Southern Adventist University
KnowledgeExchange@Southern

MS in Computer Science Project Reports School of Computing

5-13-2016

Design and Implementation of Asymptotically Optimal Mesh Slicing Algorithms Using Parallel Processing

Christopher Dant
cdant@southern.edu

Recommended Citation

Dant, Christopher, "Design and Implementation of Asymptotically Optimal Mesh Slicing Algorithms Using Parallel Processing" (2016). *MS in Computer Science Project Reports*. 2.
https://knowledge.e.southern.edu/mscs_reports/2

Design and Implementation of Asymptotically Optimal Mesh Slicing Algorithms Using Parallel Processing

Approved by:

Professor Scot Anderson, Adviser

Tyson S. Hall

Professor Tyson Hall

Professor Richard Halterman

Date Approved 5/6/16

DESIGN AND IMPLEMENTATION OF ASYMPTOTICALLY OPTIMAL MESH SLICING ALGORITHMS USING PARALLEL PROCESSING

by

Christopher Dant

A PROJECT REPORT

Presented to the Faculty of

The School of Computing at the Southern Adventist University

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Anderson

Collegedale, Tennessee

September, 2015

DESIGN AND IMPLEMENTATION OF ASYMPTOTICALLY OPTIMAL MESH SLICING ALGORITHMS USING PARALLEL PROCESSING

Christopher Dant, M.S.
Southern Adventist University, 2015

Adviser: Scot Anderson, Ph.D.

Mesh slicing is the process of taking a three dimensional model and reducing it to 2.5 dimensional layers that together create a layered representation of the model. The process is used in layered additive manufacturing, three dimensional voxelization, and other similar problems in computational geometry. The slicing process is computationally expensive, and the time required to slice an object can inhibit the viability of layered manufacturing in some industries. We designed and developed a fast implementation of the slicing process, called Sunder, that uses new asymptotically optimal algorithms and takes advantage of parallel processing platforms. To our knowledge, no other slicing implementation leverages massive parallel execution hardware, such as graphics processing units (GPUs), leaving significant potential for improvement. Furthermore, no published set of slicing algorithms completes all three major steps in the slicing process (preprocessing, slicing, and contour assembly) in linear time complexity, which our design achieves. Therefore, our implementation improves the current state of the art in mesh slicing.

COPYRIGHT

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reducing three-dimensional models to 2.5 dimensional layers is a required step for several problems in computational geometry. One of the applications of this process, called slicing, is the creation of layered machining paths used in layered additive manufacturing, or 3D printing. Motion View Software LLC, a manufacturer of 3D printers and our partner in this project, has experienced a wide range in slicer application performance, from the Slic3r [1] project on the low end to Simplify3D [2] on the high end; their comparisons show Simplify3D is over an order of magnitude faster than Slic3r. This variance shows that the 3D printing industry has no standard for slicing algorithms, suggesting room for refinement and improvement. Furthermore, while hardware-assisted mass parallelism techniques have been applied to similar problems [3, 4], we are aware of no research that applies such parallel computing techniques to the slicing problem on modern hardware [5].

This project's goal, then, was to develop slicing algorithms with a low asymptotic time complexity and implement them while exploiting unused computation resources, such as graphics processing units (GPUs), to optimize slicing performance. However, computing tasks on devices other than the CPU are device-specific by nature. It was our goal to create a slicer that performed well on a variety of available hardware. This goal adds the requirement that our application architecture must be modular and extensible so support for new devices can be added in the future. Our splicing implementation is intended for actual use in industry, and must, therefore, support at least the basic features of a real-world industrial slicer. We discuss these requirements in detail in Chapter 3. The industrial application of the slicer required that our slicer's output needed to accurately represent the three dimensional model given to the slicer as input. Finally, while our goals focused on optimizing for speed and GPU-assisted computing, our slicer had to use other system resources in reasonable quantities to be viable in a real-world environment.

Layered additive manufacturing is exciting technology that creates opportunities that were not possible or feasible in the past. Recent advances in the technology enable rapid production of prototypes or manufacture of objects that are unique to a specific case, such as medical equipment for an individual person. However, some applications of case-specific manufacturing require a faster manufacturing process than is currently available. Creating a faster implementation of the slicing process pushes additive manufacturing toward being able to meet those needs. Our slicing implementation was developed for Motion View Software LLC, a company applying additive manufacturing to case-specific industrial solutions, but the competition provided by a parallel-computing focused implementation of slicing algorithms will necessarily spur competition and growth in the industry.

In Chapter 2, we discuss more background information on the slicing problem and other solutions. In Chapter 3, we present our approach and a detailed report of our implementation. In Chapter 4, we explain our evaluation methods and goals, and in Chapter 5, give the results of our evaluation. In Chapter 6, we summarize and conclude our report.

Chapter 2

Background

Our project primarily interacts with two research areas. The first is the triangle mesh slicing problem, which has been approached many ways since the inception of additive manufacturing. The second area is heterogeneous program execution, or executing one program across multiple, varying processing platforms. Because heterogenous execution techniques and design are unique to each processing platform used in tandem with the CPU, such as GPUs or signal processors, we focus our research on heterogenous execution using our proposed hardware accelerations platform, GPUs.

2.1 Slicing Triangle Meshes

Slicing triangle meshes into layers that manufacturing machines can build is fundamentally an optimization problem. The process typically consumes immense system resources, especially processing power and memory. Most techniques focus on optimizing for either computational or memory efficiency at the expense of the other.

The slicing problem has two key variants used by different manufacturing processes. The first involves the creation of slices of varying width to best reproduce the contour of the object perpendicular to the slicing plane. Pandey et al. [6] provide a comprehensive overview of this variant, discussing issues and covering a breadth of techniques for determining the thickness of each slice. Research regarding this variant tends to focus on methods for calculating layer thickness rather than optimizing the slicing process itself. Tata et al. [7], however, present a slicing engine that uses facet grouping and feature recognition to improve the accuracy and efficiency of variable thickness slicing. Variable thickness manufacturing techniques have lost some popularity since cheap 3D printers have made additive manufacturing much more available while typically only allowing uniform layer thickness. The manufacturing process our slicer targets assumes that layers will generally be of uniform thickness.

The second variant assumes layers will be the same thickness in most cases; while layer thickness may be varied for additional durability on the top and bottom of the manufactured object, thickness is not used to more accurately manufacture the perpendicular contour. Thus, research on this variant of the problem focuses on generating the object's contour for each slice as efficiently as possible. Choi and Kwok [8, 9] propose two related techniques for uniform thickness slicing, optimizing for reduced memory usage by pulling into memory only the triangles from the mesh that are necessary to compute the current slice. These techniques are ideal for low memory systems or massive meshes but adds significant CPU overhead. Vatani et al. [10] also propose a solution where additional computation resources are used to minimize the number of triangles pulled into memory. They also use nearest-point analysis to generate the slice contours; this technique has interesting mesh correction applications but also introduces heuristic inaccuracy into contour generation. Our slicer's use case allows for systems with large memories capable of storing multiple large meshes simultaneously but has tight time constraints. Thus, algorithms that optimize for memory are not appropriate for our project.

Gregori et al. [11] present an algorithm that organizes triangles into a tree structure based on the minimum and maximum coordinates on the axis perpendicular to the slicing plane. It then uses this tree to iterate through the triangles, calculating intersections with the slices that fall in a given triangle's interval, avoiding slices outside a triangle's interval. Huang et al. [12] propose a similar technique that uses hashing instead of a tree structure, saving on prepossessing time in exchange for not avoiding all unnecessary slices outside a triangle's interval when calculating intersections. McMains and Squin [13] introduce a sweep plane slicing algorithm that begins slicing at one edge of the mesh and continues progressively across it, updating status information as the mesh is discovered and slicing along the way. This sweep plane technique is incredibly efficient for regular shapes with few vertices but may not perform well when slicing irregular or dense meshes. In addition, the vertex processing necessary for maintaining status information is involved and difficult to implement. These three techniques are all promising approaches to reducing processing time when slicing meshes. However, they are are all sequential algorithms with no obvious parallel computing applications. Our slicer should perform well across varying systems and hardware availability, and we will evaluate these algorithms for inclusion in our slicer when handling slicing performed strictly by the CPU, but other techniques will be needed for parallel processing devices such as the GPU.

Research applying parallel algorithms or GPU-assisted processing to triangle mesh slicing is scarce. Kirschman and Jara-Almonte [5] discuss slicing using parallel computing, but no specific algorithm is given and the systems used are so antiquated that the assumptions for hardware capability in a modern context likely obsolete their techniques. Liao [3] and Hsieh et al. [4] apply GPU-assisted computing to a similar problem, mesh voxelization. Mesh voxelization involves slice layers and generating contours and filling them, much like the mesh slicing problem. However, the actual algorithms are very different because voxelization produces a raster image for each layer and slicing produces a vector image for each layer. GPUs are designed for generating and displaying raster images, and thus, voxelization algorithms tend to use typical GPU graphics libraries rather than the GPU-assisted computing methods discussed in the next section that are applicable to our problem.

2.2 GPU-Assisted Program Execution

Using multiple, diverse processing units together to execute a complex task or set of related tasks is known as heterogeneous computing. GPU-assisted programming, where a GPU is used to assist a CPU by more quickly executing massively parallel sections of code, is a specific form of heterogeneous computing. Heterogeneous computing is both a developing field and very application-specific in implementation. Because of this, research on heterogeneous computing concentrates on scheduling tasks across heterogeneous systems, a problem intrinsically faced by all such systems. In GPU-assisted computing, the GPU schedules assigned tasks in an established way known to the programmer ahead of time, and the CPU assigns all tasks the GPU performs, avoiding this scheduling problem. Thus, research on generic heterogeneous computing will not apply to our project as long as GPU-assisted computing remains our hardware acceleration technique.

Two GPU computing frameworks dominate the market. Compute Unified Device Architecture (CUDA) is a proprietary framework specifically for use with Nvidia brand GPUs. OpenCL is an open-source framework maintained by Khronos Group that is supported by most processor manufacturers, including AMD, Intel, and Nvidia, three primary manufacturers of GPU chipsets. Comparison of mature versions of the two frameworks has yielded the consensus that their performance is nearly identical [14]. Because our project seeks to create a slicer that performs well on a variety of hardware, one or both frameworks may be used to allow peak performance on a variety of GPUs.

OpenCL and CUDA use a nearly identical programming model [15, 16]. For convenience, OpenCL terminology will be used here. The CPU, known as the host in a GPU programming model, is the central controller for all processing. It handles primarily sequential sections of code and sends work to the GPU for parallel sections of code. It also handles transfer of data between the main memory of the CPU, or host memory, and the main memory of the GPU, or global memory. Code to be executed on the GPU is called a kernel, which is analogous to a function in most programming languages. A kernel is a basic series of instructions that operates on a small subset of the problem's data and can be executed in parallel.

