A New Polygon Nesting Trees Data Structure to Construct a Physical Polyhedral Object

by

Patrice Arruda

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario, Canada
September 2013

Abstract

Layered manufacturing is commonly used to produce three-dimensional objects for rapid prototyping of new products, repair manufactured discontinued products or for research purposes. An object (polyhedron) is created by fusing a set of printed layers. Each layer is created by depositing fusing material to an enclosed area with a fixed thickness. The enclosed area is bounded by a polygon which may contain holes. These holes are also bounded by a polygon. That said, a polygon P is the parent polygon of Q if Q is inside of P. This parent child polygon relationship is organized into an acyclic graph (nesting tree) which represents a layer of the polyhedron. The printer uses the tree to determine which solid part of the layer is needed to be printed. Each node of the tree represents the position of the printer's nozzle to where the nozzle drop the fusing material.

We present a new data structure known as Polygon Nesting Tree Data Structure that has the capacity to output a polygon nesting tree at the given height h of a polyhedron. The data structure can produce a set of polygon nesting trees which is then sent to the printer to produce the object. The thickness of a layer is controlled by composing a number of nesting trees where a higher number of nesting trees would improve the resolution of the polyhedron. Furthermore, we show some results of the implemented polygon nesting tree data structure slicing several polyhedra. The newly developed data structure can be used to solve other related problems.

Acknowledgments

I would like to sincerely thank my supervisor Dr. Michiel Smid for providing me guidance and support throughout the development of this thesis. His expertise assisted me to the difficult phases of my masters studies.

To my family and friends, thank you very much for encouraging me and giving me the strength when I needed it the most. It was quite a difficult journey of studying and working full time. A special thanks goes to my sister Sonia Arruda who helped correct this thesis paper.

I wish to thank Dr. Jit Bose and Dr. Pat Morin for encouraging me to join the computational geometry field. Throughout the years, I have learned quite a lot in terms of geometric data structures and algorithms which subsequently helped to develop several patents in my professional career.

Furthermore, I would like to thank the School of Computer Science, the members and peers of the Computational Geometry Lab and the Faculty of Graduate Studies at Carleton University.

Table of Contents

List of Tables

List of Figures

List of Acronyms and Symbols

ρ	The slicer plane.
P or Q	A Polygon.
G	A Graph.
$a \times b$	Cross product of vectors a and b.
∂P	The boundary of polygon P.
$\mathcal{P}$	A Polyhedron.
$\mathcal{D}$	Doubly Connected Edge List Data Structure.
$\mathcal{T}$	Polygon Nesting Tree Data Structure.
T	A Polygon Nesting Tree, usually created from $\mathcal{T}$.
h	A requested height of $\mathcal{P}$ to be sliced at h to produce T.
ϖ	Sorted Vertices Array.
D	A disk.
L_D	A generic list of disks.

HT_D	A hash table containing a set of disks. The key of each disk is a unique identifier which is usually randomized.
HT_e	A hash table containing a set of edges that belonging to $\mathcal{P}$.
HT_v	A hash table containing a set of vertices that belonging to $\mathcal{P}$.
$EP(e, v)$	The end vertex of edge e where it returns the opposite vertex v.
$NOOP$	A vertex marked as NOOP (No operation) because it does not contain information to modify $\mathcal{T}$.
$\alpha(v_x)$	An operation to retrieve a set of edges not lying on ρ, containing v_x as an end-vertex.
f	A face from $\mathcal{P}$.
r	A ray shooting from a fixed point to a random destination vertex.
v_b	A vertex from $\mathcal{P}$ marked as a begin vertex.
v_m	A vertex from $\mathcal{P}$ marked as a mixed vertex.
v_e	A vertex from $\mathcal{P}$ marked as an end vertex.

Chapter 1

Introduction

1.1 Motivation

Layered manufacturing (also known as three-dimensional printing or additive manufacturing) is the process of producing a three-dimensional (3D) object from a digital file by printing it layer by layer. A layer of an object is simply the intersection of an object with a horizontal plane at a particular z-coordinate. A layer usually has a thickness of 0.05 inch and is dependent on layering manufacturing technology. All the layers are fused together to form the final product. This manufacturing process became more popular in recent years for rapid prototyping [1]. Rapid prototyping helps the user to design new transportation vehicles and household appliances or simply reconstruct a damaged object that has been discontinued from production. The traditional ways of producing prototypes or custom parts are very expensive, time consuming and labour-intensive. It may take several weeks to calibrate the machine properly in order to produce a custom-made part. With layered manufacturing, it takes several hours to create an object. Also, obsolete or damaged objects can be reconstructed. For instance, Jay Leno [2] owns a 1907 white steam car. The d-valve engine part became damaged, and he sought to replace it. However, finding an obsolete d-valve component of a vehicle made a century ago is very difficult, for most

manufacturers ended their productions a long time ago. Jay decided to take the part to a manufacturer who uses layer manufacturing technologies. The operator scanned the broken d-valve part, reconstructed the object using special software and then sent the digital object to the printer. Within 24 hours, Jay Leno had an exact replica of the d-valve part which was then installed in the engine bay.

There is a large set of technologies that can print a 3-dimensional object. One of the oldest methods is Stereolithography [1]. Stereolithography was introduced in 1988 and probably remains the best known method. Imagine a container filled with photopolymer liquid. Within this container is a small platform that moves downwards. The platform is below one layer thickness of the photopolymer liquid. An ultraviolet laser is then shot to the liquid, hardening it on contact. The laser (x, y) position is controlled by the movement of the reflective mirror. Once the layer of the object has been constructed, the platform moves one layer down and repeats the process. The object may have concavities; hence supports may need to be included to prevent cantilevered features. Once the object is fully constructed, the supports of the object are removed, and the object is sanded down to have a smooth surface. See Figure 1 for a basic overview of stereolithography.

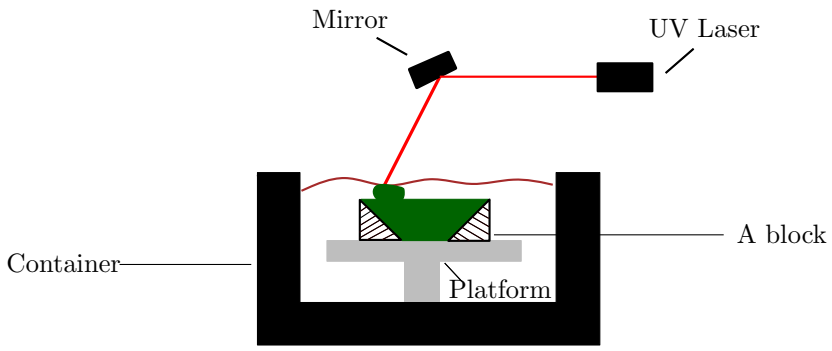


Figure 1: Basic visualization of stereolithography at work. The container is filled with liquid photopolymer and the object is supported by two blocks. The current layer is being hardened by the UV laser.

Another technique used to print an object involves the use of powder. Powder-based 3D printing was introduced at MIT in the late 1980s. The idea is that a thin powder is spread throughout a surface, and the layer is formed by depositing drops of binder from an inkjet print head. The main advantage of this technique is that the object can be coloured.

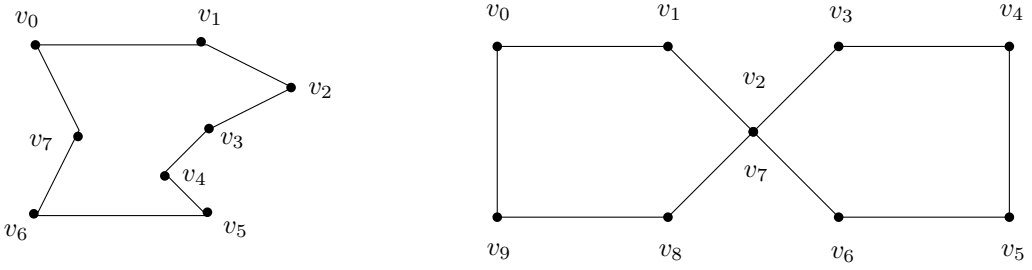
Lastly, thermoplastic-fused-deposition modeling is another approach to producing objects. A layer is generated by feeding a line of plastic through a nozzle. The nozzle melts the plastic, which is then extruded to the surface. This approach is powerful as multiple independent nozzles can be used to print several sections of the layer, thereby speeding up the printing time. This type of printing technology has revolutionized layer manufacturing since it provides a cheaper method of printing 3D objects to professionals and hobbyists.

The most important part of producing objects is the amount of time spent on the pre-processing and post-processing stages. As production time increases, the cost of the object increases. Thus, reducing the amount of time on pre-processing stage can indeed reduce the cost of the object. Regardless of which layer manufacturing technology is used to print the object, the thickness of the layer needs to be determined. Today, the thickness of the layers are usually 0.05". If the layer is more thick than the usual, the post-processing time takes longer since there are more sanding and detailing work to be completed. If the thickness of the layer was incorrectly determined and the object was printed, the object may need to be re-printed again. In an effort to reduce the pre-processing time and to avoid major human errors, it becomes essential to optimize the algorithms and data structures used to help the printers to print an object.

1.2 Preliminaries

Before going into details of the problem statement and related work, this section contains several definitions which are used throughout this thesis. We describe the nature of a polygon, polyhedron, and nesting tree as well as explain the techniques used to construct of a polygon nesting data structure. Refer to the symbols section for clarification of a symbol.

Definition 1.2.1. *A simple polygon P is the closed region of a plane bounded by a finite set of segments forming a non-intersecting closed curve C [3]. A segment is an edge e and the endpoints of e are called vertices. The boundary ∂P of the polygon is defined as the collection of its edges and vertices.*



A Simple Polygon

A Weakly Simple Polygon

Figure 2: The difference between a simple polygon and weakly simple polygon. v_2 and v_7 share the same (x, y) coordinates for the weakly simple polygon.

It is possible that a vertex $v_i = v_j = (x, y)$ in ∂P where $i \neq j$, as shown in Figure 2. In this case, the polygon is called a weakly simple polygon.

Definition 1.2.2. *A polyhedron $\mathcal{P}$ is a closed surface of faces in three dimensional space. We qualify a polyhedron $\mathcal{P}$ as a polyhedron consisting of triangular faces. Each face is composed of three vertices and edges. Each edge of $\mathcal{P}$ belongs to exactly two faces. The graph consisting of the vertices and the edges of $\mathcal{P}$ does not contain cut-vertices.*

The cut-vertex statement ensures that the polyhedron does not separate into two polyhedra when a cut vertex is removed. This statement is to avoids eccentric cases as shown in Figure 3.

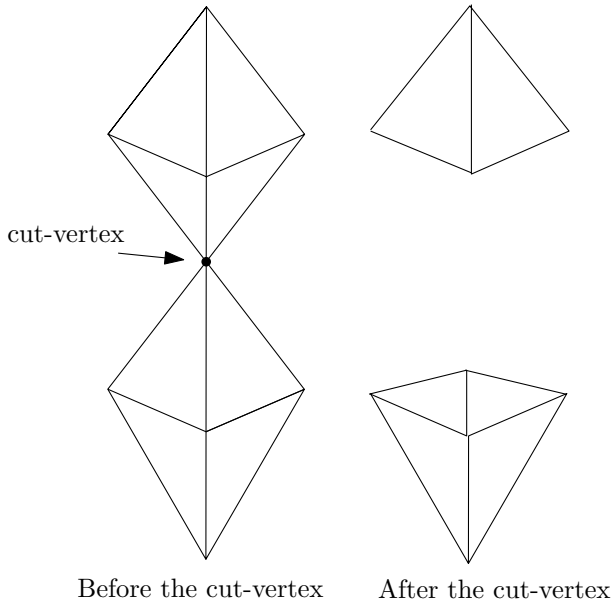


Figure 3: The left column shows the polyhedral object where vertex v is connecting the two-part polyhedron. When cutting at vertex v, the polyhedron separates into two polyhedra.

Definition 1.2.3. *Let p and q be distinct vertices in 3D. The Plücker coordinates [4] of p and q are given as a 6-dimensional vector $(d : m)$ where $d = q - p$ and $m = p \times q$ (i.e. the cross product of p and q).*

The Plücker coordinates are quite useful in computer graphics. The practicability of the Plücker coordinates lies in determining wheter a vertex p is on the left or right side, or on a face f. f must possess only three vertices in order to determine the orientation of vertex p.

Definition 1.2.4. *Let P be a polygon with vertices $(v_0, v_1, ..., v_n)$ in a horizontal plane. A disk cycle is a circular linked list of 3D segments $(e_0, e_1, e_2, ..., e_n)$. Each e_i belongs to a face of the polyhedron and contains the vertex v_i.*

Figure 4 shows an overview of a disk cycle. The edges A, B, C, and D belong to a polyhedron. When the slicer plane reaches at height z_i, a polygon (square) is created with edges e_1, e_2, e_3 and e_4. These edges are created by walking the pointers of the disk (shown in the middle of Figure 4) slicing each edge at height z_i.

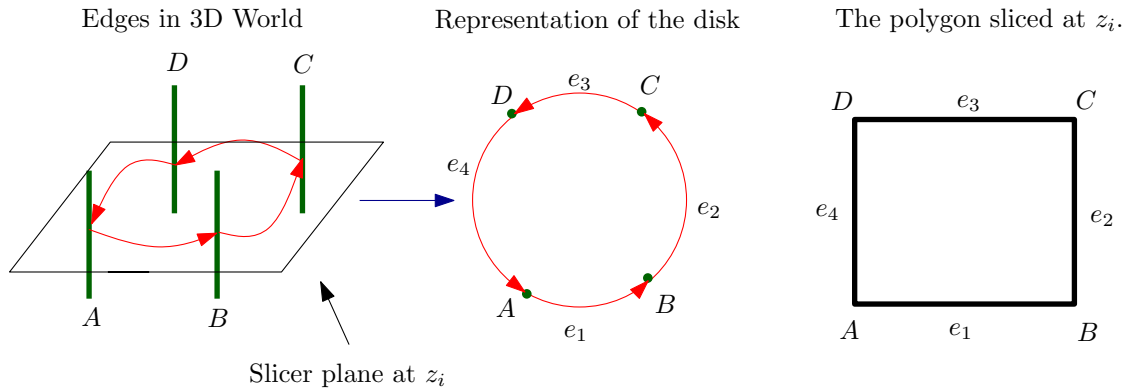


Figure 4: Representation of the disk. The red edges represent the disk which turns counter-clockwise. When sliced at z_i, a square polygon is produced. This is done by computing the intersection of each edge with the plane $z = z_i$. Each edge of the polygon corresponds to a disk edge.

