping the contour and the slits together ensures that even when the SVG is edited they will remain in the same position relative to each other and the pop-up will still stand up once it is cut.

There are four different slit combinations, two for the slices along the x -axis and two for the slices along the y -axis. The difference between the two is whether the slits start from the top or the bottom, as shown in the Figure 20.



■ Sections that would be cut off

Figure 20: Varying slit patterns for the same slice of a manually designed Bulbasaur. A) The first slit on the left starts from the top. B) The first slit from the left starts from the bottom. Note that the choice of where to start can change what could get cut off. In slice A only the front toes are cut off, whereas in slice B the rightmost section of the model is completely cut off.

The code for creating these slits is in the Appendix in the `Slit` method. The method starts by looping through all of the slices in a single direction, for instance, say the x -axis. For a single slice parallel to the x -axis, there will be the number of slits equal to the number of slices along the y -axis. Loop through all of the slices along the y -axis, adding slits alternating from the top and bottom.

For each slit, add the four points to a polyline that is part of the SVG. This will create one line that includes all of the slits for a single slice, making the organization of the SVG easier for the user to edit later. Each of these polylines is then grouped with the contour that they match with. However, the slit polylines include every slit for each slice, so even if a contour is small, all of the slits will be generated and overlaid. This means that a contour that only overlaps with a single slit will still be grouped with all of the slits for that slice. This means that each contour takes up the same amount of area in the final SVG file. Figure 21 shows the result of overlaying slits on top of a single slice of the Stanford Bunny model.

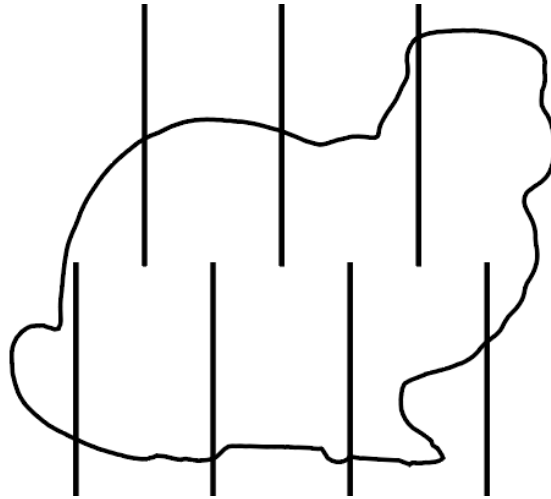


Figure 21: Result of overlaying slits on top of a single slice of the Stanford Bunny model.

4.5 Combining Contours and Slits for Exporting

To make the final SVG easier to use, each contour should be grouped with the slits so they are lined up correctly. A group in an SVG is two or more objects that when a user clicks and drags something in the group, the entire group moves together, which maintains their relative size and position. `RunTranslation_PopUp` creates two SVGs ready to be laser cut, one with all of the contours and slits parallel to the x -axis, and one with all of the contours and slits parallel to the y -axis. This makes it simpler for the user to assemble the pop-up after laser cutting, since the pieces are already split up according to their axis.

The original STL file could have points that have negative x , y , or z . Although it is possible to have negative values in an SVG file, those points would lie outside of the working area, and therefore will not be laser cut. To account for this, three shift values are calculated as shown in Equation 6.

$$x_shift = -\min_x \tag{6}$$

$$y_shift = -\min_y$$

$$z_shift = -min_z = -_bottom$$

Where min_x , min_y , and min_z are the minimum values of any point in the STL along the x -, y -, and z -axis respectively. In the z direction this minimum value is also stored as $_bottom$. These shift values are added to each point before it is added to the SVG to ensure that all of the parts will lie on the workable area.

After starting the SVG file, the system loops through all of the slices. For each contour on a slice, the system starts a group and a polyline. Next, all of the points in that contour are added to the polyline. However, the points are in three-dimensional space, and an SVG is two-dimensional. For slices parallel to the x axis, only the y and z coordinates are added to the polyline, since the x coordinate is the same for all points on the same slice. After adding all of the contours on that slice to the group, the `Slit` method is used to generate the slit pattern for this slice, and then the group is closed. The resulting SVG file is shown in the Figure 22 and 23.

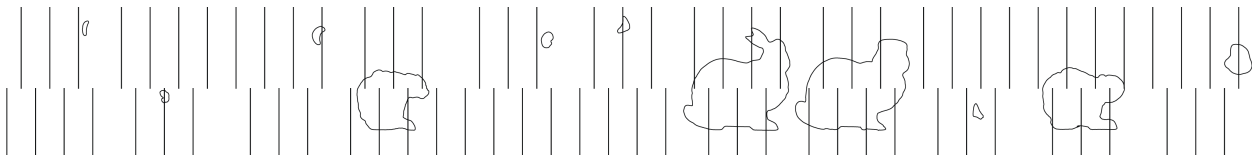


Figure 22: Resulting SVG for slices parallel to the x -axis with slits overlaid on the slices.