The GPU itself is called a compute device; devices other than GPUs can also be compute devices. A compute device is made up of multiple compute units, each of which performs assigned tasks independently of all other compute units. Each compute unit is made up of compute elements, small processing units that actually execute GPU machine language instructions. A work item is one instance of a kernel that is executed on a single compute element. Each compute element has private registers for executing its work item called private memory. Batches of work items that are assigned to a compute unit are called work groups, which share that compute unit's local memory, a cache manually controlled by the programmer [16]. Depending on implementation, some synchronization features are available between work items within a work group, but never between work groups. All compute units share the GPU's global memory, and because global memory is the only GPU memory the CPU can access, all kernel results must be stored there [15]. For a more detailed introduction to GPU-assisted programming, Owens et al. [17] provide an in-depth article.

Chapter 3

Project Approach and Solution

Our first design consideration for this project was computational performance, performing the slicing steps as rapidly as possible. We first researched current algorithms but found little research applying parallelization to the slicing problem. Furthermore, all the slicing processes we found in the literature used at least one O(n log n) time complexity algorithm. Lacking a satisfactory foundation to build off of, we started from scratch, designing our own algorithms as we proceeded through the slicing steps, hoping to improve on previous work. In this chapter we discuss the requirements of our slicing engine, Sunder, the methods and algorithms used to meet those requirements, and the deliverables generated by the project.

3.1 Project Requirements and Methods

Our slicing implementation needed to provide the functionality required by Motion View Software to use it as part of their additive manufacturing platform. It supports basic industry-standard features and settings, performs competitively at industrial workloads, and is readily extensible so functionality can be added easily by other developers after the project is complete.

The software is implemented as a Windows console application so that it can be easily integrated into existing software; no graphical user interface was desired. A console application is preferred because the slicer is meant to be a component of a complete additive manufacturing control application, and console applications can easily be used as both a component and as a standalone solution. We used C++ for the host (CPU) code to take advantage of performance gains possible when using a lower level language. Device-specific code was written in whatever language the platform supported. Our current GPU code implementation uses OpenCL's variant of C that contains extra language features for parallelization. We use AMD's implementation of and software development kit for OpenCL.

Our input was specified as two files, a three dimensional triangular mesh and a configuration file. Our software reads the configuration file and uses it to set up the parameters for the slicing process. The configuration file supports industry-standard options, such as layer thickness and size of the print head, as well as implementation specific options such as specifying the parallel processing device to use. Our implementation accepts industry-standard STereoLithography (STL) files as the input mesh type, including support for single STL files with multiple models in it.

Our software's output is an in-memory data structure of line data representing the contour(s) of the input mesh on each layer. Lines are ordered based on their position in the contour, so the next line in the data structure is adjacent to the previous line in the data structure. This output data structure will be added to and eventually converted to machine readable instructions during extension of the software.

The following non-functional requirements guided our design and implementation of the slicer. The primary non-functional requirement was a low total time required for the slicing process. We prioritized computational speed over other performance types, especially memory usage. The slicer uses as much memory as is available to it on the system up to the amount that it needs; it does not limit itself to a set amount less than the total memory of the system.

Wherever practical, we parallelized the slicing process to increase performance. The preprocessing cannot be easily parallelized and requires so few computations that the benefit would be minimal and not worth the development time. The primary slicing calculations are computed using a GPU, parallelizing the problem over hundreds of individual processors. Contour assembly is parallelized on the CPU at the layer level. The user configuration file specifies how many CPU threads the program creates. Those threads then use a queue to efficiently divide the work, each thread accepting a new layer from the queue when it has finished assembling contours on its current layer. The algorithms used in these processes are discussed in more detail in Section 3.2. In our proposal, we said that if we found problems that were well suited for parallelization but that sequential code overtook in real performance, we would provide parallel code as well for use with more powerful future parallel execution hardware. We found no such problems; all parts of our code either execute in trivial time sequentially or are improved by some form of parallelization.

Our slicer's output needed to be accurate enough for industrial use. To that end, we decided to avoid approximating our input data; many slicers reduce accuracy and improve performance by approximating the shape of each layer using fewer lines than specified by in the input mesh. While performance was our primary design concern, we decided that approximating in this way rendered our output partially incorrect and, therefore, did not meet the requirements of the project.

Specific accuracy metrics are discussed in Chapter 4.

The slicer's code needed to be modular and extensible to promote ease of maintenance and facilitate adding new features in the future. The GPU profiles feature allows the slicer to modify its parameters according to the capability of the GPU used. New profiles can be added easily, allowing the slicer to support new GPUs. The slicing process is inherently a data transformation process, and a pipeline design best fits the series of transformations the slicing process entails. A pipeline requires less emphasis on modularity than we originally thought. A single governing class is enough to organize the primary pipeline processes of the slicer. Each conceptual stage of execution is split into its own discrete function, however, and each is well-documented with comments to assist maintenance. Because of the straightfoward nature of the pipeline, new functionality can be appended easily. The high-level architecture for the slicer is illustrated in Fig. 3.1, showing the order and components of the pipeline. The CPU slicing function shown in the figure is out of the scope of this project, but it is accounted for in the design to allow for easy extension in future work.

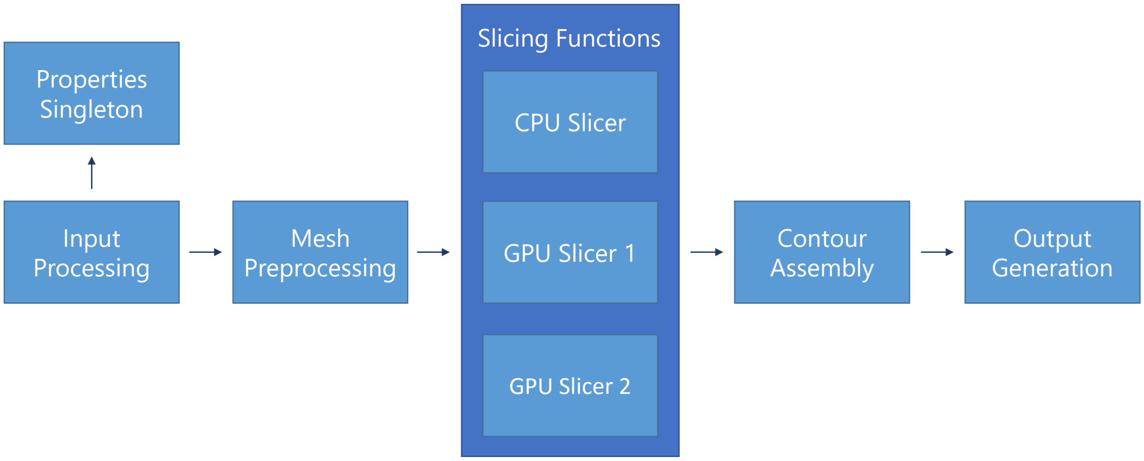

Figure 3.1: Proposed high-level architecture for the slicer application.

Figure 3.2 provides a breakdown of major tasks for the implementation of our slicing application. The percentages shown are estimates of the total project time spent on each activity.

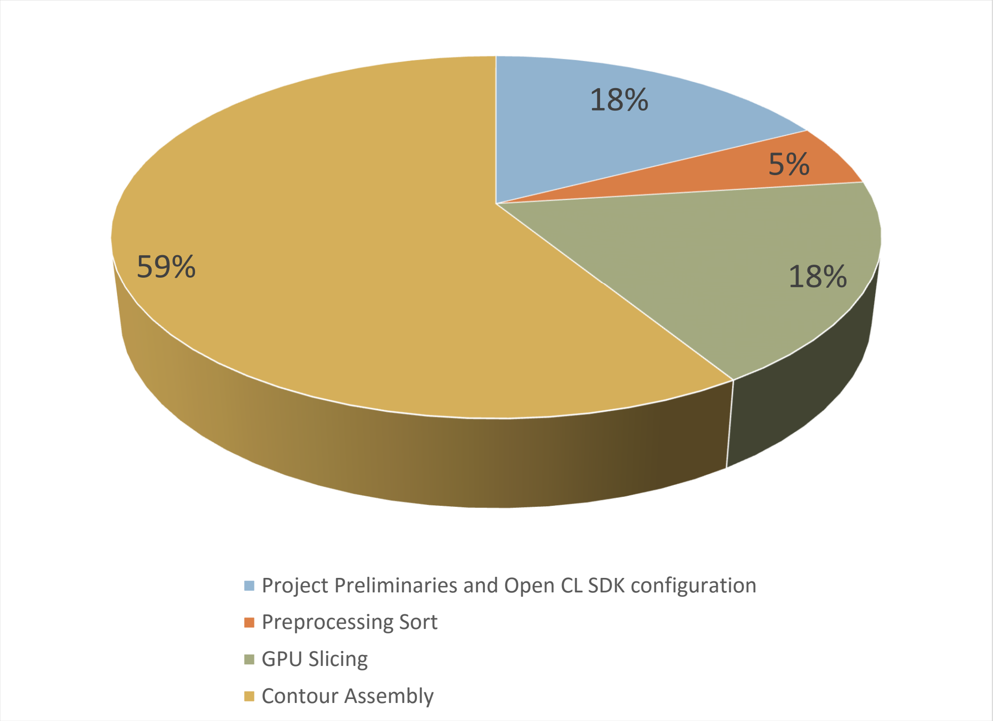

Figure 3.2: Breakdown of development activities by time spent.

3.2 Algorithms

Our slicing implementation relies on three primary algorithms. The first is the preprocessing sort algorithm that creates a list of triangles for each slice that will intersect with that slice. The second is the actual slice function itself, which calculates the intersection lines that comprise the contours. The third algorithm is the contour assembly algorithm, which takes the unsorted lines from the slice function and sorts them into ordered, complete contours.

Before defining the algorithms and providing a running time analysis, we present a definition of the values we use in the algorithms and a lemma needed for the analysis of the algorithms.