Definition 1.2.5. *Let D be a generic data structure whereby with every update operation, a new version of D is created. A partially persistent data structure has the ability to query any version of D and update the latest version of D.*

According to Driscoll et all. [5], persistence can be applied to any graph-based structure of bounded degree. There are various methods of partial persistence. The method we are using for the polygon nesting tree data structure is the fat-node method. This method is applied to the tree data structures in that every change that occurs in the tree is stored in the nodes. After several changes, the node becomes "fat".

Definition 1.2.6. *A nested polygon tree T is a tree in which each node u of T stores a polygon $polygon(u)$ as well as the boundary of $polygon(u)$. T has the following properties:*

1. *Let u and v be nodes of T. Node u is an ancestor of node v if and only if polygon(u) contains polygon(v).*

2. *The root node r stores a polygon which defines the bounding box of all the polygons on a horizontal plane.*

3. *For each node n_i at depth $d = 2i + 1$ for some $0 \leq i \leq \lceil \frac{height(T)}{2} \rceil - 1$, the orientation of the boundary of the polygon stored in n_i is counterclockwise.*

4. *For each node n_i at depth $d = 2i$ for some $1 \leq i \leq \lfloor \frac{height(T)}{2} \rfloor$, the orientation of the boundary of the polygon stored in n_i is clockwise.*

Item 2 is required; without it, a horizontal layer may have several polygons with no parent. The boundary polygon acts as a parent for any polygon missing a parent at depth 1 of T. Item 3 and 4 are needed for the printing of the layer. The orientation of each polygon determines whether it is a hole or a solid section of the layer. Suppose we want to find a polygon that contains a vertex p. The first test that we run from a node n_i is the boundary box since the comparison operation is constant time. If p is inside the boundary box, a ray shooting test is then performed on the polygon stored in n_i. The structure of T is further discussed in Section 2.3.

1.3 Problem Statement

From Section 1.1, we discussed that a polyhedral object $\mathcal{P}$ is formed by fusing all the layers. A layer Ł is a collection of non-intersecting polygons in a horizontal plane. The polygons are created through the intersection of the horizontal plane and $\mathcal{P}$ at height h. A polygon P is a two dimensional (2D) plane figure that is bounded by a closed polygonal path. This path is a list of edges where edge $e = (v_{i-1}, v_i)$ and $e' = (v_i, v_{i+1})$ are shared by vertex v_i. A polygon P could contain a set of non-intersecting polygons Q_i. In this premise, a polygon P is said to be the parent of polygon Q_i. Polygon Q_i

is a child of P which is nested in P. This parent child polygon relationship may be organized into an acyclic graph (nesting tree) data structure portrayed in Figure 5. A nesting tree is used by the printer to determine which polygon should be printed first.

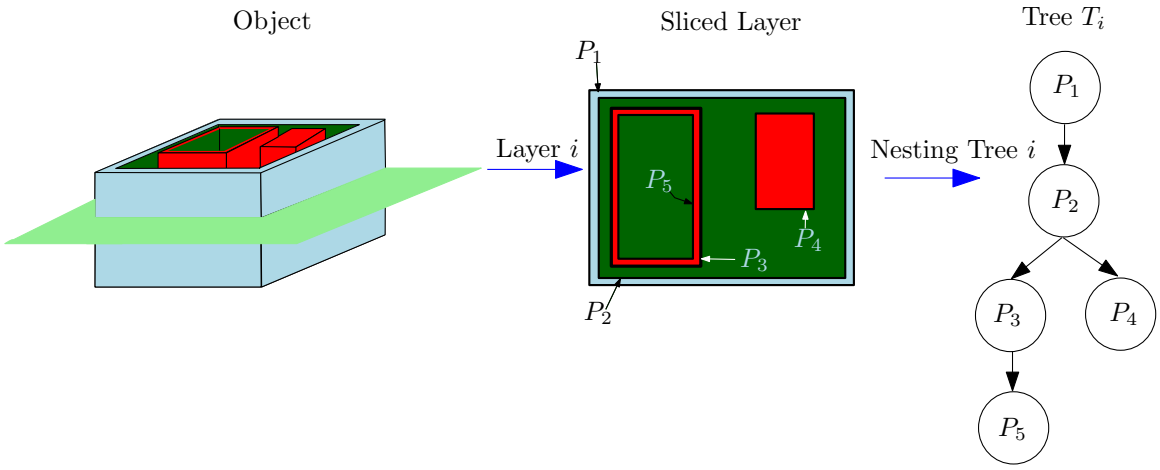


Figure 5: Representation of parent-child polygon relationship in relation to the sliced layer.

As such, we want a data structure $\mathcal{T}$ supporting the following query:

Given a height h of a polyhedral object $\mathcal{P}$, report the nesting tree T representing the layer at height h.

The new data structure is constructed by moving the horizontal plane along from negative to positive z-axis, processing each vertex of the polyhedral object $\mathcal{P}$. The data structure is an implicit representation of the sequence of nesting tree representing all layers. Each node in a nesting tree stores one polygon in the corresponding layer. The nesting tree must have query capabilities as ascertaining whether a polygon P is a leaf in a nesting tree or finding the correct node to add a new disk cycle.

We present a new practical algorithm which constructs the data structure $\mathcal{T}$. $\mathcal{T}$ is used by a layered manufacturing printer to generate a nesting tree at any height h and then print the layer. Using this new data structure, the layers' thickness is

variable, which saves costs and production time. The new data structure would help users to determine what is the best layer thickness for a measured sized object. To calculate the best layer thickness, an algorithm slices an object into specific heights. For each sliced layer, a polygon nesting tree is formed. A user is then able to study the best layer thickness from the set of polygon nesting trees. The thickness of the layer can help to determine what is the best resolution to print the object.

1.4 Background and Related Work

Several works are focused on creating the nesting relationship of polygons for each single layer of a sliced model. Babaj and Dey's polygon nesting algorithm is an example of these works [6]. The nesting relationship of the polygons is derived by decomposing each polygon into a set of polygon subchains, and then using the sweep line technique to detect each polygon subchain. When two new subchains of polygon P are detected from the starting point p, P's parent can be found by searching the closest subchain. Let Q be the polygon of the found subchain. P could be a child Q or the child of Q's parent. A polygon subchain is simply a derivative of convex-chains. These convex-chains can form a convex polygon. A convex-chain is a maximal piece of a polygonal-line where the vertices of a convex-chain forms a part of the convex polygon. A polygonal-line is a sequence of vertices $(v_i, v_{i+1}, ..., v_j)$ going clockwise direction where v_i and v_j are reflex vertices. The subchains' order of vertices either increase or decrease by x coordinate.

Babaj and Dey's algorithm to build the nesting relationship of polygons is simple. The first step is to read all the polygons and break down into subchains. Then, the endpoints of each subchain are added in an array A, and are sorted using the x coordinate as the comparison key. Once the array is sorted, a node is created for each polygon of the sliced layer. A generic binary tree is then initialized to contain the

current intersecting subchains. The key of the generic binary is the starting point of a subchain. Each point p from the array A is then processed. If a point p belongs to two subchains of polygon P, the subchains are removed from the binary tree. The subchains exist in the binary tree since the sweep line passed the beginning point of each subchain. If p is the start of the two subchains, the algorithm will search in the binary tree the closest subchain s that is near p. Let Q be the polygon that contains s. Count how many subchains of Q are above p. If the value is odd, make P a child of Q. Otherwise, make P a child of Q's parent. Finally, if s is not found, the parent of P is null.

The running time of Babaj and Dey's algorithm is $O(n + (m + N)\log(m + N))$ where n is the total number of vertices, m is the number of polygons, and N is the total number of reflex vertices. A similar algorithm developed by Arruda [7] that takes the same approach. Instead of decomposing a polygon into a set of subchains, decompose the polygon into a set of edges. The end point of each edge are marked as begin, mixed, or end. Let $e = (v_{i-1}, v_i)$ and $e' = (v_i, v_{i+1})$. A vertex v_i is a begin vertex if both vertices v_{i-1} and v_{i+1} are to the right of v_i. An end vertex occurs when v_{i-1} and v_{i+1} are to the left of v_i. A mixed vertex is composed of one vertex of the set (v_{i-1}, v_{i+1}) to the left of v_i and another vertex to the right of v_i. Before the sweep line algorithm is executed, a binary tree is initialized to store the current intersecting edges from the sweep line. When the sweep line detects a begin vertex v, it searches in the binary tree for the closest upper and lower edges e and e' near v_i. If both edges belong to the same parent, the polygon of v_i is the child of the polygon containing e. Otherwise the parent of the polygon of v_i is the parent of polygon that contains e'. Of course, if the search did not return the upper and lower edges, there is no parent polygon for the polygon of v_i. The running time of this algorithm is $O(n \log n)$ where n is the total number of vertices of the sliced layer. The running time does not include the total time it takes to construct a polygon nesting tree of an entire polyhedron.

Another algorithm developed by S.H. Choi and K.T. Kwok [8] produces nearly the same results as Babaj and Dey's algorithm. Choi and Kwok's algorithm is divided into two stages: hierarchy sorting and contour sequencing. The hierarchy sorting stage involves finding the nesting of each polygon. This is accomplished by taking each polygon P and using ray-tracing to find the parent of the polygon P. Afterwards, the polygons must be sorted from left to right.

Another method to compute the nesting tree is defined by McMains [9]. In the other works, there is a focus on generating the nesting tree for each layer. From a critical standpoint, it is not precisely specified in the works as to how the layers are sliced. McMains generalizes a slicing algorithm and data structures to efficiently slice an object into layers. For an increasing sequence $(h_1 < h_2 < h_3 < ... < h_n)$, the algorithm reports a nesting tree for each value in the sequence. When the object is sliced, there is a sequence of trees $(T_1, T_2, T_3, ..., T_n)$. If the thickness of the layer is too large or if we would like to have a tree between T_i and T_{i+1}, the algorithm is executed again to generate another sequence of trees. The execution of the algorithm can be time-consuming since the object is composed of several million vertices. A more carefully defined data structure is required to ask for a nesting tree for any given height h.

The overview of McMain's algorithm is a z (slicer) plane is moves from the bottom to the top[1] processing every z-event. A z-event is either a vertex lying on the slicer plane or a z-coordinate which is a multiple of the layer thickness. The slicer plane contains the current 3D segments that are being intersected by it. However, McMain's algorithm is not clear on how the 3D segments normal to the z-axis are handled. These segments are organized into a group of disks. See the problem statement for the definition of a disk cycle.

A z-event could be a begin vertex, a mixed vertex, or an end vertex. Let $e =$

[1]the direction is going up the z-axis to simulate the printing process

$(v_{i-1}, v_i), e' = (v_i, v_{i+1})$ be 3D segments and let ρ be a horizontal plane. Vertex v_i is a begin vertex if v_{i-1} and v_{i+1} are above ρ. v_i is said to be an end vertex if v_{i-1} and v_{i+1} are below ρ. If v_{i-1} is below and v_{i+1} is above ρ (or vice versa), v_i is a mixed vertex. For begin, mixed and end vertex z-events, the disks are either created, modified or destroyed, respectively. When the z-sweep plane reaches a fixed layer thickness, a new layer is produced through the generation of the set of polygons. Each polygon is created by finding the intersection of z-plane and the disk. The orientation of each polygon (clockwiseor counterclockwise) is also determined during the creation of the polygon creation. The hierarchy (nesting) of the polygon is computed to produce a proper output: a 3D-layered manufacturing object. A part of the paper dealt with issues such as non-manifold vertices and edges, floating-point round errors and data input cleanup such as gaps.

Most geometric algorithms that are implemented using a strong type language face issues on float-point values. Kettner et al. [10] shows the classical examples of issues faced when implementing geometric algorithms. The first issue elaborated in the paper concerns the orientation [11] (turn test) of three vertices p, q, r. We say that (p, q, r) makes a left turn if the angle between (p, q) and (q, r) is less than π and is oriented counterclockwise. If the orientation is clockwise, we say that it makes a right turn. If the vertices lie on a straight line, we say that the orientation is collinear. The orientation can be found by placing the three vertices in a matrix and finding the determinant det[2] of the matrix.

Suppose that p, q, and r lie on a line. If we place p, q, and r in a matrix M, we expect the determinant of M to be zero. There are specific coordinate values where the determinant of M is non-zero. This translates into a false result; the output of a geometric algorithm has a high chance to be incorrect.

[2]If det is a positive value, we say that it makes a left turn. If det is a negative value, it's a right turn. Otherwise it is collinear if $det = 0$.

One of the experiments performed by Kettner involed three specific vertices $p = (0.5, 0.5)$, $q = (12, 12)$ and $r = (24, 24)$. Using the following developed formula:

$$float\_orient(p.x + x(y + 1), p.y + y(u + 1), q, r)$$

where u is an increment between adjacent floating point values and x and y are the coordinates of a single pixel shown in Figure 6. In short, the first vertex is moved very slightly. Based on $0 \leq x, y \leq 255$, the resulting value is assigned to three colours: red for the negative value, blue for the positive value and yellow for the collinear value. After processing all the x and y values, the results are transferred to a scatter plot, as shown in figure 6. Each pixel in the scatter plot corresponds to the x and y values. The solid black line runs through p_2 and p_3. Intriguingly, on the scatter plot is the red and blue half-space contain several yellow pixels. There is supposed to be solid yellow band pixels on both sides of the line. Furthermore, there are red pixels in the blue half-space and blue pixels in the red half-space. From this scatter plot we can clearly observe the floating point errors that occur in the process of computing the orientation of three points.

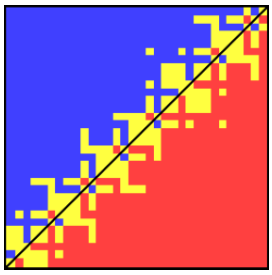


Figure 6: The orientation experiment carried out by Kettner [10].

To avoid these floating-point error issues, LEDA [12] developed a C++ class known as Rational. The Rational class keeps the value in rational form p/q where p and q are of type LEDA-integer. The integer class is implemented by a vector of long integers.

For example, when adding two rational numbers p/q and r/s, LEDA stores the results as the two integers $(ps + qr)/qs$. This process helps to avoid floating-point issues. After an operation, the LEDA rational value may not be normalized (e.g. 2/4). In this case, a function *normalize* is used to reduce the rational value. This step helps to reduce the amount of used computer memory. Invoking this function is expensive because the GCD algorithm is executed. A disadvantage of using the Rational class is the amount of memory used. To load a large dataset file that contains over 100 000 vertices, the computer memory may increase to 1 GB. Nevertheless, the issues examined in the Kettner paper do not appear when using the LEDA framework.