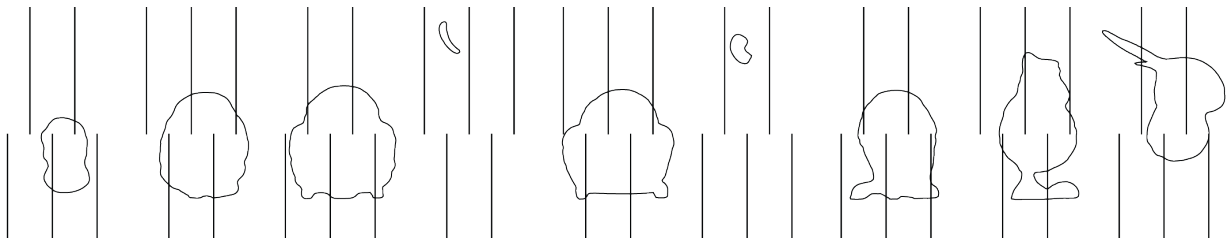


Figure 23: Resulting SVG for slices parallel to the y -axis with slits overlaid on the slices.

5. Results and Discussion

This system successfully allows 3D models to be turned into pop-up cards, as shown in Figure 24. However, there are details that the process does not handle well. For example, the

large ears of the bunny have been cut off because the system generated slits that cut all the way through them. Because the ears don't have to weave with any other slice, it is unnecessary to include those slits, but the system does not know whether or not the slits are required.

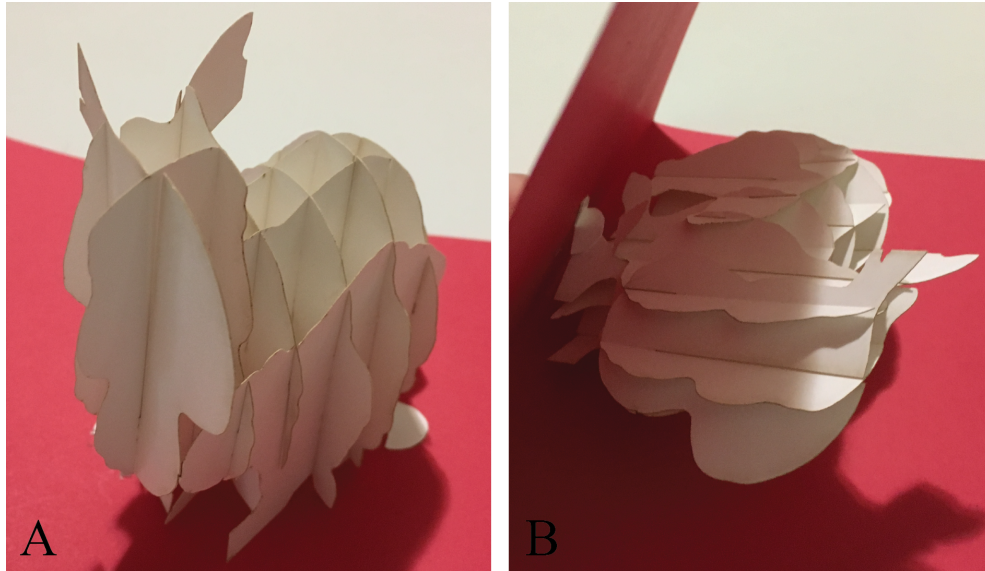


Figure 24: A fully made pop-up card of the Stanford Bunny model as created by the system. A) The model in the standing position. B) The card as it folds into the collapsed position of the model.

The user can fix erroneous details like cut off ears after the SVG has been created. If the user finds that there are parts that are cut off that could be included without affecting how the model fits together, then these edits can be made before laser cutting the pieces. Figure 26 shows the same model as Figure 24 but with some user edits. These edits are shown in Figure 25.

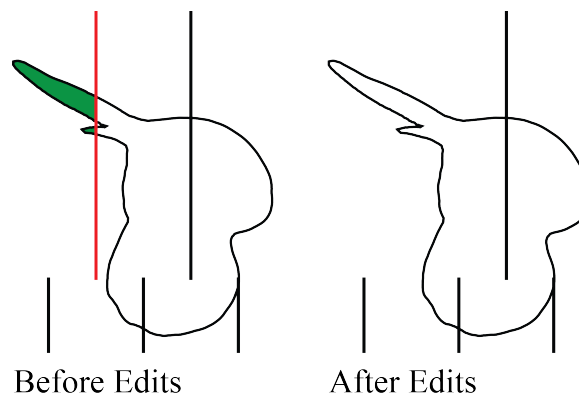


Figure 25: Edits made to a slice of the Stanford Bunny model after the system generated the SVG.