Definition 1. *Let n be the number of triangles in the mesh and m be the number of slices a particular run of the slicing algorithm contains. For each triangle n_i, we define the number of slices that it intersects as $slices(n_i)$. We define*

$$\overline{m} = \frac{1}{n}\sum_{i=1}^{n} slices(n_i)$$

Lemma 3.2.1. *Let n, m, and $\overline{m}$ be defined as in Definition 1, then there is no relationship between n and $\overline{m}$. Consequently, both m and $\overline{m}$ can be treated as constants in the analysis of the slicing algorithm.*

Proof. Increasing n does not cause more slices to intersect a given triangle. In general it has opposite effect since more triangles for a given object means smaller triangles in general and a smaller $\overline{m}$. In the worst case, every slice plane will intersect every triangle in a mesh, causing $\overline{m}$ to equal m. However, m depends on the height of the mesh and is not related to n at all. Hence even in the worst case $\overline{m}$ is a constant not related to n. For all but trivial meshes, n is much larger than m. Therefore, m and $\overline{m}$ can be treated as constants in analysis of the slicing problem. □

Algorithm 1 is our preprocessing sort algorithm. It first creates a list of buckets, or storage container lists. Each bucket is a list of the triangles that will produce a line when intersected with that bucket's slice plane. A slice plane is a coordinate on the Z, or vertical, axis that defines an infinite plane on the X-Y axis. Each slice plane corresponds to one of the discrete layers, or slices, that comprise our output data. Next, each triangle is added to one or more buckets unless the triangle is parallel to the slice planes, which results in the triangle being dropped from the triangle list. In order to quickly compute which buckets a triangle belongs in, we calculate two indices, the index of the first bucket that the triangle should be in and the index of the last bucket the triangle should be in. The triangle must also be placed in all buckets between those two indices, so these two indices give us a complete implicit list of all buckets a given triangle should be in. The indices are calculated by converting the minimum and maximum z coordinates of the triangle into integer values by subtracting the starting coordinate of the entire mesh from the current coordinate and dividing by the thickness of the slices. The algorithm then iterates over the range created by the two indices, adding the current triangle to each bucket in the range.

The bucket creation process on line 2 of Algorithm 1 requires an action for each slice and is performed only once, resulting in a time complexity of $O(m)$. The for loop on line 3 requires everything within the loop to execute once for each triangle. Lines 4 through 13 execute in constant time. The loop at line 14 repeats $\overline{m}$ times. Because the loop on line 14 is repeated for each iteration of the loop on line 3, lines 3 through 14 result in a time complexity of $O(n\overline{m})$. Thus, the total time complexity for the algorithm is $O(n\overline{m} + m)$, which simplifies to $O(n\overline{m})$. By Lemma 3.2.1, we can simplify the effective time complexity for the preprocessing algorithm to $O(n)$.

Algorithm 2 is our actual slicing algorithm, the algorithm that calculates the lines created by an intersection of a slice plane and a triangle. The algorithm calculates such an intersection for every triangle in each bucket created during the preprocessing sort. While this is represented by nested for loops in our algorithm,

Algorithm 1: Preprocessing sort.

```
   input  : Triangles $T$
   output : Buckets $B$
1  begin
2      $B \longleftarrow Create\ list\ of\ buckets()$
3      foreach triangle $t$ in $T$ do
4          Calculate min and max z-axis coordinate for current triangle
5          if current triangle is parallel to z-axis then
6              Drop triangle and continue to next triangle
7          else
8              Let $Index_{lowest}$ be the index of the lowest bucket intersecting $t$
9              $Index_{lowest} = \frac{triangle_{min\ Z} - mesh_{min\ Z}}{slice\ thickness}$
10             Similarly we have
11             $Index_{highest} = \frac{triangle_{max\ Z} - mesh_{min\ Z}}{slice\ thickness}$
12             if $index_{highest} == |slices|$ then
13                 $index_{highest} = index_{highest} - 1$
14             for $i = Index_{lowest}$ to $Index_{heighest}$ do
15                 $B[i].Add(t)$
```

lines 4-9 of our algorithm are actually executed as parallel execution threads on the hundreds of processors in the GPU. To calculate the line, three-dimensional line intersections of the slice plane with each of the three line segments that comprise the current triangle are calculated. The points created by these intersections are then checked against the bounds of the triangle. The two intersection points that are within bounds are the two ends of the intersection line. This line is then added to a list of output lines. The normal vector for this line is also calculated and added to the result data. There are special cases, such as when a slice plane passes through a corner vertex of a triangle, creating three intersecting points that are within bounds. Our implementation handles these cases in constant time, having no effect on the time complexity.

The first for loop of Algorithm 2 executes once for every bucket in the list of

Algorithm 2: Slice algorithm.

```
input  : Buckets B
output: Line Segments S
1 begin
2     foreach Bucket b in B do
3         foreach Triangle t in b do
4             Let ContourSegment be a new line segment
5             foreach LineSegment l in t do
6                 Point $p = B.SlicePlane \cap l$
7                 if $p \subset l$ then
8                     $CountourSegment.AddEndPoint(p)$
9             CountourSegment.CalculateNormal();
```

buckets. The nested for loop executes once for each triangle in the current bucket. Together, these loops cause lines 4 through 9 to be executed once for every triangle in every bucket. From Algorithm 1, we know that the total number of triangles in buckets is $n\overline{m}$. Lines 4 through 9 of Algorithm 2 all execute in constant time. Therefore, the time complexity of this algorithm is $O(n\overline{m})$. By Lemma 3.2.1, we can simplify this to a time complexity of $O(n)$.

Our final algorithm, the contour assembly algorithm, is presented in Algorithm 3. This algorithm takes a list of unsorted line segments for each layer and creates complete, ordered contours as output. The algorithm first creates a hash table that we conceptualize as a virtual screen. Each location on the virtual screen is a pixel, a list of references stored at that point. These pixels are more formally called PointLists in the algorithm. Each line segment from the input is plotted on this virtual screen by placing the endpoints of the segment in the corresponding pixel lists, accessed by using f(endpoint's x, endpoint's y) as the key. Once the lines are all plotted, a new path, or list of line segments that comprise a contour, is created. When a complete contour is found, all lines in its path are removed from the virtual screen and added, as an ordered contour, to the output. This process repeats until the virtual screen no longer has any line references in any of its pixels.

Each path is begun by picking a random pixel from the virtual screen and selecting a random line reference within the pixel that becomes our starting point. From the line reference, the location of the other endpoint of this current line is found. This other endpoint is used as a key to find the next pixel in the contour. In this pixel's line reference list are one or more lines that are not the current line used to find the pixel. If only one other line reference is present, that line is the next line for the contour. If multiple lines are present, a process of selecting the next line is used that is omitted here for simplicity. Each line found to be part of the contour is added to the current path. This process of using the current line's other point to find the next line is repeated until the next line selected is the first line of the path. When the next line is the first line of the path, the algorithm has found a complete contour. The path is then run through a postprocessing process where gaps are closed and exactly parallel adjacent lines segments are combined into longer segments. Each line in this path is then added to the final layers data structure as a new contour. As explained earlier, the line references in the virtual screen for each line in the completed path are deleted and a new path is begun by picking another random starting location. This process repeats until there are no more line references in the virtual screen.

The contour assembly algorithm has a significant number of involved edge cases, including gaps between lines, partial contours, overlapping lines, and complex shapes that have many lines on a layer meet at a single point. Our implementation handles all these cases. For the interested reader, an overview of common edge cases can be found in chapter 1 of the netfabb 6.0 User Manual [18].

Algorithm 3: Contour assembly algorithm.

```
input  : Line Segments S
output: Ordered and complete contours C
begin
    Let $V$ be a HashTable with hash function $f(x,y)$
    foreach LineSegment l in S do
        for both endpoints of l do
            if $f(p) \notin V$ then
                V.newPointList(p)
                V.addLine(l)
            else
                V.addLine(l)
    Let Path be a list of LineSegment pointers.
    while !V.isEmpty() do
        Start = V.getPointList()
        CurrentLine = Start.GetLine()
        Path.Add(CurrentLine)
        CurrentPoint = CurrentLine.OtherPoint(Start)
        do
            if only one line reference at current point then
                CurrentLine = only reference
            else
                CurrentLine = LineChooser(CurrentPoint)
            Path.Add(CurrentLine)
            CurrentPoint = CurrentLine.OtherPoint(CurrentPoint)
        while Current!=Start
        PreviousLine = Path.LastLine()
        foreach line l in Path do
            CurrentLine = ContourPostprocessing(CurrentLine, PreviousLine)
            Layer.Add(CurrentLine)
            V.Remove(CurrentLine)
            PreviousLine = CurrentLine
```

It is easiest to analyze the time complexity of the contour assembly algorithm in terms of its input. The number of input lines is equal to the number of triangles in buckets from the preprocessing stage, $n\overline{m}$. Each of these lines is acted upon three times in the algorithm: once when the line is plotted, once when the line is added to the current path, and once when the line is postprocessed and added to the final data structure. Thus, the algorithm's time complexity is $O(n\overline{m})$. By Lemma 3.2.1, we can simplify the time complexity to $O(n)$.

All algorithms in our implementation have a time complexity of $O(n)$ where n is the number of triangles in the input mesh. It is necessary at each stage of the slicing process to involve each triangle at least once; otherwise, triangles and their corresponding computed contour lines would be missing from the output data. Because the output data cannot be correctly calculated without involving every triangle in each stage, the lowest possible time complexity for any of our slicing algorithms is $O(n)$. Therefore, our implementation is asymptotically optimal.

3.3 Final Deliverables

The following materials are available to the School of Computing and Motion View Software LLC:

- Slicer application source code
- Access to project source control at `https://bitbucket.org/cdant/masters-project`
- This report from the McKee Library

Project source code is proprietary property of Motion View Software LLC.

Chapter 4

Testing/Evaluation Plan

A primary goal of our project is to provide fast slicing performance on a wide range of hardware, so modularity and extensibility are key non-functional requirements for our code. Modular design facilitates a system that uses many differing versions of the same functionality. Extensibility allows new hardware to be supported in the future. Originally, we planned on a test-driven design approach with extensive unit testing. During implementation of the project, we realized that we did not know enough about the slicing problem's edge cases for test-driven development to be practical. Instead of unit tests, we have used just-in-time testing when evaluating whether our code is producing the desired result. This approach has yielded similar results: each section of code tested with real-world input. Because the 3D models used in testing are real objects similar to what will be sliced, common intersections between code modules are tested implicitly and every code module is involved in the tests. Some of these tests essentially serve as system tests, testing the interactions of a series of modules that together make up a use instance of the entire system.

Because of the precision required for industrial applications of additive manufacturing, our slicer must create output that precisely represents the initial input object. The planned output format will be a G-code file, so a G-code simulator will be used to visualize and compare the dimensions of the generated G-code to the dimensions of the input mesh. Output dimensions must differ from the original object by less than 2% on each dimension.

The slicer has a varied list of functional requirements, most of which are not co-dependent. Not only must these functions be supported by the slicer, but all functions must be compatible with any other function that is not mutually exclusive to it. Originally, we planned a procedure pipeline, a system that had rules of interaction, ordering, and input/output types explicitly enforced. During development, we realized that once the contour assembly stage was complete, the rest of the process could easily use a single data structure that was added to and modified at each step, rendering input/output checks redundant. We decided that ordering could easily be handled without developing the full-featured procedure pipeline, so we streamlined the process. Using visual simulators created during development, we have verified the functional requirements of the process are being met.