1.5 Summary of Results and Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 A detailed explanation of the constructive polygon nesting trees data structure using the fat node persistence technique. The first section of the chapter covers the preliminaries, assumptions made. Additionally, a detailed description is provided for the execution of a constructive algorithm and the operations of the polygon nesting trees data structure. The key to querying a nesting tree for any height h is to use persistence to implicitly store the history of the algorithm.

The polygon nesting trees data structure is constructed by a horizontal plane is swepping upwards on a polyhedral object $\mathcal{P}$, processing every vertex. For every new z value of $\mathcal{P}$, the data structure creates a nesting tree. Based on a vertex p of $\mathcal{P}$ being processed, the data structure needs to decide on how to find the proper node in the nesting tree and then make the necessary modifications. These modifications include adding or removing disks from the nesting tree $\mathcal{T}$, taking a disk and splitting it into two disks in $\mathcal{T}$, or taking two disks and merging them into one disk in $\mathcal{T}$. For example, let us assume that vertex v is a begin vertex. The algorithm needs to find

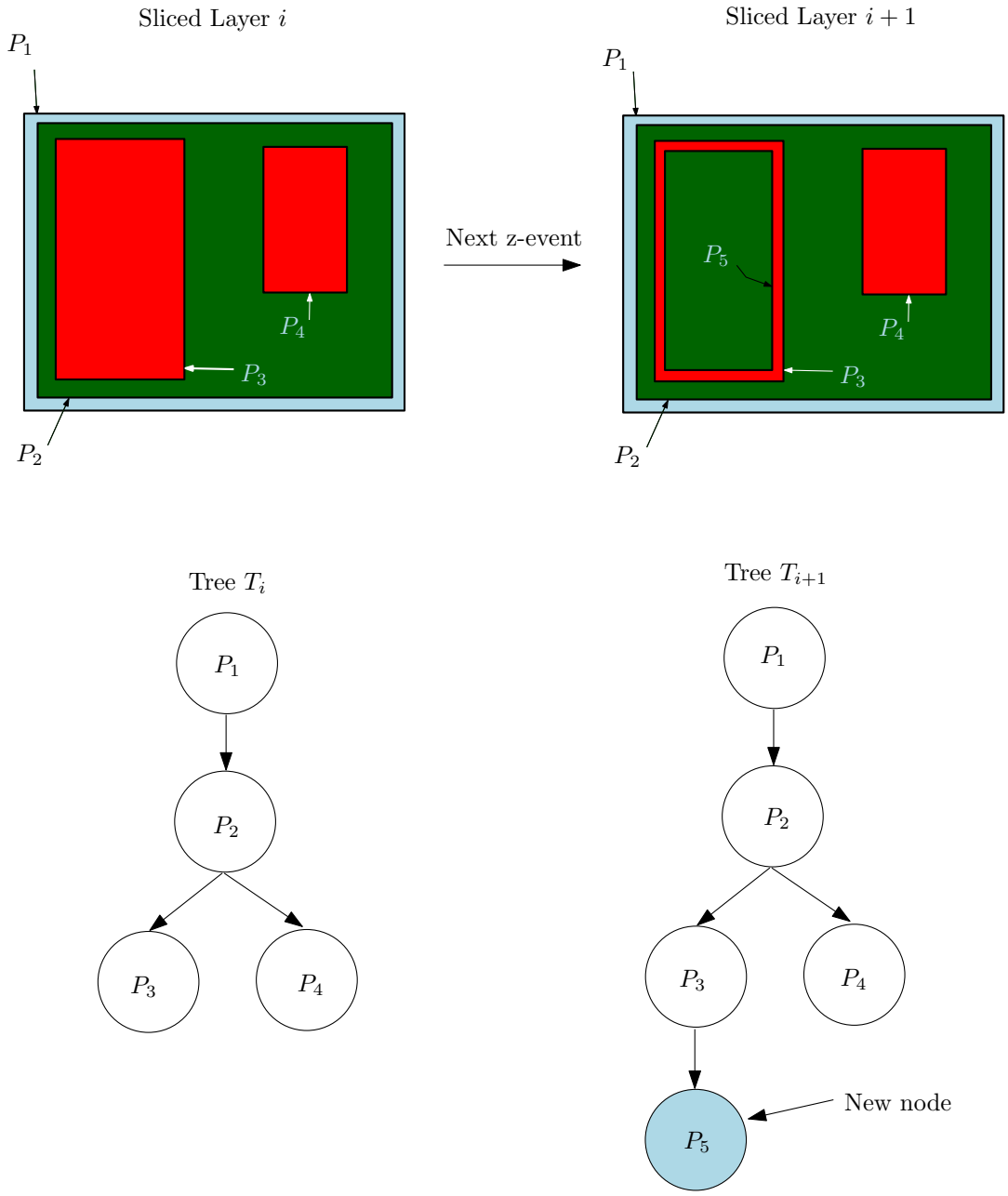


Figure 7: Representation of parent-child polygon relationship in relation to the sliced layer. Tree T_{i+1} is simply a copy of tree T_i with a new node. The new node appeared because there is a hole in polygon P_3.

the leaf of $\mathcal{T}$ by performing several geometric computations of v related to the node. It then creates a new node and attaches it to the leaf, as shown in Figure 7. Each node in $\mathcal{T}$ stores a disk cycle.

Chapter 3 This chapter explains thoroughly the implementation of the developed algorithm and data structure which uses the LEDA library. The implementation was completed using the C++ language with Microsoft Foundation Classes (MFC) framework and OpenGL for graphics manipulations.

Chapter 4 A summary of the results of the previous chapters and a presentation of appealing problems that could potentially be explored in future research.

Chapter 2

The Polygon Nesting Trees Data Structure and the Associated Algorithm

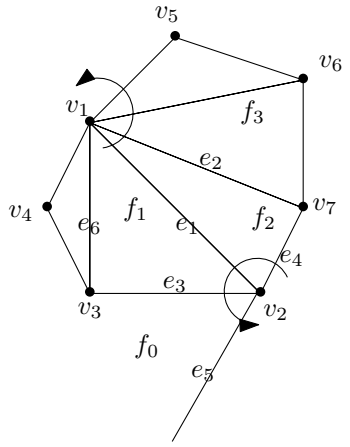
This chapter elaborates on the new polygon nesting trees data structure and the slice algorithm. Section 2.1 discusses in detail the input and output of the slice algorithm and the assumptions made in order to obtain the correct results from the slice algorithm. Section 2.2 explains in depth the nature of a disk and the set of operations offered by a disk. Section 2.3 describes the new polygon nesting data structure. The details of the slice algorithm are given in Section 2.4. Section 2.5 walks through a simple example of how the disk, polygon nesting trees data structure and slicer algorithm interact to each other. Finally, Section 2.6 analyzes the total running time of constructing the polygon nesting trees data structure.

2.1 Input and Output of the Algorithm

This section describes in detail the input and the output of the slicer algorithm. A list of assumptions is created to produce the correct output.

2.1.1 Input

The input of the algorithm must be a polyhedron $\mathcal{P}$ as defined in Definition 1.2.2. Each $\mathcal{P}$ is stored in a doubly connected edge list [13] data structure (DCEL) to perform a sample of the following queries:



	v_o	v_d	f_{left}	f_{right}	CCW_o	CCW_d
e_1	v_1	v_2	f_2	f_1	e_2	e_3
e_2	v_7	v_1	f_2	f_3	e_4	e_5
e_3	v_3	v_2	f_1	f_0	e_5	e_6

Figure 8: An overview of a doubly connected edge list data structure. v_o and v_d are the source and destination vertices, f_{left} and f_{right} are the adjacent faces to the edge. CCW_o (Counter Clockwise) is the previous edge and CCW_d is the subsequent edge.

- Given a vertex v, return a list of faces and edges that contain v.
- Given a face f, return a list of vertices and edges on f.
- Given an edge e, return the left and right face of e.

For a visualization of the DCEL, see Figure 8.

Each face f of a polyhedron must be composed of only three vertices and edges. If f has more than three vertices, it can be triangulated [13]. The triangulation of each face is outside the scope of this thesis, and will therefore not be described any further in this paper. The purpose of this restriction is to perform a vertex inclusion test in a disk cycle. Each face in a disk cycle is translated to Plücker coordinates. A ray is then shot from a vertex p to a vertex q and q lies outside of the disk cycle.

There cannot be any gaps, cracks or T-junctions in $\mathcal{P}$. The face gaps may be repaired by commercial software. $\mathcal{P}$ may have cracks as a result of the inadequate performance of the scanning operation of an object. Barequet and Sharir [14] meticulously makes recommendations on how to close these gaps properly.

Lastly, the orientation of $\mathcal{P}$ is fixed. That is, $\mathcal{P}$ cannot be translated and/or rotated to avoid the extra preprocessing of $\mathcal{P}$.

2.1.2 Output

The output of the algorithm is a polygon nesting data structure that can be used to generate a polygon nesting tree T for any height h of $\mathcal{P}$. Refer to Definition 1.2.6 for the definition of T. The data structure is further defined in Section 2.3.

2.2 Disk Data Structure

According to Definition 1.2.4, the disk cycle D (a disk, in short) data structure holds a set of edges in a particular order. To represent a set of ordered edges, we need four containers:

1. A Hash Table, D_{HT_E}, containing the set of edges. The key is the actual edge and the value to the key is a boolean value.

2. A Hash Table, D_{HT_V}, containing the set of vertices of all the edges stored in D_{HT_E}. The key is the actual vertex and the value to the key is a boolean value.

3. A single linked list L_P to hold a set of circular doubly linked lists. Each doubly linked list orders a set of edges.

D_{HT_V} and D_{HT_E} are used to query D in constant time if D contains a vertex or an edge. In order to achieve the constant time query, the cuckoo hashing [15] scheme

is used. In theory, the values of both hash tables are not being used. However, for implementation of the disk data structure, the values of both hash tables can be used for the algorithm of the edges ordering as explain below. The single linked list L_P represents a set of partitioned ordered edges. When a disk is being constructed, a group of edges may be independent of another group of edges. After the disk's construction, there is only one group (one doubly linked list D_l) of ordered edges. The first and last edge in D_l point to each other. The ordering of the edges is completed by running the algorithm $\mathcal{A}_o$ as demonstrated below:

Input

A set of edges E that define a disk, but have not been yet sorted.

Output

A single linked list to hold a set of circular doubly linked lists l.

Algorithm

1. Initialize the single linked list L_P.
2. Initialize the doubly linked list $l = \emptyset$.
3. Remove edge e from E, and set $e_{initial} = e$ and $reverse = false$.
4. Add e to the end of l if $reverse = false$. Otherwise, add e to the beginning of l.
5. Select an unprocessed face f that shares e. Mark f as processed.
6. Find an edge e_{next} that is shared by f and is present in E. Set $e = e_{next}$ Go to step 4. If no such edge is found and $e_{initial} \neq \emptyset$, set $e = e_{initial}$, $e_{initial} = \emptyset$, $reverse = true$ and go to step 5. Otherwise, add l to L_P.
7. Go to step 2 if E is not empty. Otherwise, return L_p.

$\mathcal{A}_o$ covers three scenarios. In the first scenario, there is a path that starts and ends with edge e. See Figure 9 for a visualization of the first scenario. In the second scenario, a disk is created from a triangulated polygon lying on the slicer plane. The reverse bit was placed since we may reach the end of the path with a partially built l. We need to go back to the first edge we selected, and start adding each processed edge to the head of l. We could find the edge that starts the path, but this would take the same amount of running time as building l. The last scenario covers the grouping of a set of edges. For example, when a disk is being split into two disks from a vertex v, it is difficult to partition the edges into two set of edges. We need to wait until the horizontal sweep line has swept the z-plane to partition the set of edges.

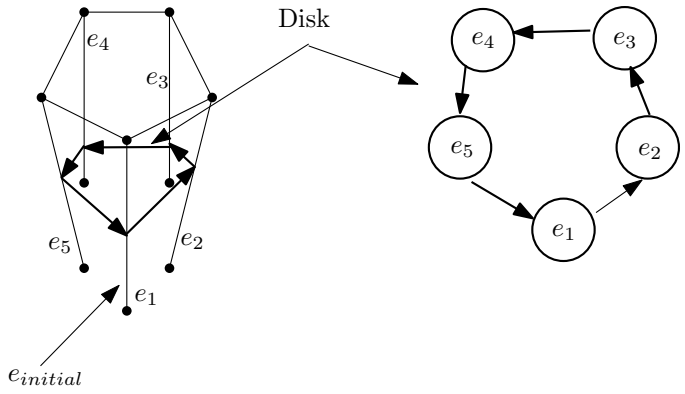


Figure 9: A set of ordered edges representing the disk.

The disk data structure provides a list of helper operations which are heavily utilized by the slicer plane algorithm and $\mathcal{T}$. The slicer plane algorithm (explained in Section 2.4.2) invokes a disk operation to create of a disk; append or replace edges to a disk; split a disk into two disks; merge a set of disks into one disk; or delete a disk. The details of each operation are provided below.

Let's define the following mathematical symbols used by the disk operations.

- ρ is the slicer plane at height h.

- D and D_i are disks, where D_i is inside D.
- E, E_r, E_e, E_s are sets of edges. E_r is a set of edges being removed from a disk, E_e is a set of edges ending at some vertex and E_s is a set of edges starting at some vertex.
- e is an edge
- v is a vertex

• ***Create***

Parameters: E

Returns: a new disk

Initialize $HT_e = \emptyset$ and $HT_v = \emptyset$. Add the set of edges E to HT_e. For each edge e in HT_e, extract the end points e and add them to HT_v. Run $l = \mathcal{A}_o$ to order the edges E_v. Create the disk D_n containing HT_e, HT_v and l, and return D_n.

• ***AddEdges***

Parameters: D, E

Returns: a new disk

AddEdges is an augment of the Create operation. Extract the set of edges from D by walking the D_l, and then add them to E. Invoke D =Create(E) and return D.

• ***RemoveEdges***

Parameters: D, E_r

Returns: a new disk

For each edge e in E_r, remove e from D_{HT_e}, D_{HT_v} and D_l. Return D if the number of elements in D_{HT_e} is greater than zero. Otherwise, return $\emptyset$.

• ***ReplaceEdges***

Parameters: D, E_o, E_n

Returns: a new disk

Define a new set of edges E. For each edge e in D_{HT_e}, if e is not in E_o, add it to E. Add all the edges in E_n to E. Invoke D =Create(E) and return D.

• ***Merge***

Parameters: v, L

Returns: a new disk

We query $\mathcal{D}$ for the set of edges E containing v as the end vertex. Let $E_s \subset E$ be the set of edges starting from v, and let $E_e \subset E$ be the set of edges ending at v. Extract the edges of each disk D in L, and add them in a new set E_n. Afterwards, exclude the set of edges in E_e from E_n, and add the edges in E_s to E_n. Invoke D = Create(E_n), and return D.