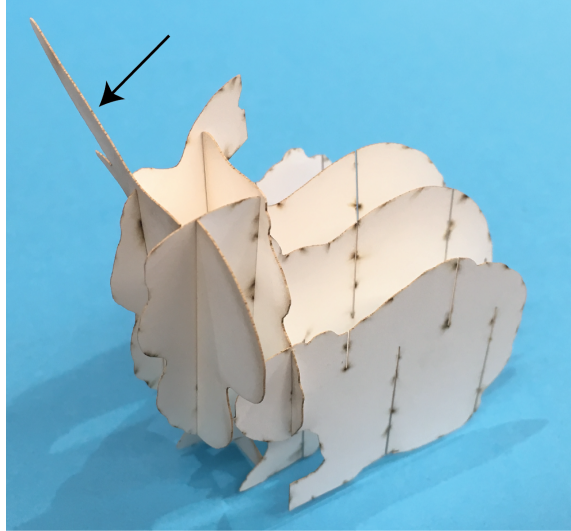


Figure 26: A fully made pop-up card of the Stanford Bunny model with small edits.

6. Conclusion and Future Work

Grid-based pop-up cards are complex and difficult to design. The designer has to visualize the final model throughout the process and ensure that the slices all line up. In a manually designed grid-based pop-up, the model tends to be simpler; symmetry is a way to reduce the number of slices that need to be designed. Grid-based pop-up card design and fabrication is difficult, but with a design tool and access to a laser cutter, there are ways to make this process easier and applicable to more complicated models that are not symmetric.

This system succeeds in slicing three-dimensional models and adding in slits that allow for the easy creation of grid-based pop-up cards. The final vector graphic can be laser cut and assembled, resulting in intricate pop-up models without having to manually design each slice. However, there are two main limitations to this system: the size of the model passed in greatly affects slicing and slit generation time and the system occasionally adds slits that would result in parts of the model being cut off.

The first problem to address for this system would be the run time of loading in the model for slicing. Currently, the load time issues limit the types of models that can be made into pop-up

cards. All of the triangles that make up the original three-dimensional model must be loaded into the system, which runs in $O(n)$ time. The larger the model, the longer it takes for this initial step. This is the main limiting factor to the system, and so pop-up cards cannot be made from large models.

Another limitation to the system is that the slits are sometimes added in such a way that part of the model is cut off. An existing work around for this is to have the user edit the resulting SVGs and remove slits that are not necessary in order to keep all of the pieces of the slice. Moving forward, additional exploration could determine a way in which the system would recognize when pieces of a slice are cut all the way through and test different slit spacing techniques to minimize creating cuts that would remove a section of the model.

This is an initial step towards creating a system that can help people do more than they can with manually designed grid-based pop-up cards. The system allows paper crafters to skip the intricate design process and use a computational system instead. The resulting SVG allows the user to make edits after the slicing has been done to correct any mistakes or change the resulting slices if they want to change the model in any way.

References

- [1] Zumbach, L., “Growing Arts and Crafts Market Isn’t Just for Kids,” *chicagotribune.com* [Online]. Available: <https://www.chicagotribune.com/business/ct-handmade-arts-and-crafts-0501-biz-20160429-story.html>. [Accessed: 26-Jan-2019].
- [2] Scrapbooker, D. H., “Scrapbooking Industry Statistics,” *LoveToKnow* [Online]. Available: https://scrapbooking.lovetoknow.com/Scrapbooking_Industry_Statistics. [Accessed: 26-Jan-2019].
- [3] “Selling Your Handmade Cards? Learn About the Card Industry,” *Spruce Crafts* [Online]. Available: <https://www.thesprucecrafts.com/greeting-card-industry-facts-and-figures-2905385>. [Accessed: 26-Jan-2019].
- [4] “By The Numbers: The Rise Of The Makerspace | Popular Science” [Online]. Available: <https://www.popsci.com/rise-makerspace-by-numbers>. [Accessed: 26-Jan-2019].
- [5] “Premium Pop-Up Cards SVG Kit - \$6.99 : SVG Files for Cricut, Silhouette, Sizzix, and Sure Cuts A Lot - SVGcuts.Com” [Online]. Available: http://svgcuts.com/index.php?main_page=product_info&products_id=204. [Accessed: 27-Jan-2019].
- [6] “Amazon.Com : Paper Spiritz 3D Valentine’s Day Card, Pop up Birthday Card, Anniversary, for Sister Mom Wife Men, with Envelopes, 3D Romantic Cards, Thank You Cards, All Occasions, Handmade Card for Kids Baby: Office Products” [Online]. Available: <https://www.amazon.com/Paper-Spiritz-Diamond-Couples-Husband/dp/B01CZMT2BS>. [Accessed: 27-Jan-2019].
- [7] “Bluebird,” *Lovepop* [Online]. Available: <https://www.lovepopcards.com/products/blue-bird-pop-up-card>. [Accessed: 30-Jan-2019].
- [8] “What Is An STL File?,” *3D Syst.* [Online]. Available: <https://www.3dsystems.com/quickparts/learning-center/what-is-stl-file>. [Accessed: 21-May-2019].
- [9] “Vector Graphic Definition” [Online]. Available: <https://techterms.com/definition/vectorgraphic>. [Accessed: 18-May-2019].
- [10] “SVG Tutorial” [Online]. Available: https://www.w3schools.com/graphics/svg_intro.asp. [Accessed: 18-May-2019].
- [11] “SVG Polyline” [Online]. Available: https://www.w3schools.com/graphics/svg_polyline.asp. [Accessed: 18-May-2019].
- [12] Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., and Ranzuglia, G., “MeshLab: An Open-Source Mesh Processing Tool,” p. 8.