The slicing process is intensive in both data and computation and is highly dependent on input. Stress testing was used to ensure that the slicer can handle the load for real-world input sizes. The slicer fails gracefully when given loads or input that it cannot handle. Failure conditions, such as overly large workloads and malformed input meshes, have been tested to ensure that the software can detect failure, abort execution, and notify the user that a problem has occurred.

Low execution time is a primary goal of our project. For the project to be considered successful, it must perform as well or better than currently available open-source alternatives. For this purpose, we compared our slicer’s total slicing time to that of the Slic3r [1] open-source project. In order for the performance test to be considered successful, our slicer must be at least four times faster than Slic3r. Input models used for this test must be large, as speed differences for trivial models are themselves trivial. In our proposal, we specified that the mesh tested should have at least 5,000 vertices and should be tall enough to require at least 400 slices. The test meshes given to use by Motion View are often only tall enough to require 200 slices. However, they are comprised of over 100,000 faces, meaning the total data size is actually many times larger than we originally specified. Furthermore, the algorithms used in slicing are often better at quickly computing more layers than more data per layer, so trading a lower number of layers for more vertices actually increases the rigor of performance testing. Memory efficiency is not a primary goal of the project, but memory consumption needs to be reasonable enough that it does not significantly impact slicing speed and allows for slicing models of practical size. Thus, the large model specified above must be sliced using no more than 10 GB of memory.

As explained in Chapter 3, the scope of the project was adjusted during development. While the eventual goal of development is still a working slicing tool for Motion View Software, development of the slicing application is still ongoing, and thus, acceptance testing will not be possible.

During development, we created visual simulators to assist in finding and solving problems. These visual simulators also aid us in evaluating our implementation. They allow us to visually confirm that the functional requirements of the project are being met by showing us the output we expect when all features are working correctly. The simulators also allow us to click on a location on the image and receive the coordinates, in the units of the input mesh, of that location. We can use this capability to confirm that our mesh differs in dimensions from the input file by less than the target maximum of 2%.

Figures 4.1 and 4.2 together provide an example of using the simulator to confirm whether a functional requirement is being met. Figure 4.1 shows a visual simulator used during perimeter generation, a process unique to layered manufacturing and part of an extension to our implementation that is still in development. The white lines in this image represent part of the contour created using the three major slicing steps discussed in this paper. A functional requirement for this extension specifies that all perimeter lines must be parallel to a line on the outermost contour. Because the white line between the two red lines is perfectly vertical, its corresponding perimeter line should also be vertical. Figure 4.2 shows that the next perimeter line is far from vertical, meaning this functional requirement is not yet being met.

A summary of tests and corresponding target outcomes is provided in Table 4.1.

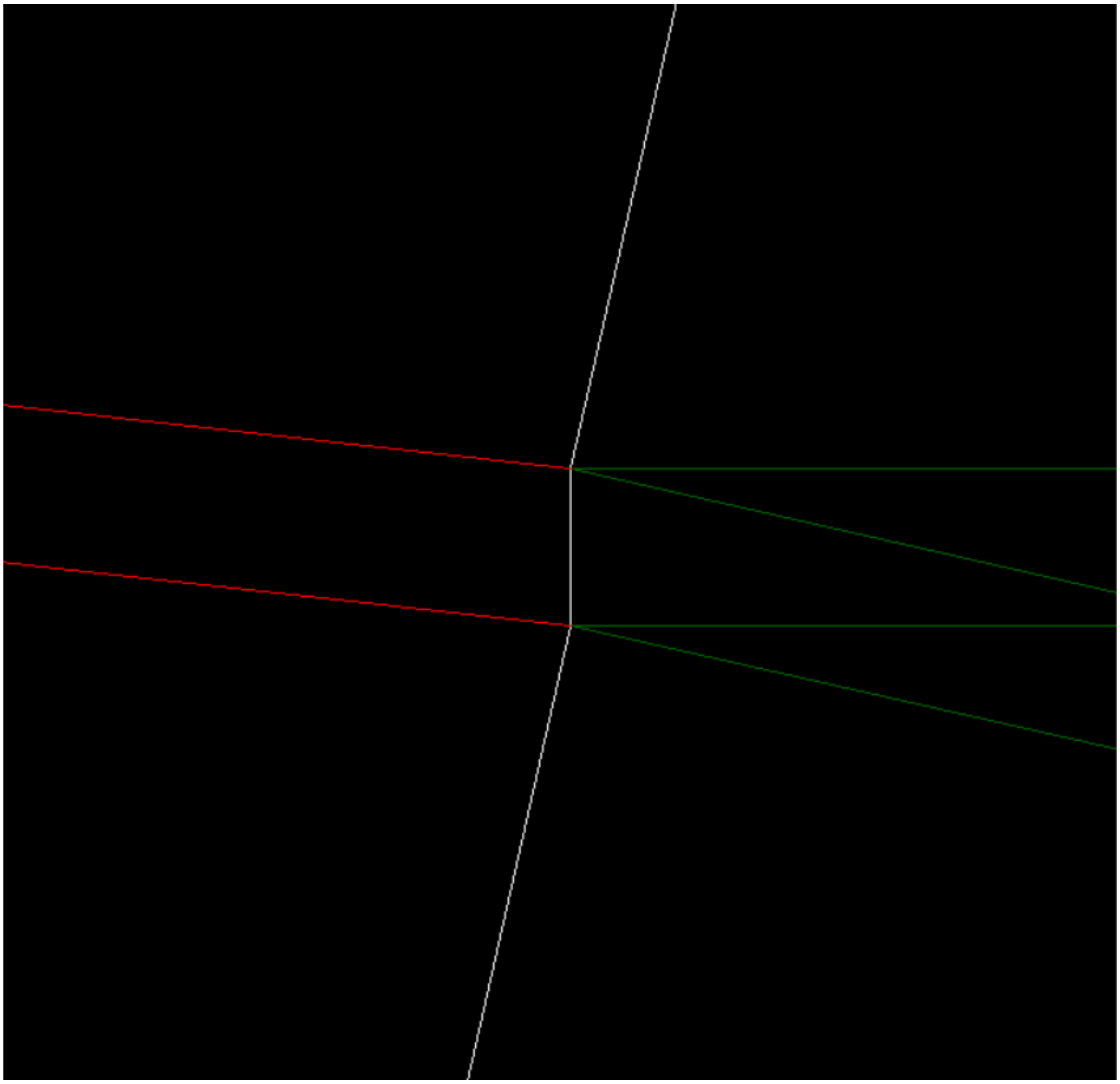

Figure 4.1: Screenshot of a visual simulator showing the outermost contour during perimeter generation.

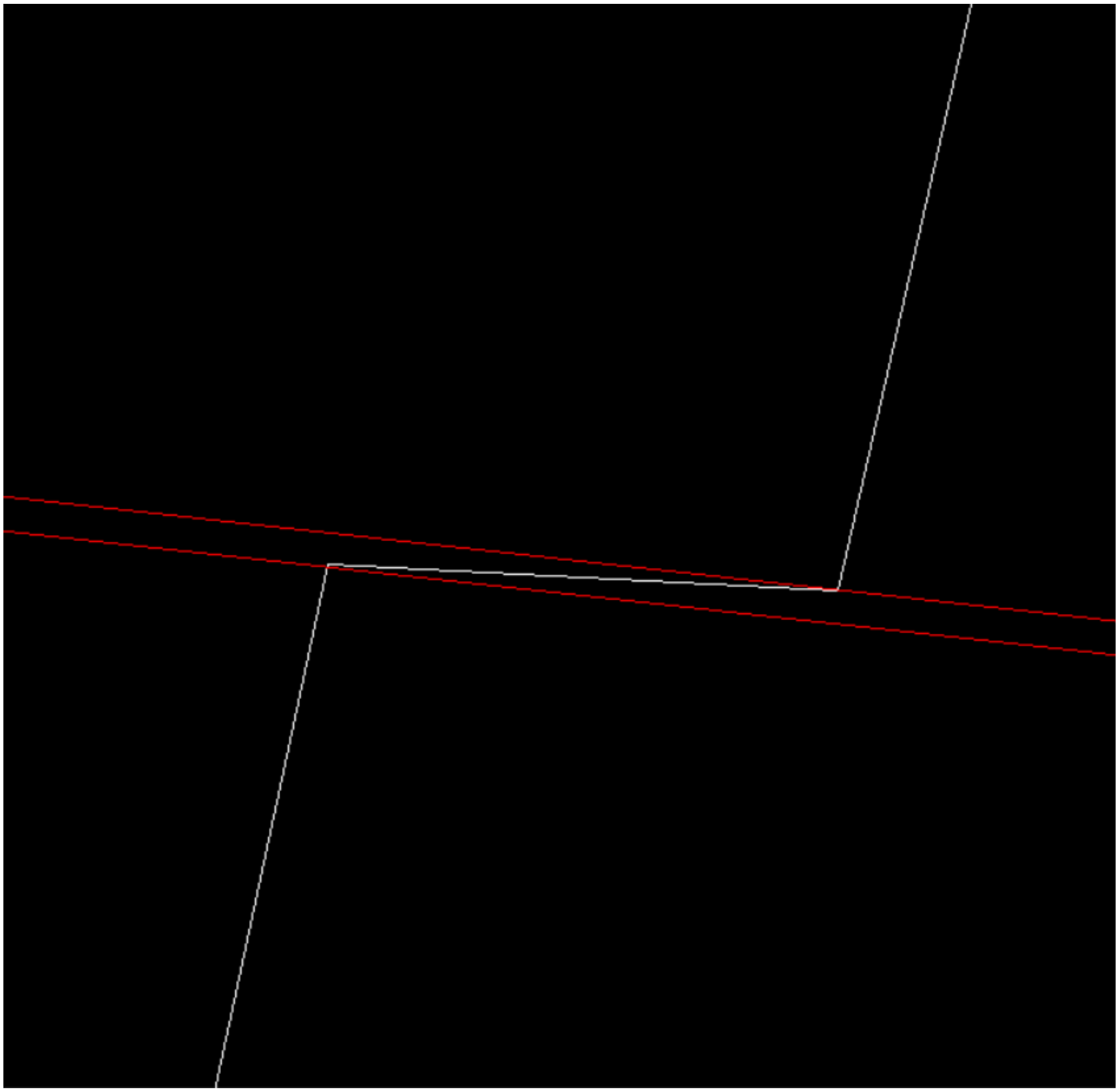

Figure 4.2: Screenshot of a visual simulator showing a perimeter line calculated during perimeter generation.

Table 4.1: Tests and target outcomes.