• ***PartialSplit***

Parameters: D, E_o, E_n

Returns: a new disk

This function is invoked when a vertex v is detected as a split vertex. Define a new set of edges E. For each edge e in D_{HT_e}, if e is not in E_o, add it to E. Invoke D = Create(E), mark D as a split disk and return D.

• ***Split***

Parameters: D, v, ρ

Returns: a single linked list L containing two new disks

This function is invoked when D is marked as a split disk and the horizontal sweep line is completed at the current ρ slicer plane. Initialize a single linked list L to hold a set of disks. Convert each doubly linked list l_d in L_P of D to a set of edges E, and invoke D_n = Create(E). Add D_n to L. Once all the doubly linked lists have been processed, return L.

• ***Has***

Parameters: D, (v or e)

Returns: a boolean value

Returns true if D_{HT_v} contains the vertex v or D_{HT_e} contains the edge e. This is to avoid executing the Inside operation.

• ***Inside***

Parameters: D, (v or e or (D_i, ρ))

Returns: a boolean value

This operation can accept three different types of parameters with D. The first or second parameter accept a vertex or an edge and Inside operation return true if it is inside disk D. Finally, the third parameter is a disk D_i. We test for the containment of disk D_i in disk D. We are certain that two disks are not overlapping because we merge the disks. We take one edge from D_i and slice it at ρ to obtain a vertex p from D_i at height ρ. p is either inside D or outside of it. This can be determined by using a ray-tracing algorithm, which is described below:

1. Build an array of faces A_f from D_l.

   Let n be the number of edges stored in D_l and $i = 0$. Create a face from the pair of edges e_i, $e_{(i+1) \bmod n}$ of D_l where e_i and $e_{(i+1) \bmod n}$ share a common vertex v_c. The face contains v_c and the vertex of each edge e_i and $e_{(i+1) \bmod n}$ that are not

equal to v_c. Store the face A_f, and increment i by one. Stop until $i = n$.

2. Select a random point q that lies on ρ and is outside of $\mathcal{P}$. Create a ray r that starts from p and ends at q.

3. Initialize the intersection counter $int\_cnt = 0$.

4. For each face f in A_f, increment int_cnt by one if r intersects f.

5. If int_cnt is odd, return true. Otherwise, return false.

The first step is to build the array of faces in order to perform the ray-tracing operation. To verify the intersection of r with a face, the edges of each face and r are converted from Cartesian to Plücker's coordinates [4]. Using Plücker's coordinates, we can easily determine if r intersects a face f. The intersection of r and f is calculated by determining the orientation of each edge of f with r. If r intersects an edge, the orientation value is zero. Otherwise the orientation is clockwise if r is above the edge, or counter-clockwise if r is below the edge. If the orientation of r to each of the edges of the face is counter-clockwise, then we can be certain that r is going through the face. See Figure 10 for more details.

Figure 10 demonstrates the case of the line intersecting a face. However, we are actually creating a three-dimensional segment r. The segment may not intersect the face. To cover this scenario, we need to perform one more calculation to determine if r truly intersects the face. The first step is to create three additional lines where L_1 contains the starting vertex of r and a vertex v of the face; L_2 contains the ending vertices r and v and L_3 contains the vertices of the face that are not equal to v. We convert all three lines to Plücker's coordinates. We then calculate the orientation of L_1 and L_2 to L_3. If both results have the same orientation, then r truly intersect the face. Otherwise, it is not intersecting the face and r intersects at one endpoint of the line segment.

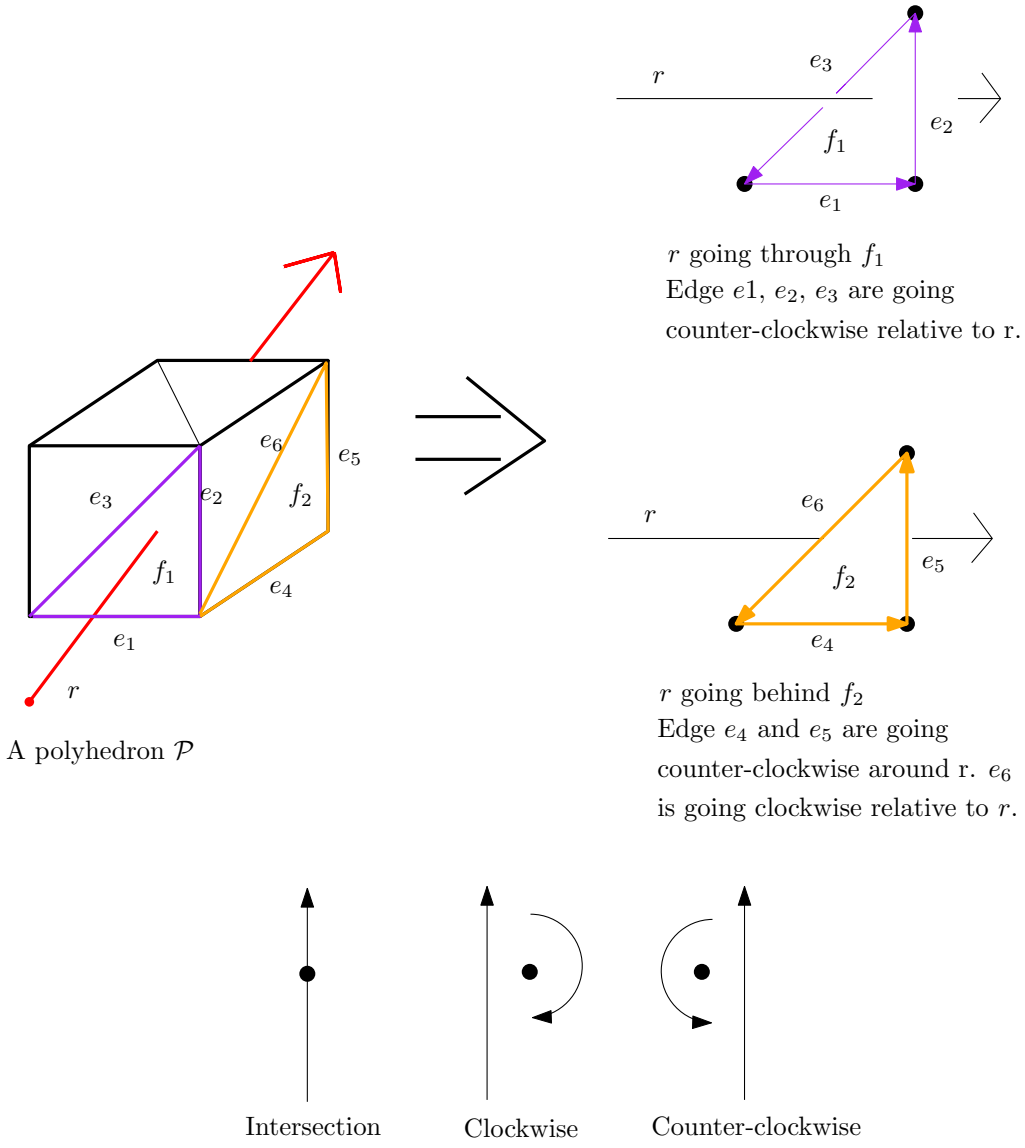


Figure 10: An overview of how r is intersecting f_1 by calculating the orientation of each edge. All the edges in f_1 run counter-clockwise to r where e_6 of f_2 runs clockwise to r. It shows r is going behind f_2.

Lastly, when r goes through an edge of f and does not lie on the slicer plane, we generate another point q and start at step two of the algorithm. With this procedure, we prevent additional processing and determine if r truly intersecting f. In some circumstances, a face can start from the slicer plane with one of its edges lying on it.

In this case, r intersects the edge lying on the plane, and we increment the *int_cnt* by one.

• ***Print***$(D,\ \rho)$

Produce a polygon by slicing D at height h. h is retrieved by asking the current z value of ρ. First, create an array A_p that will hold the list of points of the generated polygon. The size of A_p is the amount of stored edges in D_l. Each point $p = (x, y)$ in A_p is a sliced edge at ρ. Let $i = 0$ be the current index of A_p. For each edge e in D_l, slice $e = (v_s, v_e)$ at ρ and store the resulting point in $A_p[i]$. Increment i for each processed edge. After each edge of D_l has been processed, return A_p.

2.2.1 Running Time Analysis

Most of the disk operations run in $O(|E_d|)$ time where E_d is the number of edges being held by the disk. The has/inside function has a running time of $O(|E_d|)$. The has/inside function may restart a constant number of times to generate a correct random point outside of the disk.

2.3 Polygon Nesting Tree Data Structure

The polygon nesting tree data structure $\mathcal{T}$ is a data structure that produces a polygon nesting tree at a requested height h. $\mathcal{T}$ attains its full construction when the slicer plane moves from the lowest to the highest z value of $\mathcal{P}$. When a slicer plane detects a new z coordinate, a disk can be created, updated or deleted. That is, a node is being created, updated or removed from $\mathcal{T}$. When z is detected as a merging of disks, all the disks are merged to one disk. In $\mathcal{T}$, the nodes are merged into one node. The

reverse applies when a disk is split into two disks.

We need to store the changes done in $\mathcal{T}$ for every processed z value in order to produce a polygon nesting tree at the requested height. The polygon nesting trees differ minimally between height z_i and z_{i+1} of $\mathcal{P}$. We can store the changes of the polygon nesting tree at z_i in $\mathcal{T}$ to produce one at height of z_{i+1}. To achieve this such storage, a persistence data structure is used.

$\mathcal{T}$ closely follows the partial persistence data structure as described by Tarjan [5]. We use the fat node technique because a node may contain more than two children nodes. For our use, each node in $\mathcal{T}$ contains two partitions of children nodes: active and inactive. An active child node is a child node that is currently being used at the latest version i of $\mathcal{T}$. An inactive child node is a child node that was created at version a and deleted at version b, where $a \leq b \leq i$. At version i of $\mathcal{T}$, several children nodes exist and a set of children were deleted before version i. To hold a set of active and inactive children, hash tables are used with the cuckoo hashing technique [15]. Each entry in a hash table contains the disk ID as the key. Furthermore, the data value is the pointer to a child node. Hash tables help to reduce the access time to a constant value. See Figure 11 for the structural overview of a node.

When a disk operation (listed in Section 2.2) is executed, a new disk is always produced. A unique identifier ID is assigned to the new disk for reference purposes. The generation of the ID is a variable incremented by one for every created disk. The new disk is then added to the hash table HT_D. The key of the new disk is the unique identifier. Again, to achieve constant time access in searching for a disk, the cuckoo hashing [15] technique is employed. The unique identifier of the disk is then stored in version i of a node in $\mathcal{T}$. For the active and inactive children of a node, the key of each child node contains the last modified disk ID in the fields array.

Each node in $\mathcal{T}$ contains a fields array. Each entry in the fields array of a node contains a pair value of the version and the actual data. During the construction of

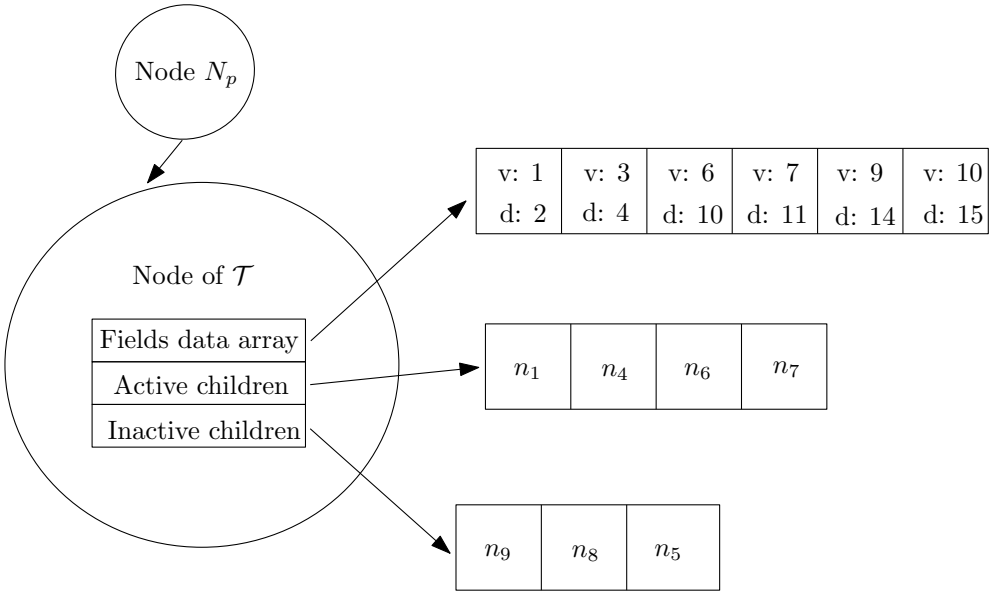


Figure 11: An overview of a node in the Polygon Nesting Tree Data Structure.

$\mathcal{T}$, we always access the latest entry of the fields array. As shown in Figure 11, the first version contains the unique identifier "2" of the disk. In version 2 of $\mathcal{T}$, there was no modification to the disk with ID "2". Hence, there was no entry in the node shown in Figure 11. In version three, the disk was modified to produce a new one that was created with ID 4.

As previously mentioned, each version of $\mathcal{T}$ is identified as the number of z values existing in $\mathcal{P}$. This is translated to an array of sorted vertices ϖ, explained in Sub Section 2.4.1. Each entry in the primary array of ϖ represents a version of $\mathcal{T}$. Each version in $\mathcal{T}$ may contain several modifications based on the amount of stored vertices in the secondary array. As such, $\mathcal{T}$ is always modified at the latest active version. The active version is incremented when the primary walker is moved to the next entry of the primary array in ϖ.

$\mathcal{T}$ needs the two additional hash tables HT_{new}, HT_{split} to store the newly created nodes and the nodes marked as split. The key of each hash table is the node, and the value is the parent of the node. As the horizontal sweep line travels from the left

to the right on the z slicer plane, a disk D is being constructed. This disk is known as an open disk. When another disk D_c is being constructed, there is a possibility of node N_{D_c} being a child of node N_D. N_{D_c} should be stored in another node. After the horizontal sweep line is completed, D and D_c have become closed disks. We can verify that each child of N_d is pointing to the correct parent. If not, the child node is then relocated to the correct node. As for HT_{split}, we let the horizontal sweep line swept along on the z slicer plane. Once the sweep is completed, for each marked node in HT_{split}, we split on the disk. This operation execution groups the edges into two partitions.

The following sub sections list the operations offered by $\mathcal{T}$. Each operation is closely tied to the disk operations. That is, when the split of a disk occurs at some vertex, $\mathcal{T}$ will simply execute the split operation of the disk.