A Code

A.1 Triangle Plane Intersection

```
1. std::vector<Vector3<T>> IntersectPlane(Plane<T> p) {
2.     std::vector<Vector3<T>> edge;
3.
4.     for (int i = 0; i < 3; ++i) {
5.         Vector3<T> v_1 = _vertices[i];
6.         T dist_1;
7.         if (p.onPlane(v_1, dist_1)) {
8.             edge.push_back(v_1);
9.         }
10.    }
11.    for (int i = 0; i < 3; ++i) {
12.        Vector3<T> v_1 = _vertices[i];
13.        T dist_1;
14.        if (p.onPlane(v_1, dist_1)) {
15.            continue;
16.        }
17.        int j = (i + 1)%3;
18.        Vector3<T> v_2 = _vertices[j];
19.        T dist_2;
20.        if (p.onPlane(v_2, dist_2)) {
21.            continue;
22.        }
23.        if (i == j) {
24.            continue;
25.        } else if (dist_1 * dist_2 < 0) {
26.            if (dist_1 < 0) {
27.                dist_1 = -1 * dist_1;
28.            }
29.            if (dist_2 < 0) {
30.                dist_2 = -1 * dist_2;
31.            }
32.            Vector3<T> triangle_edge = v_2 - v_1;
33.            Vector3<T> plane_intersection = v_1 + triangle_edge*(dist_1/(dist_1 + dist_
34.                2));
35.            edge.push_back(plane_intersection);
36.        }
37.    }
38.    return edge;
39.
40. private:
41.     Vector3<T> _vertices[3];
42. };
```

A.2 RunTranslation PopUp

```
1. /*
2.  * This is the method used to make the pop up structure
3.  */
4. void RunTranslation_PopUp(std::vector<std::vector<std::vector<Vector3<T>>>>& contour,
5.     T material_thickness,
```

```

6.     T slit_line_height) {
7.
8.     // Calculate the actual slit location from the ratio of the height that's passed in
9.     T slit_line_location = slit_line_height*(_top - _bottom);
10.
11.    // Initialize data structures
12.    std::vector<std::vector<std::pair<Vector3<T>, Vector3<T>>>> intersection_edges_x;
13.    std::vector<std::vector<std::pair<Vector3<T>, Vector3<T>>>> intersection_edges_y;
14.
15.    // Run the slicing for slices parallel to the x axis
16.    Slicing_Accelerated_X(_tri_mesh, intersection_edges_x);
17.
18.    // initialize the data structure for the contours
19.    std::vector<std::vector<std::vector<Vector3<T>>>> contourForSlits_x;
20.    contourForSlits_x.clear();
21.
22.    // Create the contours
23.    CreateContour(_tri_mesh, intersection_edges_x, contourForSlits_x);
24.
25.    // Calculate how much the slices need to be shifted over in the SVG
26.    T x_shift = -1*_min_x;
27.    T y_shift = -1*_min_y;
28.    T z_shift = -1*_bottom;
29.
30.    // Start the SVG for slices parallel to the x axis
31.    std::ofstream x_svg;
32.    x_svg.open("/home/computationalfabrication/CompFab/ComputationalFabrication/data/x_
svg.svg");
33.    x_svg << "<svg height='18in' width='32in'>\n";
34.
35.    // Fill the SVG file
36.    int total_contour = 0;
37.
38.    // For each slice
39.    for (int i = 0; i < contourForSlits_x.size(); ++i) {
40.        // For each contour on that slice
41.        for (int j = 0; j < contourForSlits_x[i].size(); ++j) {
42.            x_svg << "<g transform='scale(96)'>\n";
43.            x_svg << "<polyline points='";
44.
45.            // For each point in that contour
46.            for (int k = 0; k < contourForSlits_x[i][j].size(); ++k) {
47.                // Add the point to the SVG file
48.                Vector3<T> point = contourForSlits_x[i][j][k];
49.                x_svg << std::to_string(point[1] + y_shift + total_contour*( _max_y -
_min_y));
50.                x_svg << ",";
51.                x_svg << std::to_string(point[2] + z_shift);
52.                x_svg << " ";
53.            }
54.            x_svg << "'\n style='fill:none;stroke:black;stroke-width:1' />\n";
55.
56.            // Add the correct slits to this contour
57.            Slit(x_svg, material_thickness, slit_line_location,
58.                y_shift + total_contour*( _max_y -
_min_y), z_shift, i, _min_y, _max_y, _dx_x);
59.            x_svg << "</g>";
60.            total_contour += 1;
61.        }
62.    }