Test	**Outcome**
Just-in-time testing	Each code module produces correct output for real-world input
Integration	Common interactions tested and each module represented at least once
Accuracy	Output dimensions vary from original object less than 2%
Functional	Functional requirements met
Failure conditions	Failure detected, execution aborted, and user notified
Execution time	Faster than Slic3r project for mesh with >100,000 vertices that results in >200 slices
Memory consumption	Memory usage does not peak above 10 GB for mesh with $\leq$ 100,000 vertices that results in $\leq$ 200 slices

Chapter 5

Results

Using the methods explained in Chapter 3, we built our implementation of a mesh slicer, referred to from here on as Sunder. The required results of our just-in-time and integration testing, as discussed in Chapter 4, is correct, sliced output of an input mesh. Each module must be implemented correctly in at least the general case in order for the module to create correct output. Due to the nature of a pipeline process like slicing, if the modules are not properly integrated, the result will also be incorrect.

The following figures are screenshots of our visual simulators showing the output of Sunder for our two primary test meshes used during development. Figure 5.1 shows the 3D mesh of our first test file, demo.stl. This mesh is a 3D model of a person's teeth and gums. Figure 5.2 shows the first sliced layer of this mesh, the bottom-most slice. The contour displayed in the simulator follows the shape of the base of the input mesh from Fig. 5.1. Figure 5.3 is a close-up view of part of this first slice's contour, revealing the individual lines that comprise the contour. Slightly darker areas seen between lines are not gaps; they are artifacts of the simple Windows drawing API used for the simulators. Figure 5.4 displays a later slice from the demo mesh output on which the outlines of individual teeth can be seen. Figure 5.5 shows the 3D mesh for our second test file, 20arch.stl. This file is comprised of the demo mesh rotated to be standing upright and copied twenty times. Figure 5.6 shows a slice near the base that contains 40 contours, two for each of 20 arches. Figure 5.7 shows a much higher slice where the two sides of each arch have combined into one contour that shows the beginning of canine teeth. From extensive visual testing using the simulators, we conclude that we are producing correct slicing output that matches the input meshes.

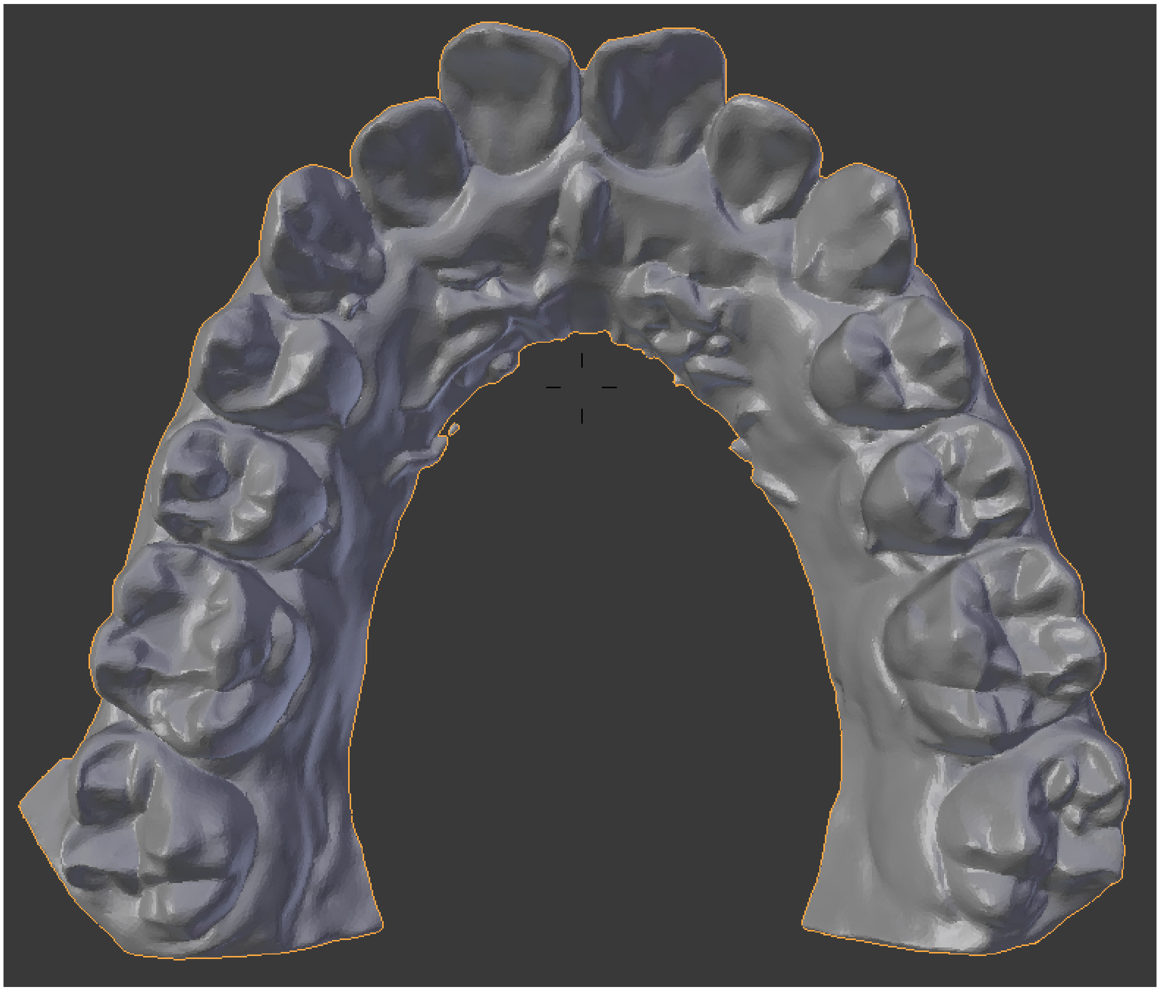

Figure 5.1: Screenshot of the demo.stl mesh displayed in the open-source modeling software Blender.

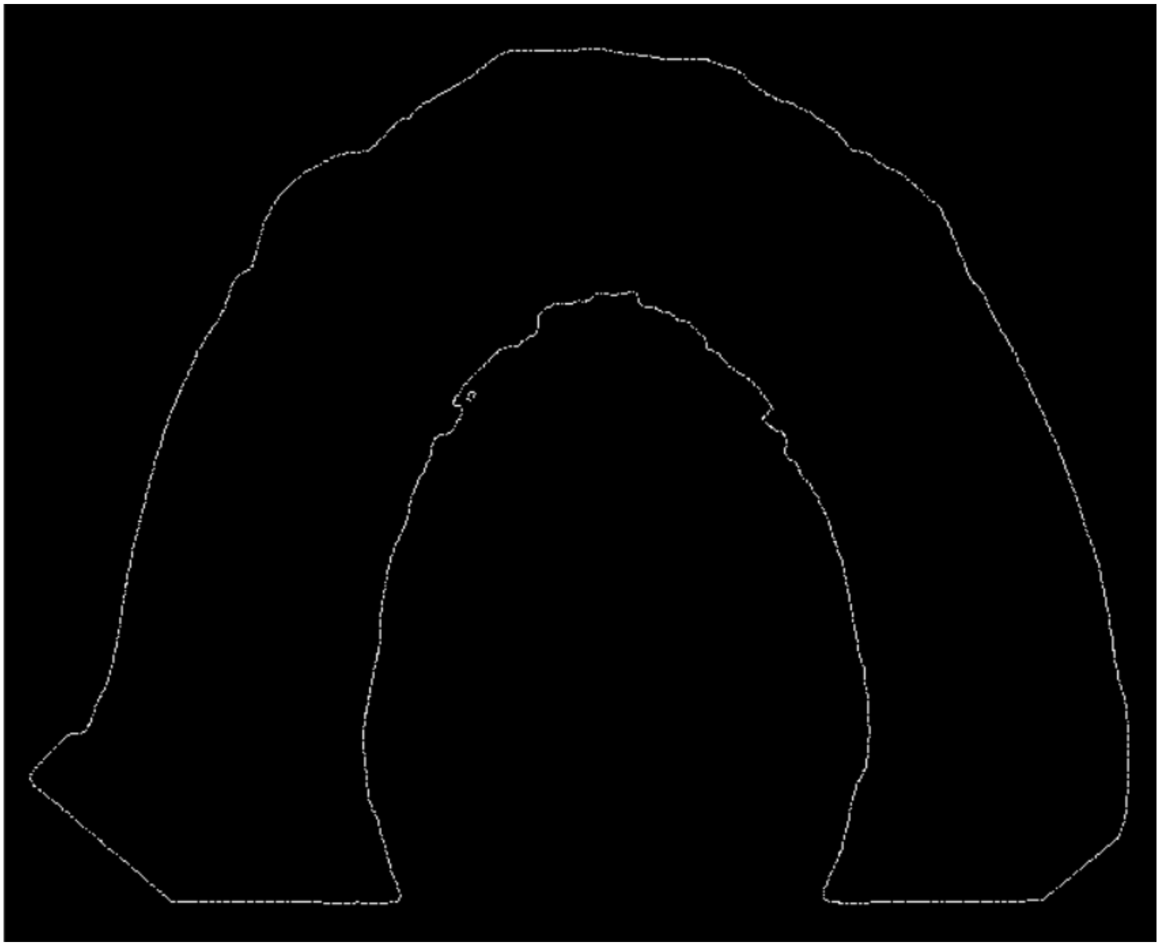

Figure 5.2: Screenshot of the first slice of the slicing output from the demo mesh.