2.3.1 Initialization

Before initializing $\mathcal{T}$, we need to calculate the bounding box of $\mathcal{P}$. First we extract the minimum and maximum x, y, and z value of $\mathcal{P}$. We then increment each maximum value by one and decrement each minimum value by one. Additionally, we create eight vertices. Each vertex is created by choosing the minimum and maximum of each coordinate and then adding them to the set. A set of four edges is subsequently created from the set of vertices. Each vertex represents one corner of the bounding box. Each edge represents the vertical edge running from one corner to another. Finally, a disk D_r is created from the set of edges.

D_r is added to HT_D with an ID of zero. The root node N_r of $\mathcal{T}$ is created. The first field contains a version number of zero and the disk D_r. We initialize an empty linked list in the active children field and an empty linked list in the inactive children field of N_r. We create the variable v_i and set it to one. The v_i variable is used to stamp the field of a node.

2.3.2 Search Operation

The search operation is imperative for the construction of $\mathcal{T}$. The update operations of $\mathcal{T}$ (defined in Section 2.3.3) searches for either a node containing a disk D or a node that potentially containing a child node holding D. The search algorithm described below is executed in the latest version v_i of $\mathcal{T}$.

Let N_{ret} be the root, $N_p = \emptyset$ be the parent node of N_{ret}, D be the disk to search the $\mathcal{T}$ node and ρ be the slicer plane at height h. Then, the search algorithm is as follows:

1. Query the child node in the active children field of N_{ret} which contains D. This process entails looking up the ID of D in the active children hash table. If a child node N_c is returned from the search, set $N_p = N_{ret}$, $N_{ret} = N_c$, and go to step 3.

2. Invoke the $Inside(D_c, D, \rho)$ disk operation for each active child N_c in N_{ret}. If the operation returned true while processing N_c, set $N_p = N_{ret}$, $N_{ret} = N_c$, and go to step 1.

3. Return N_{ret} and N_p.

Step 1 of the search algorithm verifies whether one of the active child nodes of N_{ret} contains the disk D. If no such child is found in step 1, we verify whether a child node of N_{ret} can hold D like the child node in step 2. If a child node N_c is found, we repeat step 1 with N_c since there could be a child node containing D. Otherwise, we return N_{ret}. We set N_{ret} to the root node of $\mathcal{T}$ since a disk could not be contained within any of the N_r active children. As such, the N_r node is the node that defines the boundary of $\mathcal{P}$. Lastly, the search algorithm always returns the parent of N_{ret}. The parent node may be used for the update operations of $\mathcal{T}$.

The search algorithm described above is used also to search for the disk containing a vertex or an edge. This search is required when the slicer plane is processing each vertex of the polyhedron. For example, when a vertex v is detected by the slicer plane, one of the very first operations performed is a search in $\mathcal{T}$ for the disk containing v. If no such disk exists, the search algorithm returns the disk containing v. This returned disk is helpful when updating a disk. To address the edge case, when searching for the disk containing the edge, the edge is actually sliced at the current z coordinate of the slicer plane.

2.3.3 Update Operations

The update operations represent a modification of $\mathcal{T}$. The update operations are used when a new disk is created or removed, or a value in a disk is updated. It may occur also when two disks have been merged to one disk or a disk has been split. The end result is a modified $\mathcal{T}$. Below is a list of six possible updates to $\mathcal{T}$.

- ***Add($\mathcal{T}$, D)***

This function simply adds a disk D to $\mathcal{T}$. First, locate which node in $\mathcal{T}$ will contain D. To find the correct node, we execute the search operation function (defined in Section 2.3.2) to retrieve the node N_p that contains D. Third, retrieve an inactive child node N_D in N_p. If no such node exists, create a new node N_D. Finally, create a field with version v_i. The data is assigned to the ID of D. Add D to HT_D, and add N_D to the active children field of N_p. Add N_D to the HT_{new}.

- ***Remove($\mathcal{T}$, D)***

This function removes the disk D from $\mathcal{T}$. We execute the search operation to find the node N_D that contains D. Retrieve the parent node N_p of N_D, which is stored

in the N_D parent field. Create a field with version v_i and null data. The null data represent the node's deletion being deleted at a specific version. Move N_D from the active children to the inactive children in N_p.

- ***Update($\mathcal{T}$, D_o, D_n)***

Simply replace the current disk D_o by D_n at version v_i. Search in $\mathcal{T}$ for the node N containing D_o. If the fields data array contains an entry of version v_i, replace the data value with the ID of D_n. Otherwise, create a new field entry with the version v_i and data value of D_n's ID. Add D_n to HT_D.

- ***Merge($\mathcal{T}$, v, L_D)***

Merge a list of disks L_D into one disk D_m from the vertex v. Execute the $Merge(v, L_D)$ disk operation to obtain the disk D_m. For each disk D_q in L_D, query $\mathcal{T}$ for the node containing D_q. Add the resulting queried node to the list L_n. Each node in L_n must point to the same parent node N_p because all disks in L_D are merging towards the same vertex v. N_p is obtained from querying the tree node containing the first disk of L_D.

We need to take an inactive child node N_{D_M} from N_p in order to hold the newly created disk D_m. If one does not exist, simply create a new node N_{D_M}. In either case, add N_{D_M} to the active children field of N_p. Create a new entry in the fields array of N_{D_M} with the version value of v_i, and assign D_m's ID to the data value. For each node in L_N, add the pointer of each active child node to the active children field of N_{D_M}. Remove each node N_d in L_n by invoking the $Remove(\mathcal{T}, N_d)$ operation.

- ***PartialSplit($\mathcal{T}$, D, v)***

The vertex v is marked as a split vertex. We first ask the DCEL data structure to

return the set of edges E_o that ends with vertex v and the set of edges E_n that starts with v. We execute $D_{new} = PartialSplit(D, E_o, E_n)$ and then search in $\mathcal{T}$ for the node N_D (and the parent node N_p) that holds D. We add D_{new} to the active children field of N_D. Finally, N_D is added to HT_{split} where the key is N_D and the value is N_p.

• ***Split($\mathcal{T}$, N_D, N_p)***

We split the disk stored in N_D into two disks. The splitting of the disk is accomplished by executing the $Split(D, v, \rho)$ disk operation which returns disks D_1 and D_2. These disks are added to HT_D. The parent node N_p of N_D will be the parent for nodes N_{D_1} and N_{D_2}. N_{D_1} holds D_1 and N_{D_2} holds D_2. N_{D_1} and N_{D_2} are either created or borrowed from the inactive children fields of N_p. They are subsequently added to the active children. For each active child N_c of N_D, we conduct a test to determine if the disk in N_c belongs to either D_1 or D_2. This test is performed by executing the $Inside$ disk operation. Depending on the test result, a pointer copy of N_c is placed in the active children field of either N_{D_1} or N_{D_2}. Finally, we remove N_D by invoking the remove function of $\mathcal{T}$.

• ***IncrementVersion($\mathcal{T}$)***

The $version$ variable is incremented by one. The $version$ variable represents the latest version at $\mathcal{T}$, and is used to create a field entry to every created, modified or deleted node in $\mathcal{T}$. This function will also verify that every new node N in HT_{new} is pointing to the correct parent node. First, we search the latest stored disk D in N for the node N_p containing D. N_p is considered as the correct parent node N. If the value of $HT_{new}[N] \neq N_p$, then we reallocate N to be a child of N_P. Also, for every node N in HT_{split}, we perform $Split(\mathcal{T}, N, HT_{split}[N])$. Finally, we clear all the entries in both HT_{new} and HT_{split}.

2.3.4 Querying a Polygon Nesting Tree at Height h

This section elaborates on the algorithm that queries a polygon nesting tree at height h of $\mathcal{P}$. As explained in Section 1.3, the algorithm described below provides a possible solution to the problem statement. Querying a Polygon Nesting Tree at height h helps to determine the best thickness of a sliced layer of $\mathcal{P}$.

Before explaining the internal working of the algorithm, let us briefly discuss a data structure known as a sorted array of vertices. In a sorted array of vertices, each entry in the array contains a pair value of z coordinate and an array of all the vertices containing the z coordinate. The array holding the pair value is referred as the primary array, and the array in the pair value is known as the secondary array. The primary array is sorted from the lowest to the highest z coordinate of $\mathcal{P}$. Contrastingly, each secondary array is sorted by the x coordinate and then the y coordinate of each vertex. ϖ is discussed in depth in Section 2.4.1.

The key to generating a polygon nesting tree T is to use the pre-order tree traversal algorithm. The traversal algorithm creates T while walking on $\mathcal{T}$ at a specific version. The specific version is determined by locating the entry index in the primary array of ϖ, containing the largest z value that is less than or equal to h. The index value is equal to the version.

Each node in T contains the following information:

- A fixed array of 2D vertices representing a polygon.
- A fixed array of children nodes.
- A pointer to the parent node.

The fixed array of vertices represents the boundary of a polygon, and the vertices v_i

and v_{i+1} represent an edge of the boundary. The orientation of a polygon's boundary can be either go clockwise or counter-clockwise. The second fixed array contains pointers to the children nodes. According to Definition 1.2.6, the polygon of a node contains the polygon of each child's node. The last field is to points to the parent node for easy tree traversal.

We now describe the algorithm that builds T at the requested height h of $\mathcal{P}$.

1. Search in the primary array of ϖ for the version v_i that is closest to h.

   At v_i, the z value of the primary array is less than or equal to h, and h is less than the z value stored at v_{i+1}.

2. Initialize T.

3. Run the pre-order tree algorithm to construct T. Let $N_{\mathcal{T}}$ be the root of $\mathcal{T}$. Then,

   (a) Create node N for T. N is the root node of T if T is empty.

   (b) Perform a binary search in the fields data array of $N_{\mathcal{T}}$ to locate the disk D that has a stamp version of v_i or less.

   (c) Slice D at height h to produce a polygon. The slicing of D is performed by executing the $Print(D, h)$ disk operation. Store the polygon in N.

   (d) Initialize the linked list L.

   (e) Add every active and inactive child of $N_{\mathcal{T}}$ to L containing a version v_i or less.

   (f) Initialize an array in N to hold the set of children nodes.

   (g) For each child N_c in L:

      i. Set $N_{\mathcal{T}} = N_c$, and go to step 3a. Step 3a returns a new node N_n.

      ii. Add N_n to N's fixed array.

iii. Set the parent node of N_n to N.

4. Correct T to satisfy Definition 1.2.6.

   (a) Sort the children nodes of each node in T. The comparison key is the left-most point of a polygon.

   (b) Correct the orientation of each polygon in T. For each odd level of T, the orientation of the boundary of each polygon is counter-clockwise. For each even levels of T, the boundary of each polygon is clockwise.

5. Return T.

Step 1 to 3 are relatively straightforward. We locate the version that should walk on $\mathcal{T}$, initialize T, and then invoke the pre-order tree traversal algorithm. We commonly initialize $\mathcal{T}$ by setting the root node pointer equal to null. In step 4, we correct T to remain true to Definition 1.2.6 which stipulates that the computation used to maintain T is at once intense and complex since we need to remember which level of T we are in and we need to insert the child node in the correct index of the array. The children nodes of a node must be in a specific order. That is, the first child node represents the left-most polygon, the second child node represents the second left-most polygon and so forth. This ordering of the children nodes is completed by extracting the left-most point of each polygon and then running a generic sorting algorithm such as Quicksort [16]. The second repair is done on the orientation of the boundary of each polygon in T in keeping with Definition 1.2.6.

There are two steps involved in repairing the orientation of the boundary of a polygon. The first step is to determine which orientation the polygon is following. We take three vertices a, b, c and perform a test turn. b is the left-most vertex of the polygon, a is the predecessor of b and c is the successor of b. If the resulting turn is right, the orientation is clockwise. Otherwise the orientation is counter-clockwise.

The second step is to verify the correctness of the orientation. If the orientation is incorrect, we simply reverse the fixed array of vertices of the polygon.

2.3.5 Running Time Analysis

At one specific z value, $\mathcal{T}$ will contain the largest set of nodes. Let us define a cube that contains $|E_c|$ number of edges (the worst case is a large set of vertical edges running from one corner to another at all corners of the cube, and no two edges intersect). There is a possibility that the root contains one cube and that it possesses c number of children. Each child contains a scaled-down version of a cube of $|E_c|$ number of edges. Each child contains additional c number of children with an even smaller scaled-down version of a cube of $|E_c|$ number of edges. Then $\mathcal{T}$ would contain a total of $O(c^d d)$ nodes, where d is the maximum depth of the cube within the cube.

For the search operation, if we want the right-most leaf of the tree, we may need to execute the *has/inside* disk operation c times for each tree level. The above statement translates to:

$$c \cdot Inside(c_1) + c \cdot Inside(c_2) + ... + c \cdot Inside(c_d)$$
$$= c \cdot O(|E_{c_1}|) + c \cdot O(|E_{c_2}|) + ... + c \cdot O(|E_{c_d}|)$$
$$= c(O(|E_{c_1}|) + O(|E_{c_2}|) + ... + O(|E_{c_d}|))$$

$|E|$ is the total number of edges of an imported dataset. Then

$$|E_{c_1}| < |E|,\ |E_{c_2}| < |E|,\ ...,\ |E_{c_d}| < |E|$$

which translates to

$$= c(O(|E|) + O(|E|) + ... + O(|E|))$$
$$= c \cdot d \cdot O(|E|)$$
$$= O(cd|E|)$$

The search operation takes the worst running time of $O(cd|E|)$. All of the update

operations use the search operation plus some constant value depending on the $\mathcal{T}$ operation. Thus, each update $\mathcal{T}$ operation takes $O(cd|E|)$ time.

Finally, to produce a Polygon Nesting Tree at a specific height h, we first do a binary search to locate the $\mathcal{T}$ version. This takes $O(\log |V|)$ time since each vertex could be a version in $\mathcal{T}$. We then need to traverse every node in $\mathcal{T}$ and slice the disk at height h. The worst possible case to possess have a maximum number of nodes is $O(c^d d)$. Considering that we could be slicing all the edges of all polyhedra, the total worst running time is $O(|E|c^d d)$.

2.4 The Slicer Algorithm

The slicer algorithm is the key to building the polygon nesting tree. The algorithm is decomposed into three major stages: pre-processing, executing the slicer plane algorithm (construction of $\mathcal{T}$) and post-processing. All three stages of the algorithm are formulated as a sequence of steps:

1. Import $\mathcal{P}$ from a storage device.
2. Construct the $\mathcal{D}$ data structure to store the imported $\mathcal{P}$.
3. Create a sorted array ϖ of vertices of $\mathcal{P}$.
4. Initialize $\mathcal{T}$ and the slicer plane ρ.
5. Execute the slicer plane algorithm to construct $\mathcal{T}$.
6. Output the results to a storage device or to the user.