```

```

63. // Close the SVG file
64. x_svg << "</svg>";
65. x_svg.close();
66.
67. // Visualize the contours created
68. VisualizeContour("/home/computationalfabrication/CompFab/ComputationalFabrication/d
ata/bunny-contour-x.ply", 0.001, contourForSlits_x);
69.
70. // Run the slicing for slices parallel to the y axis
71. Slicing_Accelerated_Y(_tri_mesh, intersection_edges_y);
72.
73. // initialize the data structure for the contours
74. std::vector<std::vector<std::vector<Vector3<T>>>> contourForSlits_y;
75. contourForSlits_y.clear();
76.
77. // Create the contours
78. CreateContour(_tri_mesh, intersection_edges_y, contourForSlits_y);
79.
80. // Start the SVG for slices parallel to the y axis
81. std::ofstream y_svg;
82. y_svg.open("/home/computationalfabrication/CompFab/ComputationalFabrication/data/y_
svg.svg");
83. y_svg << "<svg height='18in' width='32in'>\n";
84.
85. // Fill the SVG file
86. total_contour = 0;
87.
88. // For each slice
89. for (int i = 0; i < contourForSlits_y.size(); ++i) {
90.     // For each contour on that slice
91.     for (int j = 0; j < contourForSlits_y[i].size(); ++j) {
92.         y_svg << "<g transform='scale(96)'>\n";
93.         y_svg << "<polyline points='";
94.
95.         // For each point in that contour
96.         for (int k = 0; k < contourForSlits_y[i][j].size(); ++k) {
97.             // Add the point to the SVG file
98.             Vector3<T> point = contourForSlits_y[i][j][k];
99.             y_svg << std::to_string(point[0] + x_shift + total_contour*(max_x -
_min_x));
100.             y_svg << ",";
101.             y_svg << std::to_string(point[2] + z_shift);
102.             y_svg << " ";
103.         }
104.
105.         y_svg << "'\n style='fill:none;stroke:black;stroke-width:1' />\n";
106.
107.         // Add the correct slits to this contour
108.         Slit(y_svg, material_thickness, slit_line_location,
109.             x_shift + total_contour*(max_x -
_min_x), z_shift, i + 1, _min_x, _max_x, _dx_x);
110.         y_svg << "</g>";
111.         total_contour += 1;
112.     }
113. }
114. // Close the SVG file
115. y_svg << "</svg>";
116. y_svg.close();
117.
118. VisualizeContour("/home/computationalfabrication/CompFab/ComputationalFabrication/d
ata/bunny-contour-y.ply", 0.001, contourForSlits_y);

```



```

119.}
120.
121.private:
122.    mesh::TriMesh<T> _tri_mesh;
123.
124.    /* Variables for slicing */
125.    T _bottom, _top, _dx, _min_x, _max_x, _dx_x, _min_y, _max_y, _dy;
126.
127.    /* accelerated data structure */
128.    data_structure::IntervalTree<T> _interval_tree;
129.
130.    std::vector<std::vector<std::vector<Vector3<T>>>> z_map;
131.    std::vector<std::vector<std::vector<Vector3<T>>>> x_map;
132.    std::vector<std::vector<std::vector<Vector3<T>>>> y_map;
133.};

```

A.3 Accelerated Slicing Parallel to the X- and Y-Axis

```

1.  /*
2.  * Accelerated slicing to generate slices parallel to the x axis
3.  */
4.  void Slicing_Accelerated_X(mesh::TriMesh<T>& tri_mesh,
5.    std::vector<std::vector<std::pair<Vector3<T>, Vector3<T>>>> &intersection_edges) {
6.
7.    std::vector<Eigen::Vector3i>& elements = tri_mesh.elements();
8.    std::vector<Vector3<T>>& vertices = tri_mesh.vertices();
9.    std::vector<Eigen::Vector3i>& edges = tri_mesh.edges();
10.
11.    intersection_edges.clear();
12.    int i = 0;
13.
14.    // For each layer that will be sliced
15.    for (T x = _min_x; x <= _max_x; x += _dx_x) {
16.        // Get the triangles that cross this plane
17.        std::vector<std::vector<Vector3<T>>>& candidates = x_map[i];
18.
19.        // Initialize data structures
20.        std::vector<std::pair<Vector3<T>, Vector3<T>>> intersections_one_plane;
21.        intersections_one_plane.clear();
22.
23.        // Initialize a plane parallel to the x-axis
24.        geometry::Plane<T> plane(Vector3<T>(x, 0, 0), Vector3<T>(1, 0, 0));
25.
26.        // For each triangle being considered
27.        for (int ii = 0; ii < candidates.size(); ++ii) {
28.            // Calculate the triangle plane intersection
29.            std::vector<Vector3<T>>& tri = candidates[ii];
30.            geometry::Triangle<T> triangle(tri[0], tri[1], tri[2]);
31.            std::vector<Vector3<T>> intersections = triangle.IntersectPlane(plane);
32.
33.            std::pair<Vector3<T>, Vector3<T>> edge;
34.            int size = intersections.size();
35.
36.            // If the triangle intersects the plane in two places (full intersects)
37.            if (size == 2) {
38.                // Add the two points where it intersects as an edge