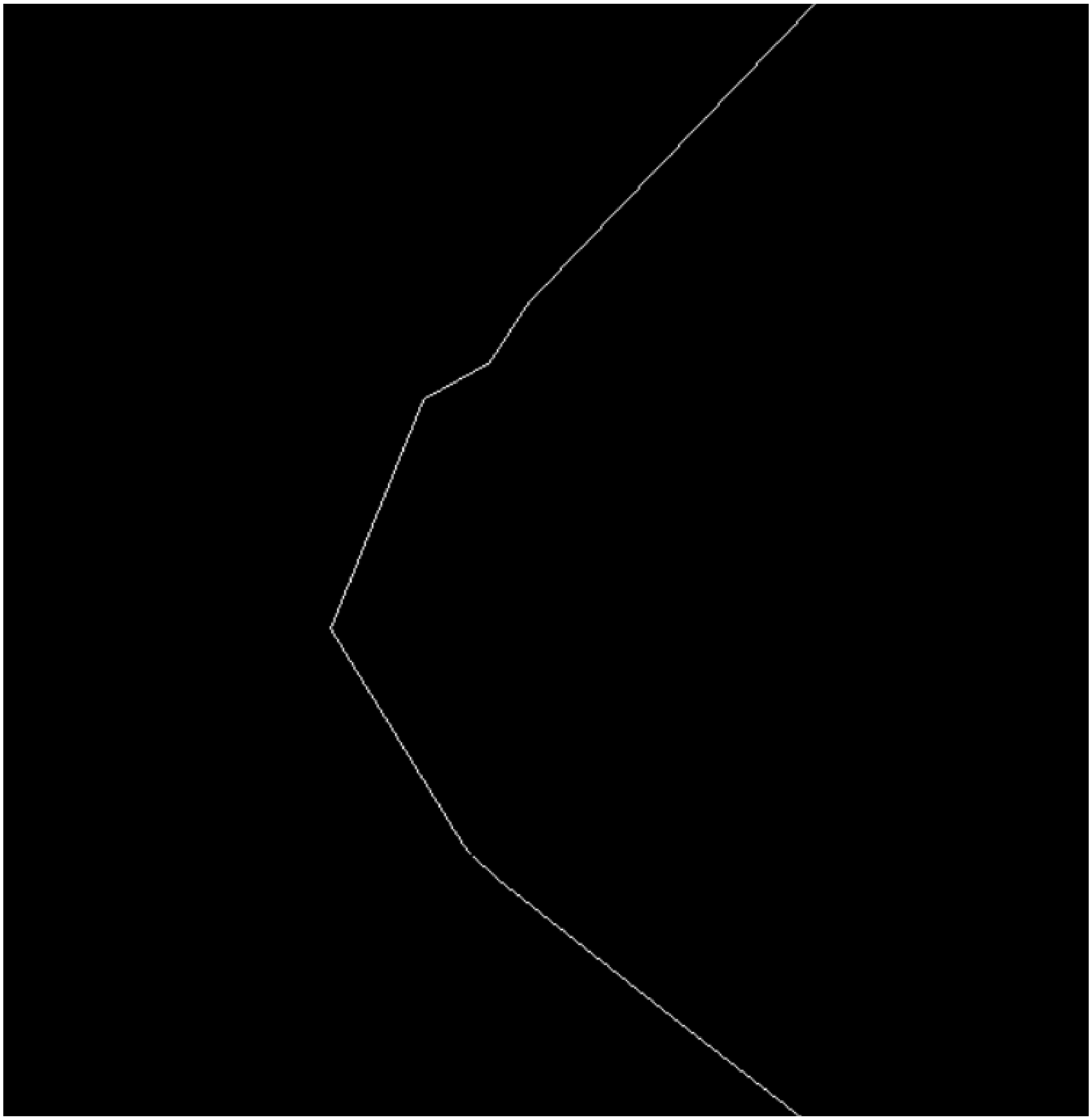

Figure 5.3: Zoomed screenshot of part of the first slice from the demo mesh.

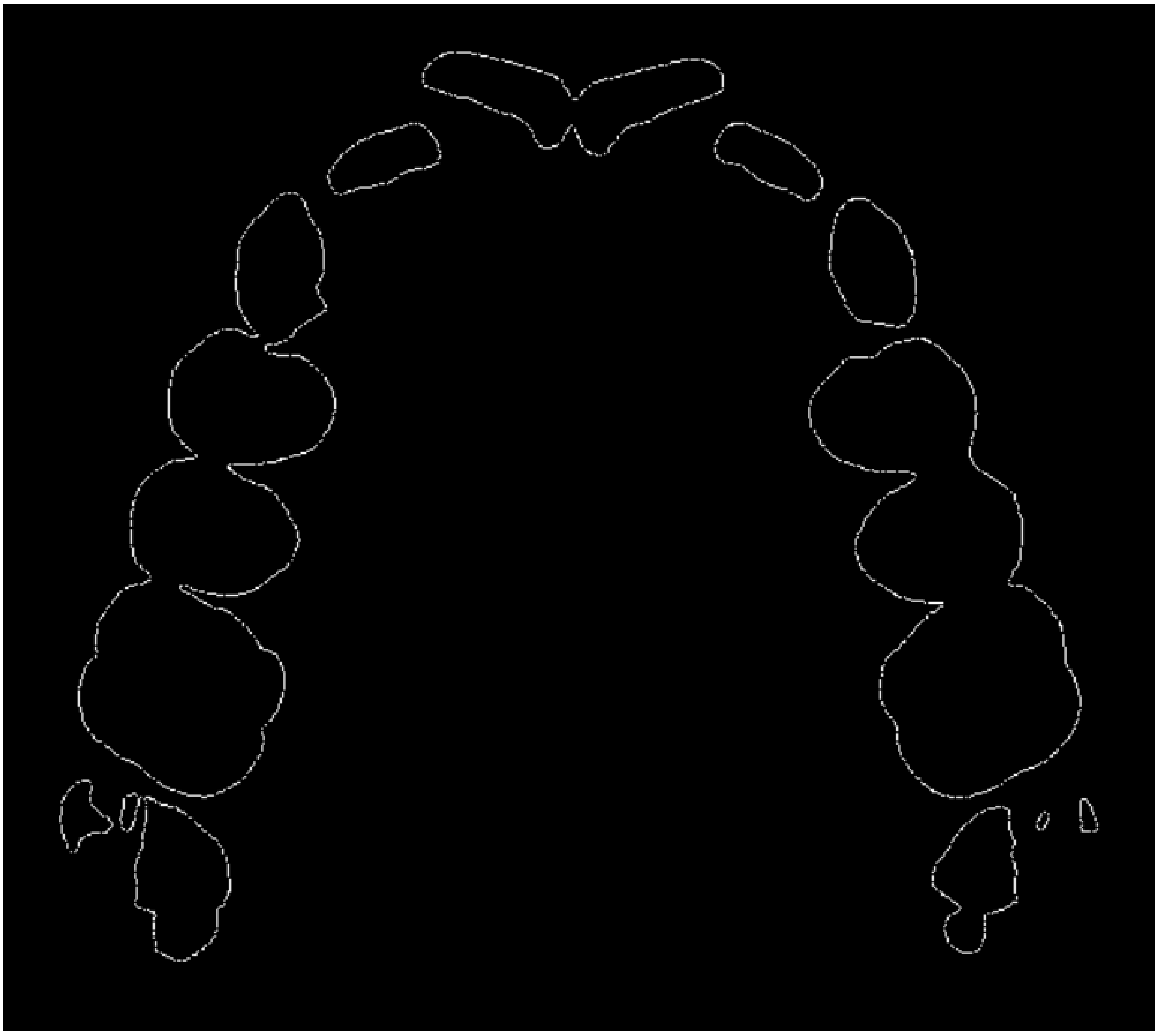

Figure 5.4: Screenshot of a higher slice from the demo.stl mesh.

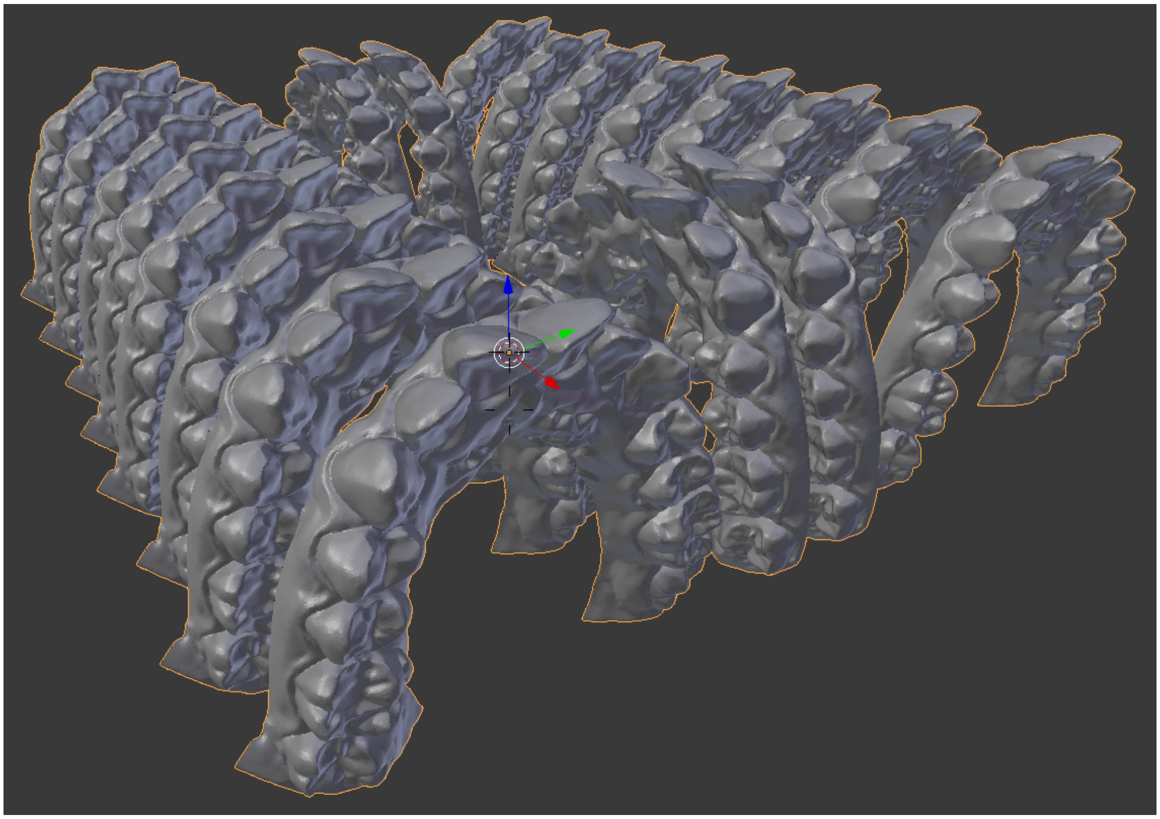

Figure 5.5: Screenshot of the 20arch.stl mesh displayed in Blender.