Steps 1 to 4 comprise the pre-processing stage. Step 5 is known as the execution stage, and step 6 is the post-processing stage. Each stage is explained thoroughly in the following sub-sections.

2.4.1 Pre-Processing

A polyhedron can be stored in multiple file formats, including PLY, STL and AMF [17]. These file formats always contain the basic polyhedron data, which are a list of vertex, edges and faces. The data inside the file can be easily extracted and stored into a query data structure. A prominent data structure used to store a polyhedron is the Doubly Connected Edge List, which is explained in Section 2.1.1. The Doubly Connected Edge List was chosen as the query data structure, for the amount of stored information is much less than that of a winged edge data structure [18]. Recall that according to Definition 1.2.2, we do not authorize the designation of a vertex of $\mathcal{P}$ as a cut vertex. The Definition 1.2.2 simplifies the storage system in a query data structure.

Depending on the file format, a block of vertices, edges and faces are read from the data file and stored in $\mathcal{D}$. Additionally, the normal of each face is calculated for graphical display of the polyhedron.

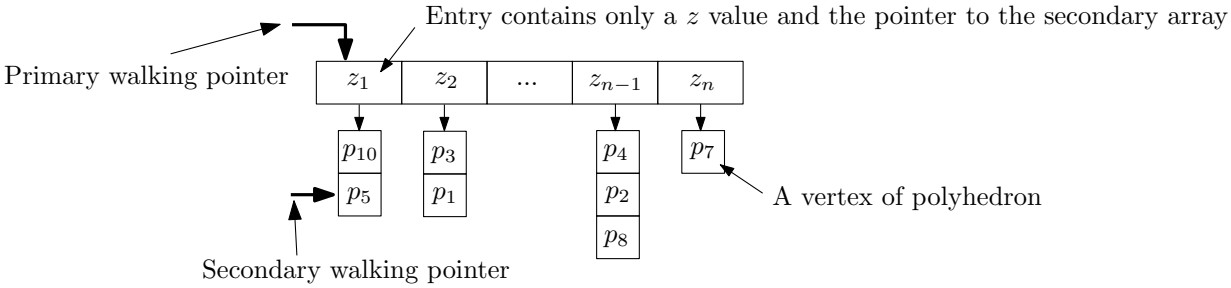


Figure 12: An overview of the sorted array ϖ. Each entry in the primary array contains only the z value from $\mathcal{P}$ and a pointer to a secondary array. A secondary array contains the list of vertices of $\mathcal{P}$.

After the construction of $\mathcal{D}$, the next step is to create an array of sorted vertices ϖ. The construction of ϖ is different from the conventional way. In the conventional manner, all the vertices in a single array are stored. As shown in Figure 12, each entry of the array contains a z-value from $\mathcal{P}$ and a pointer to a secondary array. The

secondary array contains a list of vertices of $\mathcal{P}$, in which z-value of each vertex is equal to the z value stored in the entry of the primary array. As such, all the values in the primary array are sorted by the z value. The entries of each secondary array are sorted by comparing the x-value first and then the y-value of a pair of vertices.

Once ϖ becomes available, the slicer plane ρ is initialized for it needs ϖ to process all the vertices. Finally, the $\mathcal{T}$ data structure is initialized and is ready for the execution of ρ.

2.4.2 Slicer Plane Execution

In order to build $\mathcal{T}$, we need to slice $\mathcal{P}$ by processing every vertex of $\mathcal{P}$. The main technique used by the slicing algorithm is that of a plane sweeping from the lowest to the highest value z of a vertex. When a z-value vertex has been detected, a sweep line on the xy plane is then executed from the smallest to the largest x-value vertex. If $x_i = x_{i+1}$, then the vertex of the $min(y_i, y_{i+1})$ is processed first.

The ϖ data structure is the structure that emulates the sweeping of ρ. As shown in Figure 12, there are two walking pointers. The primary walking pointer points to the entry of the primary array. The secondary walking pointer points to the active secondary array. The secondary array becomes active when the primary walking pointer points to the entry where the secondary array is located. The primary walking pointer is referred to as the sweeping plane ρ and the secondary walking pointer is referred to the sweep line on the xy plane of ρ.

As explained in Section 2.3, $\mathcal{T}$ contains a function known as the *IncrementVersion*. The function is invoked when the secondary walking pointer has reached the end of the active secondary array. *IncrementVersion* was developed to prevent unnecessary creations of the $\mathcal{T}$ versions.

As each vertex v is detected, v is marked as a start, a mixed or an end vertex. The marking of v is accomplished by the following steps:

1. Query the edges from $\mathcal{D}$ containing v as an end-vertex. Let E be the set of edges returned from the query.

2. Initialize two counters: $below = 0$ and $above = 0$

3. For each edge e in E

   (a) Retrieve the other end-vertex v_d.

   (b) Increment the counter *below* if $v_d.z < \rho$ or increment the counter *above* if $v_d.z > \rho$.

4. Mark v as start if $above > 0$ and $below = 0$, mixed if $above > 0$ and $below > 0$ and end if $above = 0$ and $below > 0$.

If the counters are equal to zero, then v is marked as $NOOP$ and v is marked as visited because there are no changes to be made in $\mathcal{T}$. After marking v, the begin-, mixed- or end-vertex case algorithm is executed. Each vertex case is explained in the next sub sections.

Start Vertex Case

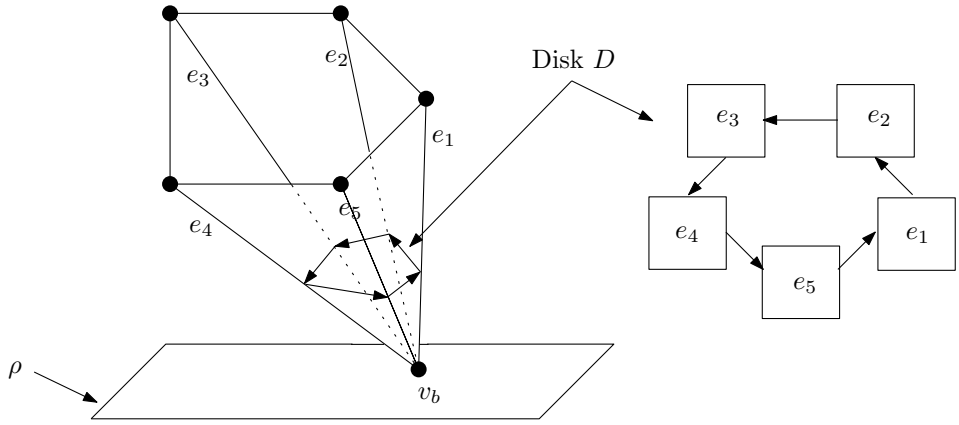
Start vertex v_b is defined as the set of edges $E_{v_b} = \{(v_b, v_e) : (v_b.z = \rho \wedge v_e.z > \rho) \vee v_b \text{ is marked as } NOOP\}$. Essentially, the start vertex case is classified as the creation of a disk which is added to $\mathcal{T}$. When v_b is detected by ρ, we query $\mathcal{D}$ for the set of edges E_{v_b}. There are two scenarios that arise when resolving E_{v_b}. In either scenario, the end result is v_b being identified as visited.

Let $\vec{n}$ be the normal of ρ.

Scenario One: $(e.v_e - e.v_b) \cdot \vec{n} \neq 0 \quad \forall e \in E_{v_b}$

In this scenario, a disk containing all the edges of E_{v_b} is created. For every edge e in E_{v_b}, edge e is shared by two faces f_i and f_j. Two edges of f_i and f_j are in E_{v_b}.

In other words, if we select face f_i and ask for the incident unprocessed face which shares v_b, we will obtain f_j. If we ask for the unprocessed incident face on f_j, we will get f_k. If we keep repeating the query with the face returned from the previous queried incident face (that was not processed), we will eventually get f_i. The creation of the disk is accomplished by ordering the edges in E_{v_b}. Once the disk is created, it is added to $\mathcal{T}$ by executing the *Add* operation. See Figure 13 for an example of how the disk is created when v_b is detected.



v_b is discovered where the edges are above the slicer plane

The new disk D representing the discovered edges from point v_b

Figure 13: Vertex v_b has been discovered by ρ. A new disk D is created by ordering the set of edges E, and D is then added to the $\mathcal{T}$ data structure.

Scenario Two: $\exists e \in E_{v_b}$ such that $(e.v_e - e.v_b) \cdot \vec{n} = 0$.

This scenario is more complicated than the first scenario because a disk may have been created or may result in a merging of several disks into a single disk. An example of this scenario involves a triangulated polygon lying on ρ. Let $\alpha(v_b)$ be the set of horizontal edges. For each edge e in $\alpha(v_b)$, we query the disk D_q in $\mathcal{T}$ that contains the end-vertex $ep = EP(e \in \alpha(v_b), v_b)$. Add each searched D_q (if they exist) from the query of each ep to list L. If ep is marked as $NOOP$, we need to extract the horizontal edges $\alpha(ep)$, and then add them to set $\alpha(v_b)$. Repeat this process until all the edges in $\alpha(v_b)$ have been processed.

Let s be the number of disks stored in L and $\beta = E_{v_b} \backslash \alpha(v_b)$. If $s = 0$, disk D is created with the edges from set β, and D is added to $\mathcal{T}$. If $s = 1$, the edges from β are added to the only disk D stored in L. The node of $\mathcal{T}$ containing D is updated to create a new version of the node. The *update* operation of $\mathcal{T}$ is the one that will update the current disk in $\mathcal{T}$. Otherwise, take all the disks in L and merge them to the new disk D_m that includes the edges from the β. The merging of the disks and the update in $\mathcal{T}$ are completed by the $Merge$ operation of $\mathcal{T}$.

$\mathcal{T}$ needs to be updated correctly when a set of disks in L is merged to the single disk D_m. Each disk in L is associated with a node in $\mathcal{T}$. Let $\delta = \{N_d | d \in L \wedge N \text{ is a node in } \mathcal{T}\}$ and let N_{Dm} be the node containing D_m. All the nodes in δ are pointing to the same parent node N_p since the merge is occurring at v_b. All the nodes' children in δ become the children of N_{Dm}. N_p contains N_{Dm} as a child. An example of resolving this case is shown in Figure 14. D_1 and D_2 are created when vertices v_1 and v_2 are detected by the sweep line. The edges that contain the endpoint v_5, v_6 and v_8 are added in disk D_2. The edges that contain the endpoint v_3, v_4 and v_7 are added in disk D_1. When the sweep line reaches vertex v_{10}, D_1 and D_2 are merged to a new disk, named D_3.

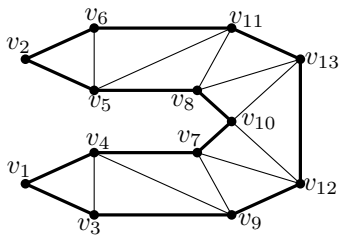


Figure 14: A triangulated polygon lying on ρ. Each vertex has the edges going out of the page.

There is a possibility that a polygon lying on ρ contains another polygon (a hole) as shown in Figure 15. If we follow the procedure previously described, the non-horizontal edges containing the end vertex of v_5, v_6, v_7 and v_8 are added to the disk

D_1. To handle this complication, we need to determine if the non-horizontal edges of v_6 should be included in D_1. We verify the existence of a face with vertices v_1 and v_6. If no such face exists, a new disk, D_2, is created. It contains the non-horizontal edges of v_6. When the sweep line reaches v_5, L will contain two disks, D_1 and D_2. L is constructed as a result of querying the disk of each end-vertex of the horizontal edges connected to v_5. We verify the existence of a face with a non-horizontal edge from D_1 and a non-horizontal edge from D_2. For D_1, no such face exists; hence D_1 is removed from L. As for D_2, such a face does exists; thus, the non-horizontal edges of v_5 are added. The verification procedure is appended to the previously discussed procedure when processing the list L.

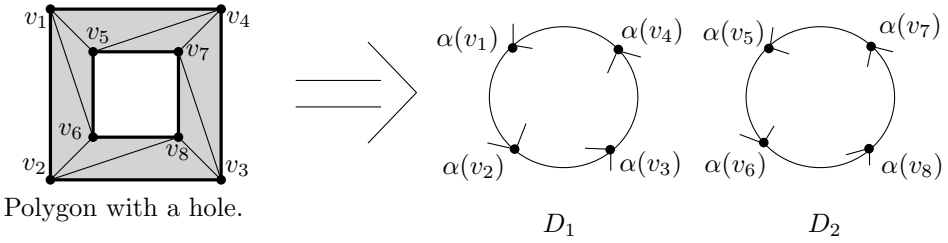


Figure 15: A triangulated polygon with a hole lying on ρ. The thick edges represent the boundary of the polygon. Each vertex has a set of edges going out of the page.

Mixed Vertex Case

The Mixed vertex v_m is a set of edges $E_{v_m} = \{(v_m, v_e) : (v_m.z = \rho \wedge v_e.z > \rho) \vee v_m \text{ is marked as } NOOP\} \cup \{(v_b, v_m) : v_b.z < \rho \wedge v_m = \rho\}$. There are three scenarios in handling v_m: Splitting a disk into two disks, two disks merging to one single disk or replacing the current edges in a disk with E_{v_m}. We need to execute three operations before determining which scenario will best handle v_m. In the first operation, we extract the set of edges starting from vertex v_m. Let $\beta(v_m) \subset E_{v_m}$ and $\beta(v_m) = \{(v_m, v_e) : v_e > \rho\} \cup \{(v_b, v_m) \vee (v_m, v_e) : v_m \text{ is marked as } NOOP\}$ be the set of edges

resulting from the operation. Let $\gamma(v_m) = \alpha(v_m) \backslash \beta(v_m)$ be the edges ending at vertex v_m. For each edge e in $\gamma(v_m)$, we query $\mathcal{T}$ for the disk containing edge e, and store the queried disk in list L. We need to ascertain that the queried disk does not exist in L since a set of edges can belong to the same disk. If the number of elements in L is equal to one, then we need to determine if v_m is a split operation of disk D in L or if the edges in D are updated. If there are two disks in L, v_m is classified as the merging of the disks in L to one disk. If L contains more than two disks, the slicing algorithm is aborted, for $\mathcal{P}$ is in violation of Definition 1.2.2.