```

```

39.             edge = std::make_pair(intersections[0], intersections[1]);
40.             intersections_one_plane.push_back(edge);
41.         }
42.     }
43.     intersection_edges.push_back(intersections_one_plane);
44.     i += 1;
45. }
46. }

```

```

1.  /*
2.  * Accelerated slicing to generate slices parallel to the y axis
3.  */
4.  void Slicing_Accelerated_Y (mesh::TriMesh<T>& tri_mesh,
5.      std::vector<std::vector<std::pair<Vector3<T>, Vector3<T>>>> &intersection_edges) {
6.
7.     std::vector<Eigen::Vector3i>& elements = tri_mesh.elements();
8.     std::vector<Vector3<T>>& vertices = tri_mesh.vertices();
9.     std::vector<Eigen::Vector3i>& edges = tri_mesh.edges();
10.
11.    intersection_edges.clear();
12.    int j = 0;
13.
14.    // For each layer that will be sliced
15.    for (T y = _min_y; y <= _max_y; y += _dx_x) {
16.        // Get the triangles that cross this plane
17.        std::vector<std::vector<Vector3<T>>>& candidates = y_map[j];
18.
19.        // Initialize data structures
20.        std::vector<std::pair<Vector3<T>, Vector3<T>>> intersections_one_plane;
21.        intersections_one_plane.clear();
22.
23.        // Initialize a plane parallel to the y-axis
24.        geometry::Plane<T> plane(Vector3<T>(0, y, 0), Vector3<T>(0, 1, 0));
25.
26.        // For each triangle being considered
27.        for (int ii = 0; ii < candidates.size(); ++ii) {
28.            // Calculate the triangle plane intersection
29.            std::vector<Vector3<T>>& tri = candidates[ii];
30.            geometry::Triangle<T> triangle(tri[0], tri[1], tri[2]);
31.            std::vector<Vector3<T>> intersections = triangle.IntersectPlane(plane);
32.
33.            std::pair<Vector3<T>, Vector3<T>> edge;
34.            int size = intersections.size();
35.
36.            // If the triangle intersects the plane in two places (full intersects)
37.            if (size == 2) {
38.                edge = std::make_pair(intersections[0], intersections[1]);
39.                intersections_one_plane.push_back(edge);
40.            }
41.        }
42.        intersection_edges.push_back(intersections_one_plane);
43.        j += 1;
44.    }
45. };
46.
47. private:
48.     mesh::TriMesh<T> _tri_mesh;
49.
50.     /* Variables for slicing */

```

```

51.     T_bottom, _top, _dx, _min_x, _max_x, _dx_x, _min_y, _max_y, _dy;
52.
53.     /* accelerated data structure */
54.     data_structure::IntervalTree<T> _interval_tree;
55.
56.     std::vector<std::vector<std::vector<Vector3<T>>>> z_map;
57.     std::vector<std::vector<std::vector<Vector3<T>>>> x_map;
58.     std::vector<std::vector<std::vector<Vector3<T>>>> y_map;
59. };

```

A.4 CreateContour

```

1. void CreateContour(mesh::TriMesh<T>& tri_mesh,
2.     std::vector<std::vector<std::pair<Vector3<T>, Vector3<T>>>> &intersection_edges,
3.     std::vector<std::vector<std::vector<Vector3<T>>>>& contours) {
4.
5.     for (int i = 0; i < intersection_edges.size(); ++i) {
6.         std::vector<std::pair<Vector3<T>, Vector3<T>>> intersection_edges_plane = inter
7. section_edges[i];
8.
9.         std::vector<std::vector<Vector3<T>>> contours_plane;
10.
11.         while (intersection_edges_plane.size() > 0) {
12.             std::vector<Vector3<T>> contour;
13.             contour.push_back(intersection_edges_plane[0].first);
14.             contour.push_back(intersection_edges_plane[0].second);
15.             intersection_edges_plane.erase(intersection_edges_plane.begin(), intersecti
16. on_edges_plane.begin() + 1);
17.
18.             bool contour_complete = false;
19.
20.             while (!contour_complete) {
21.                 AddToContour(contour, intersection_edges_plane, contour_complete);
22.             }
23.             contours_plane.push_back(contour);
24.         }
25.     }
26.
27. private:
28.     mesh::TriMesh<T> _tri_mesh;
29.
30.     /* Variables for slicing */
31.     T_bottom, _top, _dx, _min_x, _max_x, _dx_x, _min_y, _max_y, _dy;
32.
33.     /* accelerated data structure */
34.     data_structure::IntervalTree<T> _interval_tree;
35.
36.     std::vector<std::vector<std::vector<Vector3<T>>>> z_map;
37.     std::vector<std::vector<std::vector<Vector3<T>>>> x_map;
38.     std::vector<std::vector<std::vector<Vector3<T>>>> y_map;
39. };