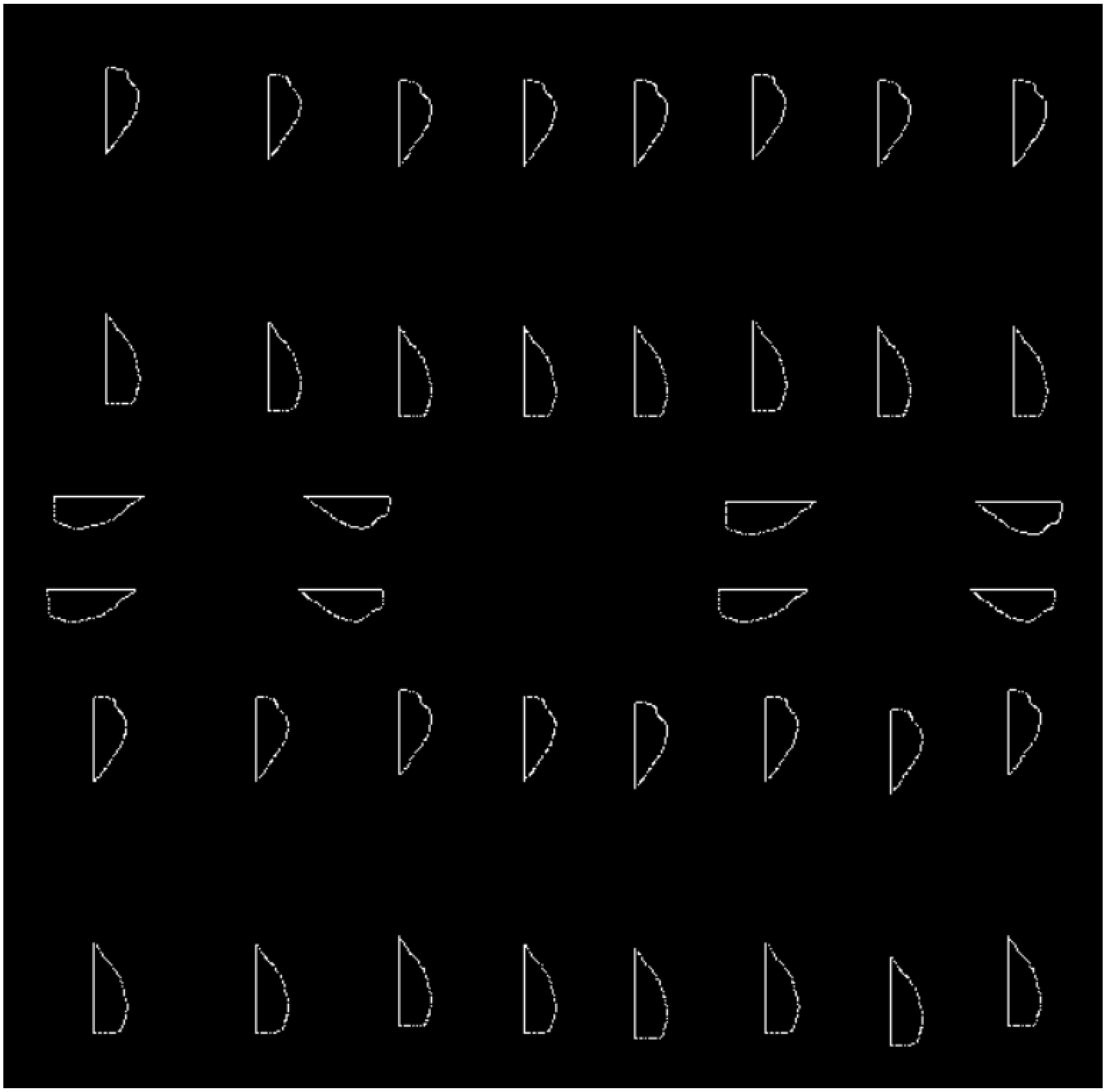

Figure 5.6: Screenshot of a slice near the base of the 20arch mesh.

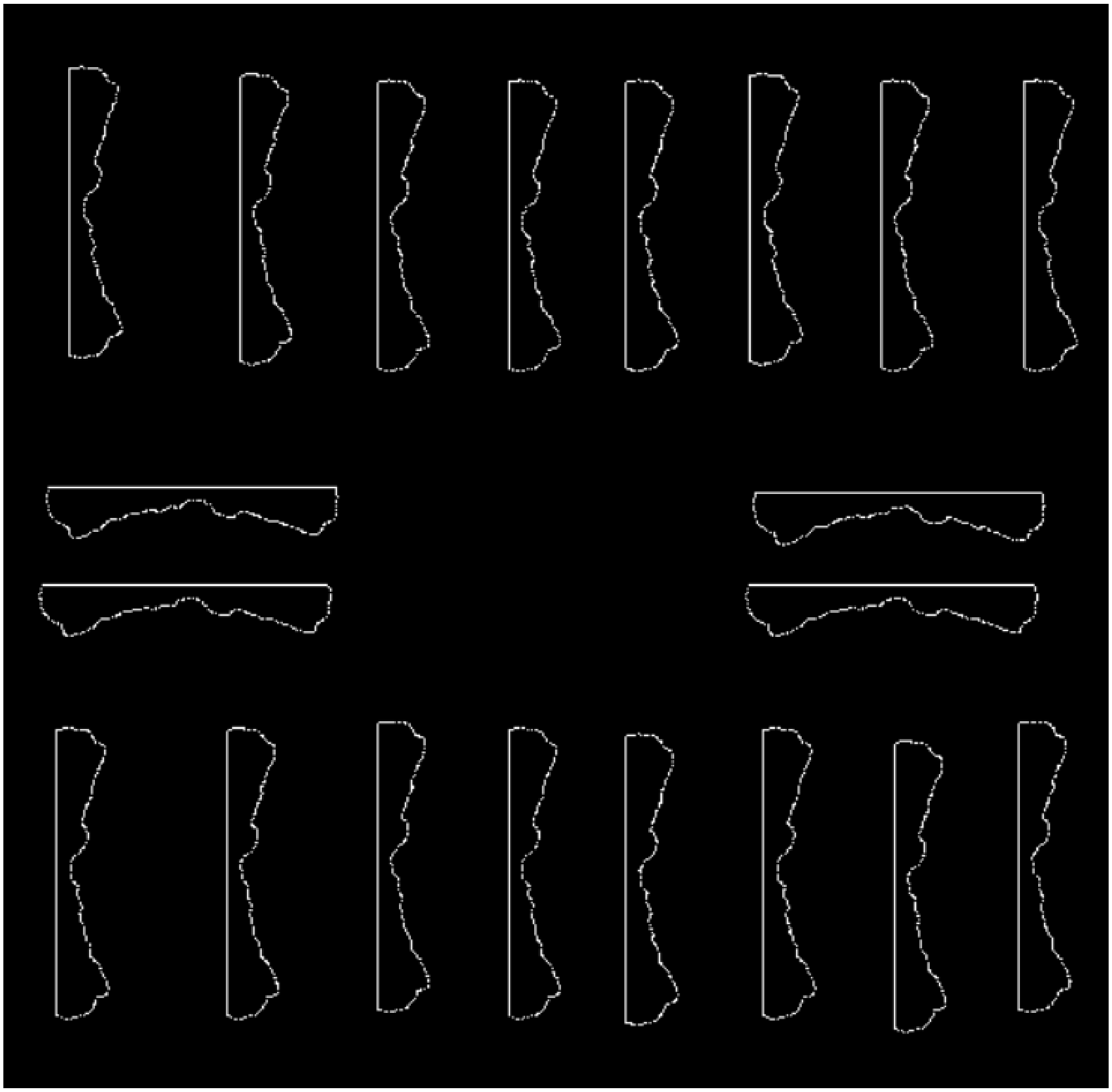

Figure 5.7: Screenshot of a slice near the top of the 20arch mesh.

These screenshots all show output lines calculated during the second stage of our implementation, the actual slicing step. The preprocessing step has no data that can be easily visualized. However, it is the preprocessing step that creates the canonical list of triangles that should be checked against each slice plane. Each entry in this list results in one of the output lines, so if the preprocessing step was incorrect, either extraneous lines or gaps would be evident in the displayed output.

The output of the final stage, contour assembly, is difficult to visualize. If the contour assembly stage produces correct output, the result will look identical to the figures shown thus far. The difference is that the data is no longer in the form of an unordered, potentially unrelated list of lines; instead, the lines are grouped in ordered contours that each make a complete, gapless loop. We inspected a representative sample of the data using our development environment's debugger and confirmed that the endpoint of any given line was the exact starting point of the next line to the full precision allowed by single-precision floating point values. In the visual simulator for the contour assembly process, correctly created contours are displayed in white and all other lines from the slicing stage that have not been included in a contour are displayed in red. Figure 5.8 shows a slice of the demo mesh after contour assembly has been completed. This particular slice features two red lines, intentional artifacts generated by handling of an edge case during the slicing stage on the GPU. Because these artifact lines are removed during contour assembly, they are displayed in red. The rest of the contour is displayed in white, showing that the contour was created correctly.

Our evaluation plan specified that the slicer output should differ from the original input by no more than 2% in any dimension. STL files use single precision floating point values, as does all components of our slicer, meaning no avoidable data loss due to precision has occurred. The precision of a single precision floating point value is such that errors due to precision will be far less than 2% of any original value. Our implementation never approximates lines or contours and rounds values in exactly one location. During contour assembly, if a gap is found between the endpoints of two lines and that gap is less than one micrometer in length, then one of the endpoints is moved so that the gap is bridged without the creation of a new line. That method of rounding means that any dimension could

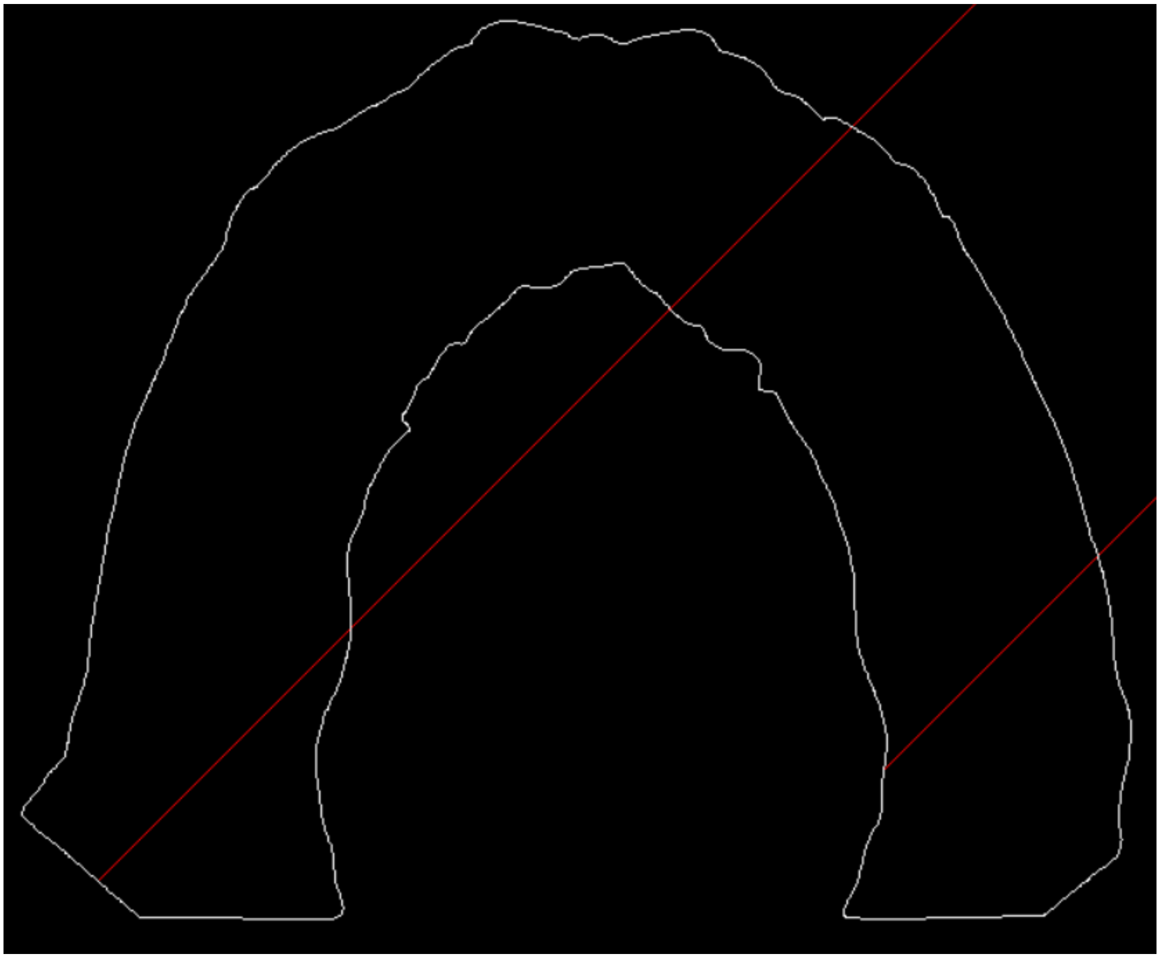

Figure 5.8: Screenshot of a slice from the contour assembly stage showing red lines.