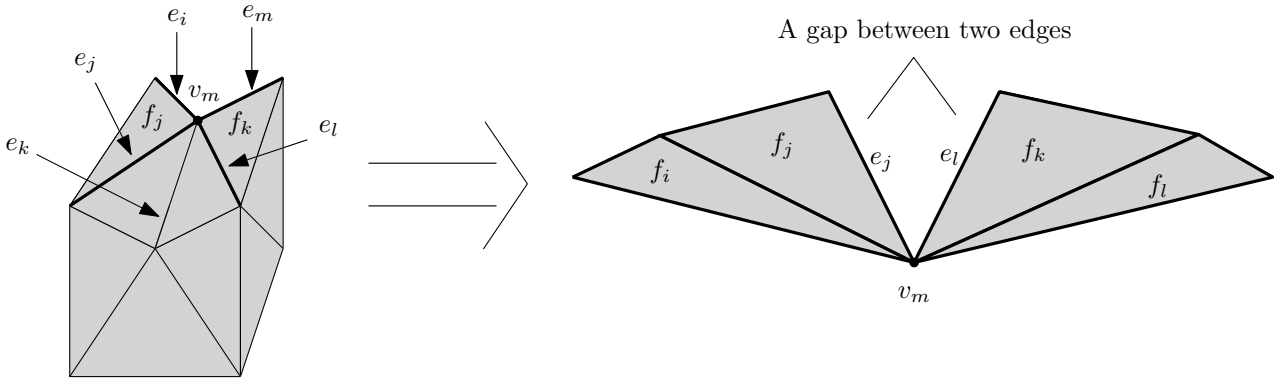


Figure 16: Vertex v_m is discovered by ρ. v_m contains a gap which designates v_m as a split vertex.

In order to determine if v_m is a split of D, we check for any missing face between the set of edges $\beta(v_m)$ as depicted in Figure 16 We accomplish this task by executing the walking algorithm $\mathcal{W}$, as described below:

1. Pick edge e_p from $\beta(v_m)$. Set $e_{initial} = e_p$.

2. Mark e_p as processed.

3. Select face f_p, which is sharing edge e_p and has not been processed. Mark f_p as processed.

4. Extract edge e_n, which is shared by f_p, $e_n \neq e_p$ and is in $\beta(v_m)$.

5. If e_n exists, set $e_p = e_n$ and go to step 2. Otherwise, if the secondary face containing $e_{initial}$ is not processed, set $e_p = e_{initial}$ and go to step 3. Furthermore, check if all the edges in $\beta(v_m)$ have been marked as processed.

6. If all the edges in $\beta(v_m)$ have been marked as processed, designate v_m as an edge replacement case. Otherwise, mark v_m as a disk split case.

The primary function of the walking algorithm is that it verifies whether all the edges in $\beta(v_m)$ have been marked as processed. This function guarantees that there is no missing face sharing two edges in $\beta(v_m)$. If a face is indeed missing, the algorithm will be aborted before processing all the edges. As shown in Figure 16, edge e_k is not included in $\beta(v_m)$. Assuming e_l was the edge selected in step one of $\mathcal{W}$. $\mathcal{W}$ will be aborted before it processes e_i and e_j since $e_k \notin \beta(v_m)$.

Scenario One: Replacing a set of edges belonging to disk D with $\beta(v_m)$.
Remove all the edges in D that contain v_m as an end vertex. Add the new set of edges $\beta(v_m)$ to D. Re-order all the edges in D, and notify $\mathcal{T}$ that D has been modified. $\mathcal{T}$ will make the necessary alterations to the node containing D. These alterations are covered by the *update* operation of $\mathcal{T}$.

Scenario Two: Splitting disk D into two disks.
In this scenario, two disks from D are generated by removing the set of edges D that contains v_m as an end vertex, and then adding the set of edges $\beta(v_m)$ in D. The operation, *PartialSplit*, that splits D into two disks is explained in Section 2.2.

The splitting operation of the disk will return two disks, D_{p_1} and D_{p_2}. We then need to update $\mathcal{T}$. The node N_D that contains D in $\mathcal{T}$ is split into two nodes whereby each node contains the newly partitioned disk. N_D may contain children. It follows that each child of N_D either points to the parent node containing D_{p_1} or

D_{p_2}. The child's parent can be easily detected by determining whether the child's disk is contained in D_{p_1} or in D_{p_2}. This scenario is handled by executing the *Split* operation of $\mathcal{T}$.

Scenario Three: Merging a set of disks in L into one disk, D_m.
This scenario involves extracting the set of edges of each disk in L and adding them into a newly created disk, D_m. The operation of merging a set of disks into one is explained in Section 2.2. Once a new disk is created, we need to update $\mathcal{T}$. The update operation of $\mathcal{T}$ is described in scenario 2 of the start vertex case.

End Vertex Case

The end vertex case is simply the opposite of the start vertex case. The end vertex v_e is a set of edges $E_{v_e} = \{(v_b, v_e) : (v_b.z < \rho \wedge v_e.z = \rho) \vee v_e \text{ is marked as } NOOP\}$. This case usually results in the deletion of a disk in $\mathcal{T}$. When v_e is discovered, we query $\mathcal{D}$ for the set of edges E_{v_e} containing v_e as the end-vertex. Here is the overall algorithm $\mathcal{A}_e$ used to handle a detected vertex, v_e:

1. Query for disk D that contains v_e.
2. Extract the set of edges $\alpha(v_e)$ from E_{v_e}.
3. Remove all the edges stored in $\alpha(v_e)$ from D. This task is accomplished by executing the RemoveEdges disk operation.
4. If the RemoveEdges disk operation returns a disk, update the disk in $\mathcal{T}$ by executing the Update $\mathcal{T}$ operation. Otherwise, remove the disk from $\mathcal{T}$ by executing the Remove $\mathcal{T}$ operation.

$\mathcal{A}_e$ covers several scenarios. The simplest scenario is that of edges $\alpha(v_e)$ merging to v_e. Since all the edges in $\alpha(v_e)$ end at a single vertex, there exists a disk D that

contains $\alpha(v_e)$. D is removed (by executing the *remove* operation of $\mathcal{T}$) from $\mathcal{T}$ after the execution of $\mathcal{A}_e$. The node that contains D is a leaf in $\mathcal{T}$, for D is shrunk to v_e. The deletion of disk D implies that any other disk contained in D was removed at some height of $\mathcal{P}$ before v_e was discovered.

Another scenario that $\mathcal{A}_e$ addresses is that of a disk not being removed from $\mathcal{T}$. As shown in Figure 17, when the sweep line reaches vertex v_5, the edge (v_1, v_5) is removed from disk D. The same is true for v_6, v_{11} and v_{12}. After processing v_{12}, D contains the edges of the upper part of $\mathcal{P}$. When the sweep line reaches vertex v_{16}, the disk has a count of zero stored edges. D is then removed from $\mathcal{T}$.

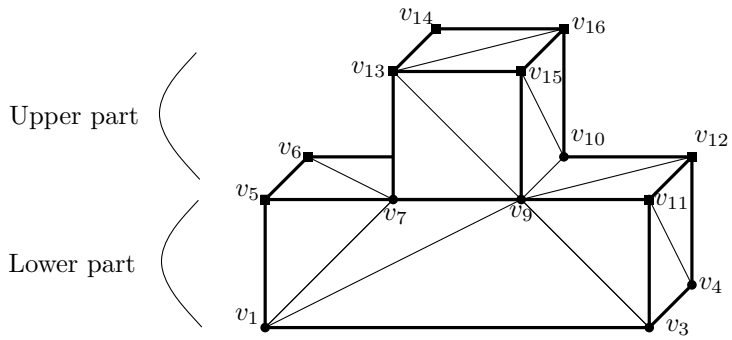


Figure 17: A triangulated $\mathcal{P}$. A square vertex represents an end vertex.

When disk D is removed from $\mathcal{T}$, the node containing D could have children. In Figure 18, when ρ reaches the height of $\mathcal{P}$ at z_2, disk D_1 is deleted since ρ detected a set of vertices marked as end vertices. In this scenario, the children of the deleted node containing D point to the parent of the deleted node. Referring again to Figure 18, when D_1 was deleted from $\mathcal{T}$, the node containing D_2 was a child of the root node. This scenario is covered in the *remove* operation of $\mathcal{T}$.

$\mathcal{A}_e$ also handles the special case shown in Figure 15. Let us assume that vertices v_1 to v_8 in Figure 15 are marked as end vertices. When the polygon lies on ρ, there are two disks, D_1 and D_2, representing the polygon and the hole of that polygon. When the sweep line processes v_1 and v_2, D_1 is reduced to only the set of edges $\alpha(v_3)$

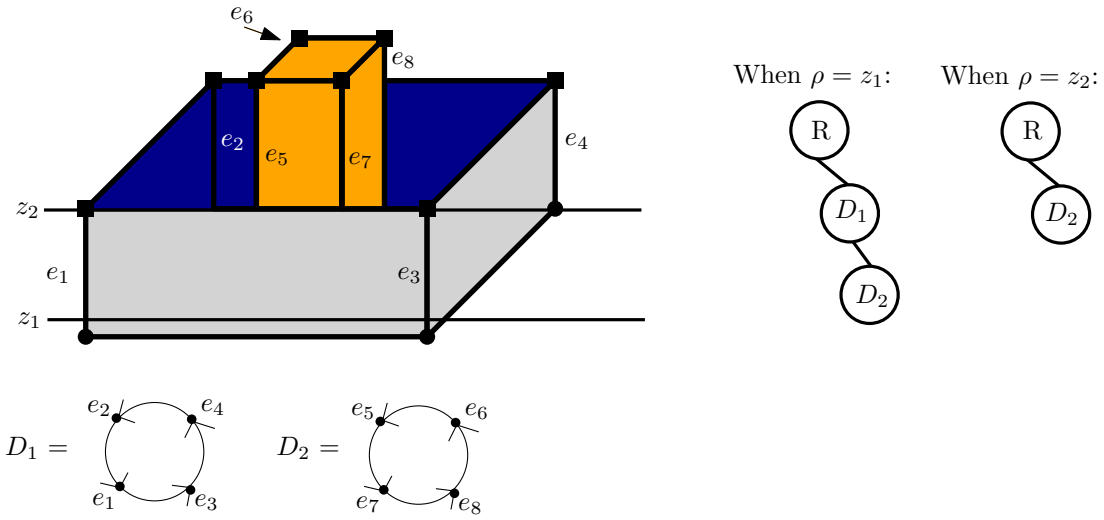


Figure 18: A scenario in which the node of a disk is deleted with children. The square vertices are marked as end vertices.

and $\alpha(v_4)$. D_2 is removed from $\mathcal{T}$ after processing v_7. Finally, D_1 is removed after processing v_4. According to $\mathcal{A}_e$, a disk must exist when an end vertex is detected. The processing of the end-vertex will render the reduction of the disk by $\alpha(v_x)$.

2.4.3 Post-Processing

$\mathcal{T}$ is constructed after all the vertices of $\mathcal{P}$ have been processed. A set of verification algorithms will then be executed to ensure the integrity of $\mathcal{T}$. A sample of these verification algorithms verify the hierarchical relationship between the parent and child nodes. They also check the integrity of the disks. Lastly, $\mathcal{T}$ can be modified to save to the non-volatile memory of the computing device.

2.5 Example of the Slicer Algorithm

In this subsection, we give a simple example of how the slicer algorithm, the polygon nesting tree data structure and the disk interact to each other. Starting from Figure 19, the slicer algorithm detects three vertices v_1, v_2 and v_3 lying on the slicer plane.

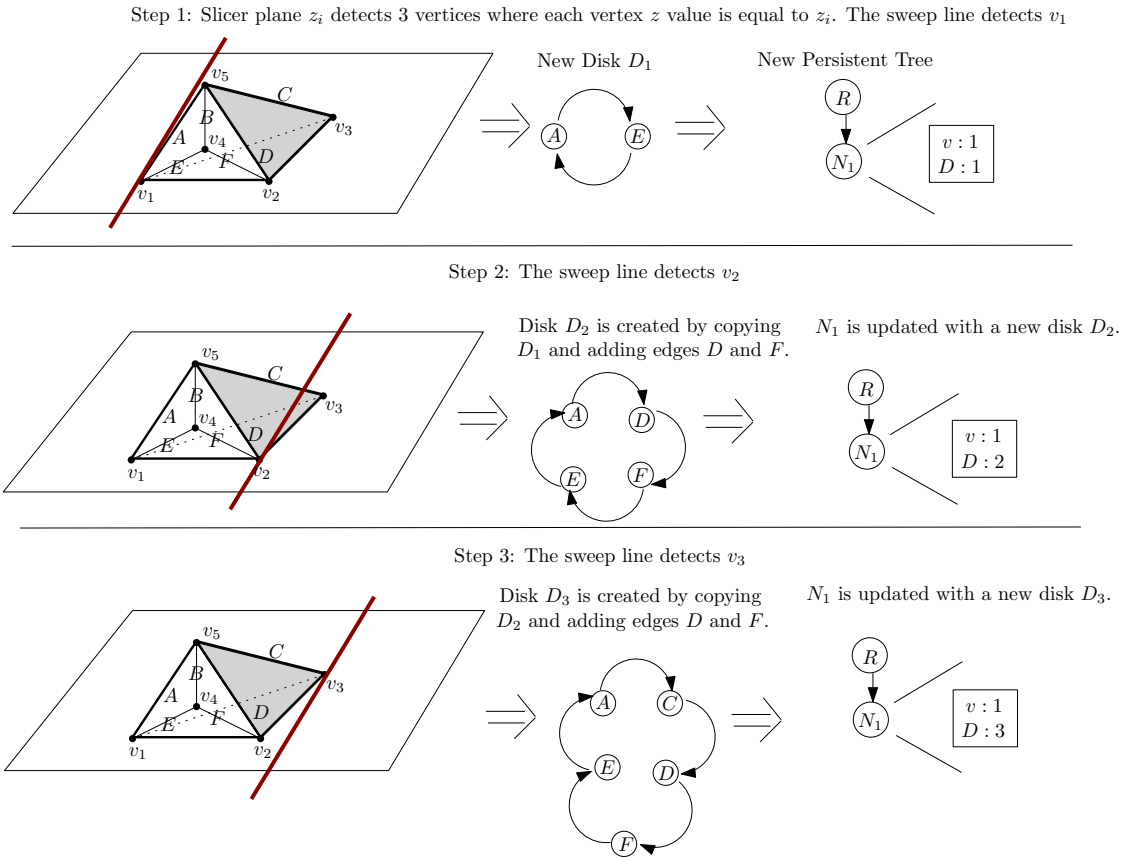


Figure 19: An overview on how the begin vertex is handled by the sweep plane followed by the sweep line. In this particular case, a new disk D_1 is created and is updated twice to contain all the non-horizontal edges of v_1, v_2, v_3.