```

A.5 AddToContour

```
1.  /*
2.  * Adds a the next closest point to the contour being created
3.  */
4.  void AddToContour(std::vector<Vector3<T>>& contour,
5.    std::vector<std::pair<Vector3<T>,
6.    Vector3<T>>>& intersection_edges_plane,
7.    bool& contour_complete) {
8.
9.    // Start the closest distance as really far away
10.   T closest_distance = 99999999.0f;
11.   Vector3<T> closest_vertex;
12.   Vector3<T> connected_vertex;
13.   int closest_index = -1;
14.
15.   // Check if all of the contours have already been created
16.   if (intersection_edges_plane.size() == 0) {
17.     contour_complete = true;
18.     return;
19.   }
20.
21.   // For all the edges not added to a contour yet
22.   for (int i = 0; i < intersection_edges_plane.size(); ++i) {
23.     // Get the first and second element fo the edge
24.     Vector3<T> first_v = intersection_edges_plane[i].first;
25.     Vector3<T> second_v = intersection_edges_plane[i].second;
26.
27.     // Calculate the distance between the vertices and the last vertex added to the
28.     // contour
29.     T dist_1 = (first_v - contour[contour.size() - 1]).norm();
30.     T dist_2 = (second_v - contour[contour.size() - 1]).norm();
31.
32.     // Check if either of these vertices are closer than the last closest vertex
33.     if (dist_1 < closest_distance) {
34.       closest_distance = dist_1;
35.       closest_vertex = first_v;
36.       connected_vertex = second_v;
37.       closest_index = i;
38.     }
39.     if (dist_2 < closest_distance) {
40.       closest_distance = dist_2;
41.       closest_vertex = second_v;
42.       connected_vertex = first_v;
43.       closest_index = i;
44.     }
45.     // Check if the contour is finished and the loop should be closed
46.     T dist_first = (contour[0] - contour[contour.size() - 1]).norm();
47.     if (dist_first < 1e-6) {
48.       contour_complete = true;
49.
50.       closest_distance = dist_first;
51.       contour.push_back(contour[0]);
52.
53.     } else {
54.       // Add the vertex to the contour
55.       contour.push_back(connected_vertex);
56.       intersection_edges_plane.erase(intersection_edges_plane.begin() + closest_index
, intersection_edges_plane.begin() + closest_index + 1);
```

```

57.     }
58. }
59.
60. private:
61.     mesh::TriMesh<T> _tri_mesh;
62.
63.     /* Variables for slicing */
64.     T _bottom, _top, _dx, _min_x, _max_x, _dx_x, _min_y, _max_y, _dy;
65.
66.     /* accelerated data structure */
67.     data_structure::IntervalTree<T> _interval_tree;
68.
69.     std::vector<std::vector<std::vector<Vector3<T>>>> z_map;
70.     std::vector<std::vector<std::vector<Vector3<T>>>> x_map;
71.     std::vector<std::vector<std::vector<Vector3<T>>>> y_map;
72. };

```

A.6 Slit

```

1.  /*
2.   * Adds slits to an SVG file. This was successful and is the method used in the final v
3.   */
4.  void Slit(std::ofstream& svg,
5.           T material_thickness,
6.           T slit_height_location,
7.           T horiz_shift,
8.           T z_shift,
9.           int layer,
10.          T min,
11.          T max,
12.          T dx) {
13.
14.     int num_slit = 0;
15.
16.     // For each slice
17.     for (T y = (min + dx); y <= (max - dx); y += dx) {
18.         svg << "<polyline points="";
19.
20.         // If the slit number and the layer number are either both odd or both even
21.         if (((num_slit % 2 == 0) && (layer % 2 == 0)) || ((num_slit % 2 != 0) && (layer
22. % 2 != 0))) {
23.             // Add a slit coming from the top
24.             svg << std::to_string(y - material_thickness/0.5 + horiz_shift);
25.             svg << ",";
26.             svg << std::to_string(_top - z_shift);
27.             svg << " ";
28.
29.             svg << std::to_string(y - material_thickness/0.5 + horiz_shift);
30.             svg << ",";
31.             svg << std::to_string(slit_height_location);
32.             svg << " ";
33.
34.             svg << std::to_string(y + material_thickness/0.5 + horiz_shift);
35.             svg << ",";
36.             svg << std::to_string(slit_height_location);
37.             svg << " ";