be inaccurate by up to one micron. If such an inaccuracy existed on either side of a contour, cross section of a contour at a given point could be inaccurate by up to two micrometers. Two micrometers is 2% of 100 micrometers, so any object with dimensions larger than 100 micrometers will have less than 2% inaccuracy on any dimension. Considering that the nozzle diameter, the width of the lines that comprise a 3D printed object, of Motion View's laser-based 3D printer is approximately 100 micrometers, objects smaller than that are not only impractical, but technically improbable given existing hardware. Therefore, we conclude that we have met our accuracy requirement for all practical inputs.

We implemented Sunder as a Windows console application according to the specifications laid out in Chapter 3. We accept STL input files and a configuration file and create an in-memory representation of complete, gapless contours as our output. The configuration file supports industry-standard parameters such as nozzle width and layer height, as well as implementation-specific parameters such as choice of acceleration device, number of CPU threads to create for parallelization, and whether to display the visual simulators used during development. Figure 5.7 shows that our implementation can slice meshes containing multiple discrete objects and combine or separate contours as necessary to best represent the input mesh. Therefore, we have met the functional requirements for the project.

Sunder properly handles failure conditions, as specified in our evaluation plan. Modern C++ techniques have allowed us to safely use an exception-driven error handling system in our code without causing memory leaks or other similar resource locks. Our code actively identifies known potential problems and throws an exception if such an issue is detected. When an unexpected exception occurs, the same exception-processing code handles it. Either way, all resources used in the program are properly released and the program is gracefully terminated. The user is shown the text of the exception in the console window as well as a suggestion to check the log file for more detailed information on the failed execution attempt. Thus, Sunder meets our goals for failure handling.

The primary nonfunctional requirement of this project was that our implementation must be four times faster than the Slic3r slicer when using a mesh for input that contained over 100,000 vertices and which results in over 200 slices. Both of our test meshes meet these requirements. Table 5.1 shows a comparison of execution times for Sunder and Slic3r slicing the test meshes. All tests were done

Table 5.1: Execution time results and comparison.

Slicer	Mesh	Time in seconds	Comparison factor
Slic3r	demo.stl	2.56	1
	20arch.stl	165.34	1
Sunder	demo.stl	1.62	1.58
	20arch.stl	5.05	32.74

on the same system with the tests immediately following each other, so external factors on the Windows system are approximately equal. The times from Sunder were timed using a stopwatch class in the code that reported the elapsed time in milliseconds. Slic3r only reports time for the entire slicing process including extensions specific to 3D printing, but it does output a statement indicating when it has completed each stage of the process. Thus, the times from Slic3r are timed using a physical stopwatch and based on when it reports the slicing steps completed. All time values are an average of multiple timings to increase accuracy and compensate for minor changes in outside factors. The comparison factor column is a multiplicative value representing the factor by which Sunder is faster than Slic3r. Thus, the comparison factors for Slic3r itself are one. Both slicers sliced the demo mesh in negligible time, fast enough that timing Slic3r physically was difficult to do accurately. When slicing the demo mesh, Sunder was only 1.58 times faster than Slic3r, short of our original goal. However, when the larger 20arch mesh is sliced, Slic3r requires 165 seconds while Sunder finishes in just over 5 seconds. Sunder is, therefore, over 32 times faster than Slic3r when slicing 20arch, over 8 times our original performance goal. Our requirements stated that Sunder should be over four times faster for a mesh that is larger than 100,000 vertices and 200 slices. Since 20arch meets those requirements, we have met our performance goal, despite the fact that Sunder is not over four times faster for every mesh of the required size. Because the demo mesh slicing time was so low for both slicers and because the time Slic3r requires increases so quickly for a larger input, we suggest that Sunder meets our performance goals for any mesh large enough that the slicing time becomes a barrier to real-world application of the slicer.

Interestingly, Sunder did not require approximately twenty times as long to slice the 20arch mesh as the demo mesh, despite the fact that the demo mesh is exactly twenty times the data. Our algorithms have linear time complexity, so anything less than a linear increase implies some other factor is present in a meaningful way. We believe that this sublinear increase indicates that a significant portion of the execution time currently observed for Sunder is due to constant-time overhead. Therefore, we would need to continue to increase the input size before we would see an approximately linear increase in execution time. However, the 20arch mesh already strains the memory capacities of the GPU and consumes 20% of our memory goal. Therefore, we believe that our time-based optimization of the slicing process has been so effective that we have shifted the focus back to memory usage as the most important factor preventing the slicing of larger and more complex objects. Slic3r's quick increase in execution time with a larger data size, greatly exceeding a linear increase, also indicates that our linear algorithms are superior to those employed by some slicers still used in industry.

Our final evaluation metric is memory consumption. We specified that we must use no more than 10 GB of memory for a mesh of 100,000 vertices and 200 slices. Sunder consumes about 243 MB of memory when slicing the demo mesh and 2.2 GB of memory when slicing the much larger 20arch mesh. Therefore, we have met our memory consumption goals.

Chapter 6

Conclusion

Slicing is a process vital to important problems in computational geometry, including additive manufacturing. Additive manufacturing is rapidly gaining popularity, and more and more fields are finding ways to take advantage of the rapid manufacturing it allows. In order to facilitate time-sensitive uses of additive manufacturing and other problems that involve slicing, the slicing process must be as fast as possible. Hardware acceleration techniques using GPUs and other parallel execution hardware are underutilized by current slicing software, leaving significant potential for faster performance. In cooperation with Motion View Software, we have developed Sunder, a slicer that utilizes new algorithms that are asymptotically-optimal in running time and parallel execution techniques to dramatically improve slicing performance. This allows slicing to be applied to increasingly large and complex problems. Our future work will extend Sunder into a full slicing platform for 3D printing, bringing new algorithms and faster parallelization to the rest of the 3D printing process.

Bibliography

[1] Slic3r: G-code generator for 3d printers. [Online]. Available: http://slic3r.org/ 1, 4

[2] Simplify3d. [Online]. Available: https://www.simplify3d.com/software/ 1

[3] D. Liao, "Gpu-accelerated multi-valued solid voxelization by slice functions in real time," in *Proceedings of the 24th Spring Conference on Computer Graphics*. ACM, 2008, pp. 113–120. 1, 2.1

[4] H.-H. Hsieh, Y.-Y. Lai, W.-K. Tai, and S.-Y. Chang, "A flexible 3d slicer for voxelization using graphics hardware," in *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM, 2005, pp. 285–288. 1, 2.1

[5] C. Kirschman and C. Jara-Almonte, "A parallel slicing algorithm for solid freeform fabrication processes," *Solid Freeform Fabrication Proceedings, Austin, TX*, pp. 26–33, 1992. 1, 2.1

[6] P. Mohan Pandey, N. Venkata Reddy, and S. G. Dhande, "Slicing procedures in layered manufacturing: a review," *Rapid prototyping journal*, vol. 9, no. 5, pp. 274–288, 2003. 2.1

[7] K. Tata, G. Fadel, A. Bagchi, and N. Aziz, “Efficient slicing for layered manufacturing,” *Rapid Prototyping Journal*, vol. 4, no. 4, pp. 151–167, 1998. 2.1

[8] S. Choi and K. Kwok, “A tolerant slicing algorithm for layered manufacturing,” *Rapid Prototyping Journal*, vol. 8, no. 3, pp. 161–179, 2002. 2.1

[9] S. Choi and F. Kwok, “A memory efficient slicing algorithm for large stl files,” in *Proceedings of Solid Freeform Fabrication Symposium*, 1999, pp. 155–162. 2.1

[10] M. Vatani, A. Rahimi, F. Brazandeh, and A. S. Nezhad, “An enhanced slicing algorithm using nearest distance analysis for layer manufacturing,” in *Proceedings of World Academy of Science, Engineering and Technology*, vol. 37, 2009, pp. 721–726. 2.1

[11] R. M. Gregori, N. Volpato, R. Minetto, and M. V. Da Silva, “Slicing triangle meshes: An asymptotically optimal algorithm,” in *Computational Science and Its Applications (ICCSA), 2014 14th International Conference on*. IEEE, 2014, pp. 252–255. 2.1

[12] X. Huang, Y. Yao, and Q. Hu, “Research on the rapid slicing algorithm for nc milling based on stl model,” in *AsiaSim 2012*. Springer, 2012, pp. 263–271. 2.1

[13] S. McMains and C. Séquin, “A coherent sweep plane slicer for layered manufacturing,” in *Proceedings of the fifth ACM symposium on Solid modeling and applications*. ACM, 1999, pp. 285–295. 2.1

[14] M. Feldman. (2012, February) Opencl gains ground on cuda. [Online]. Available: http://www.hpcwire.com/2012/02/28/opencl_gains_ground_on_cuda/ 2.2

[15] C. Zeller. (2011) Cuda c/c++ basics. [Online]. Available: http://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf 2.2

[16] C. Woolley. Introduction to opencl. [Online]. Available: http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf 2.2

[17] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008. 2.2

[18] netfabb. (2015, June) Netfabb 6.0 user manual. [Online]. Available: http://www.netfabb.com/manuals_download.php?nid=&fid=83 9

[19] M. Reddy, *API Design for C++*, T. Green, Ed. Morgan Kaufmann Publishers, 2011.

[20] Reprap flavor g-code. [Online]. Available: http://reprap.org/wiki/G-code