```

```

38.         svg << std::to_string(y + material_thickness/0.5 + horiz_shift);
39.         svg << ",";
40.         svg << std::to_string(_top - z_shift);
41.         svg << " ";
42.
43.     } else {
44.         // Add a slit coming from the bottom
45.         svg << std::to_string(y - material_thickness/0.5 + horiz_shift);
46.         svg << ",";
47.         svg << std::to_string(_bottom - z_shift);
48.         svg << " ";
49.
50.         svg << std::to_string(y - material_thickness/0.5 + horiz_shift);
51.         svg << ",";
52.         svg << std::to_string(slit_height_location);
53.         svg << " ";
54.
55.         svg << std::to_string(y + material_thickness/0.5 + horiz_shift);
56.         svg << ",";
57.         svg << std::to_string(slit_height_location);
58.         svg << " ";
59.
60.         svg << std::to_string(y + material_thickness/0.5 + horiz_shift);
61.         svg << ",";
62.         svg << std::to_string(_bottom - z_shift);
63.         svg << " ";
64.     }
65.     svg << "\n style='fill:none;stroke:black;stroke-width:1' />\n";
66.     num_slit += 1;
67. }
68. }
69.
70. private:
71.     mesh::TriMesh<T> _tri_mesh;
72.
73.     /* Variables for slicing */
74.     T _bottom, _top, _dx, _min_x, _max_x, _dx_x, _min_y, _max_y, _dy;
75.
76.     /* accelerated data structure */
77.     data_structure::IntervalTree<T> _interval_tree;
78.
79.     std::vector<std::vector<std::vector<Vector3<T>>>> z_map;
80.     std::vector<std::vector<std::vector<Vector3<T>>>> x_map;
81.     std::vector<std::vector<std::vector<Vector3<T>>>> y_map;
82. };

```

A.7 GenerateMaps

```

1. void GenerateMaps(T min_x, T max_x, T dx, T min_y, T max_y) {
2.     // Set the values of the private variables
3.     _min_x = min_x;
4.     _max_x = max_x;
5.     _dx_x = dx;
6.     _min_y = min_y;
7.     _max_y = max_y;
8.
9.     // Initialize the data structures for the interval trees
10.    for (T h = min_x; h <= max_x; h += _dx_x) {

```

```

11.     x_map.push_back(std::vector<std::vector<Vector3<T>>>());
12. }
13.
14. for (T h = min_y; h <= max_y; h += _dx_x) {
15.     y_map.push_back(std::vector<std::vector<Vector3<T>>>());
16. }
17.
18. // Fill the interval trees
19. // For each triangle in the mesh
20. for (int i = 0; i < _tri_mesh.elements().size(); ++i) {
21.     std::vector<Vector3<T>> triangle;
22.     for (int j = 0; j < 3; j++) {
23.         triangle.push_back(_tri_mesh.vertices(_tri_mesh.elements(i)[j]));
24.     }
25.
26.     // Find the minimum and maximum values of the triangle along both x and y axes
27.     float v_min_x = std::min(triangle[0][0], std::min(triangle[1][0], triangle[2][0
28. ]));
29.     float v_max_x = std::max(triangle[0][0], std::max(triangle[1][0], triangle[2][0
30. ]));
31.     float v_min_y = std::min(triangle[0][1], std::min(triangle[1][1], triangle[2][1
32. ]));
33.     float v_max_y = std::max(triangle[0][1], std::max(triangle[1][1], triangle[2][1
34. ]));
35.
36.     // Find the minimum layer in the x direction that the triangle crosses
37.     int min_x_layer = floor((v_min_x - min_x)/dx);
38.     int max_x_layer = ceil((v_max_x - min_x)/dx);
39.
40.     // Add the triangle to all of the intervals that it crosses along the x axis
41.     for (int j = min_x_layer; j <= max_x_layer; ++j) {
42.         x_map[j].push_back(triangle);
43.     }
44.
45.     // Find the minimum layer in the y direction that the triangle crosses
46.     int min_y_layer = floor((v_min_y - min_y)/dx);
47.     int max_y_layer = ceil((v_max_y - min_y)/dx);
48.
49.     // Add the triangle to all of the intervals that it crosses along the y axis
50.     for (int j = min_y_layer; j <= max_y_layer; ++j) {
51.         y_map[j].push_back(triangle);
52.     }
53. }
54. private:
55.     mesh::TriMesh<T> _tri_mesh;
56.
57.     /* Variables for slicing */
58.     T _bottom, _top, _dx, _min_x, _max_x, _dx_x, _min_y, _max_y, _dy;
59.
60.     /* accelerated data structure */
61.     data_structure::IntervalTree<T> _interval_tree;
62.
63.     std::vector<std::vector<std::vector<Vector3<T>>>> x_map;
64.     std::vector<std::vector<std::vector<Vector3<T>>>> y_map;
65. };

```