The sweep (red) line is activated to process each vertex. When the sweep line detects v_1, a new disk D_1 is created with edges A and E since v_1 is marked as begin vertex. As mentioned in previous section, the horizontal edges are not added in the disk. The new disk D_1 is then added to the Polygon Nesting Tree Data Structure $\mathcal{T}$ with version 1. Version 1 represent the first z value of the polyhedron detected on the plane. Afterwards, the sweep line detects v_2. v_2 is also marked as a begin vertex and there is an edge containing v_1 and v_2. The slicer algorithm searches in the persistent

tree for the disk that contains v_1. Once the disk has been found (which is D_1), D_1 is cloned to a new disk D_2 where D_2 now has D and F edges. D_2 replaces D_1 in $\mathcal{T}$ since the slicer plane did not move to the next z-value. The process of handling v_3 is the same as handling v_2.

The sweep plane is then moved to the next z value in Figure 20 which is vertex v_4. Since the sweep plane moved, the version in $\mathcal{T}$ is incremented by one. The sweep line is activated and detects v_4. The slicer algorithm then searches in $\mathcal{T}$ for the disk that contains the edges E and F. The search results of $\mathcal{T}$ returns D_3. D_4 is created by copying the edges of D_3, removing edges E and F and add edge B. D_4 is then added to $\mathcal{T}$ with version 2.

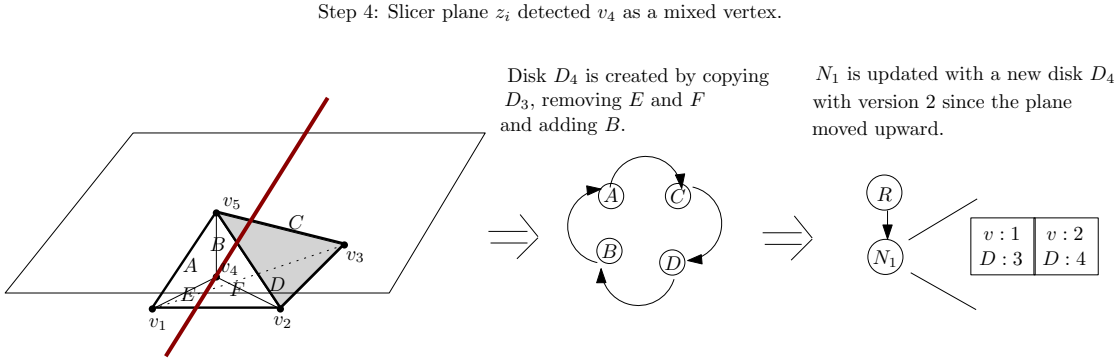


Figure 20: An overview on how the mixed vertex is handled by the sweep plane and line. In this particular case, a new disk D_4 is created from disk D_3 by replacing the edges E, F with B.

Finally, in Figure 21, the sweep plane is moved to v_5. v_5 is marked as an end vertex. The slicer algorithms asks $\mathcal{T}$ for the disk containing the edges A, B, C and D. The search results of $\mathcal{T}$ returns D_4. Since all the edges are in D_4, the node N_1 is then marked as inactive node. That is, N_1 will never be used again for future new disks. N_1 is also kept to generate polygon nesting trees at a specific height h.

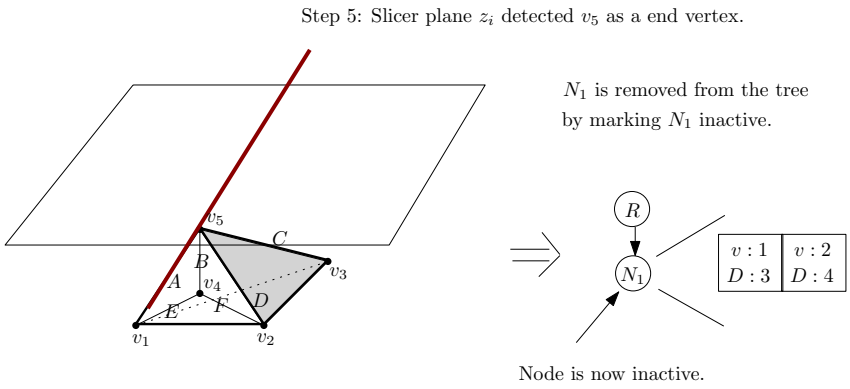


Figure 21: An overview how the end vertex is handled by the sweep plane and line. v_5 marks the node N_1 as an inactive node. An inactive node is referred to a node deleted at a particular version. In this case, the version was 3.

2.6 Running Time

Below is the overall analysis of the Slicer Plane Algorithm based on the sequence steps outlined at the beginning of the section:

1. Importing the set of polyhedra takes $O(|V| + |F|)$ time.

2. The construction of DCEL takes $O(|V| + |F|)$ time.

3. Using quicksort, constructing the sorted array ϖ of vertices takes expected $O(|V| \log V)$ time.

4. Initializing ρ takes $O(1)$.

5. The Slicer Plane Algorithm processes every vertex v in the ϖ. There is $O(|E_v|)$ time of edges processing for every vertex case. Then, an update operation is invoked. An update operation has a running time of $O(cd|E|)$, hence, the Slicer Plane Algorithm has $O(cd|V||E|)$ running time.

If we take the most expensive step, the Slicer Plane Algorithm has a running time of $O(cd|V||E|)$ where $|V|$ is the total number of vertices, $|E|$ is the total number of edges, c is the maximum number of children polyhedra and d is the depth of the hierarchy of polyhedra.

Chapter 3

Implementation of The Polygon Nesting Trees Data Structure and the Slicer Algorithm

This chapter focuses on the implementation details of the deduced algorithm and data structure that were presented in Chapter 2. The first section elaborates on the details of a newly developed application called the Plane Slicer Viewer. The Plane Slicer Viewer integrated and tested the theoretical methods of Chapter 2. Furthermore, the Plane Slicer Viewer includes an essential library known as the Library of Efficient Data types and Algorithms (LEDA), which helped to avoid floating-point errors and provided a set of essential tools for basic geometry computation. Lastly, we discuss the results of the implemented algorithm.

3.1 Implementation Details

This section covers with the implementation details of the Plane Slicer Viewer and the process associated with handling an imported polyhedral object. Also provided in this section is a brief analysis on LEDA's crucial function in the Plane Slicer Viewer application.

3.1.1 Plane Slicer Viewer - An Application to Test the Polygon Nesting Trees Data Structure and the Slicer Algorithm

The Polygon Nesting Trees Data Structure $\mathcal{T}$ and the slicer algorithm were verified by a developed application called the Plane Slicer Viewer. This application can read a polyhedron, construct $\mathcal{T}$ and display a polygon nested tree of a requested height by an user. See Figure 22 for an illustration of the developed application.

The Plane Slicer Viewer's first task is to read the polyhedron from a dataset. We choose the Standard Polygon Format (PLY) as the standard file format for the Plane Slicer Viewer. The primary reason for selecting the PLY file format is that the structure of the file provides the capability of avoiding trivial errors such as the overlapping of faces or the intersection of faces. The avoidance of such trivial errors is accomplished by first providing an array of vertices followed by an array of faces. Each face is composed of a set of vertex indices, and each vertex index is directs to the actual point in the vertices array.

The PLY file structure usually contains at least three major sections: the header, the vertices and the faces. Several PLY files may only contain point clouds. The header describes the remaining format of the file. Several PLY files may differ by providing more information such as the normal and colour of each vertex. The vertices section is a list of x, y, z values. Usually, a vertex is represented by three floating point values. Lastly, the faces section is a list of faces. For our use, each face contains exactly three vertices.

During the processing of a PLY file, the Doubly Connected Edge List (DCEL) is being constructed. The DCEL data structure would process the header section of the PLY file to extract the number of vertices and faces. By default, the Plane Slicer Viewer requires each vertex to have a normal to enhance the visualization of

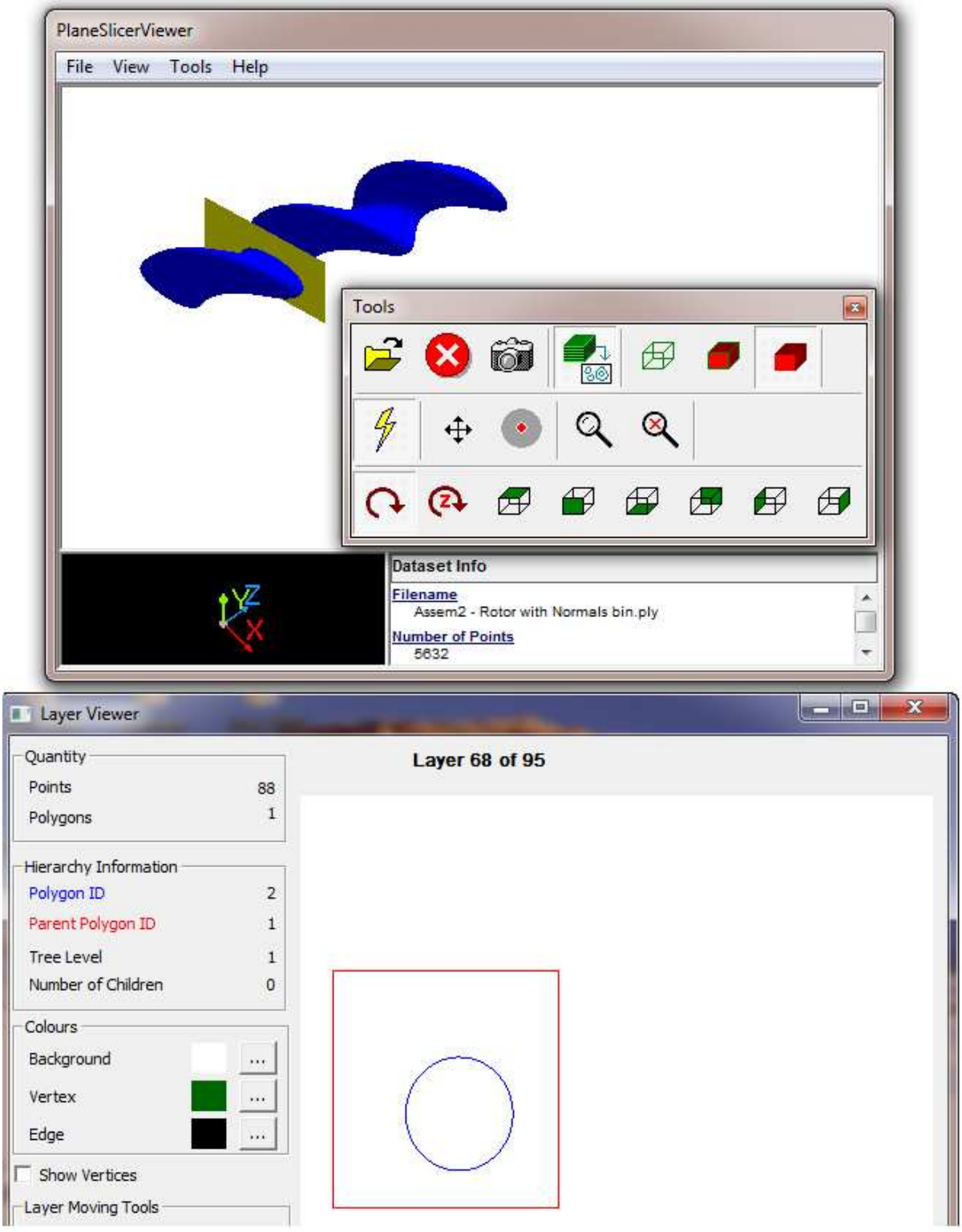


Figure 22: An overview of the Plane Slicer Viewer. The first window shows the imported polyhedron. The yellow plane is the plane slicing approximately the middle of the polyhedron. The second window shows the polygon being produced by the slicing plane.

the rendered object. The defined PLY parser in the application reads all the vertices from the PLY file and stores them in an array known as the Points Array. The Points Array is owned by the DCEL data structure. Each face of the PLY file is then read individually and stored in the DCEL's Face hash table. When a face is being added in the DCEL, the edges representing the boundary of the face are created and stored in a hash table. To satisfy Definition 1.2.2, for each edge $e = (s_p, e_p)$, two directional half-edges are created. One half-edge starts from s_p and ends at e_p and vice versa for the other twin half-edge. A half-edge is pointing to the next-half edge of the same face and contains a pointer to the twin half-edge. If Definition 1.2.2 is violated, a half-edge will point to more than two twin half-edges. This violation is detected during the construction of DCEL.

After processing a PLY file, the constructed DCEL goes under a validation algorithm. The validation algorithm asserts the following statements:

- Each face of polyhedron $\mathcal{P}$ contains exactly three vertices
- Each face contains exactly three circular linked half-edges
- Each half-edge contains a twin half-edge

If one of the statements fails to be asserted, the Plane Slicer Viewer would report the imported polyhedron as an violation of Definition 1.2.2. The construction and validation of DCEL is relatively quick based on the number of edges and faces, as shown in Table 2 [1]. For example, the Sample Bottle dataset has a considerable amount of vertices and faces, and took approximately 54 milliseconds to construct and validate its DCEL.

After the construction of DCEL, a set of drawing element arrays are created in order to render the imported polyhedron. These drawing arrays (in OpenGL:

[1]The results were obtained by executing a 32-bit version of Plane Slicer Viewer running on AMD A8-3870 3.3Ghz CPU, 8 GB of DDR3 DRAM Frequency 667 Mhz, running on 64-bit version of Windows 7.

Dataset Name	Number of Vertices	Number of Faces	Construction and Validation Time of DCEL (milliseconds)
68 VISTAS junio 2001	19	34	1
Stone Crusher	408	812	6
Sample Bottle	8352	16700	54
Assembly 2 - Rotor	5632	11260	102

Table 2: Construction and validation time of Doubly Connected Edge List for each polyhedron.

vertex arrays) are offered by the OpenGL [19] framework in order to reduce the rendering time when rotating, panning or zooming a polyhedron. OpenGL provides a framework to quickly develop an application to view and render two- or three-dimensional polyhedra.

After creating the necessary drawing objects, the Slicer Algorithm is then executed, as described in Chapter 2, Section 2.4.1. Once the Slicer Algorithm has processed the polyhedron, the Polygon Nesting Trees Data Structure is created and can be used to query a Polygon Nesting Tree at a specified height. In Figure 22, the second window shows a particular Polygon Nesting Tree instance. The outer polygon represents the bounding box and the inner polygon represents the sliced polyhedron object.

3.1.2 Usage of C++ Computational Geometry Library: Library of Efficient Data types and Algorithms

A major part of the Slicer Algorithm implementation was completed by heavily utilizing a free edition of the library called the Library of Efficient Data types and Algorithms (LEDA). LEDA [12] provides an essential data type (in C++ terms, class)

called rational. The rational class provides software developers trouble-free issues regarding floating-point errors. As explained in Chapter 1, Section 1.4, there are cases in which several computational geometric operations return a false positive value. If we use the rational class, the computation results are always mathematically correct. However, there are two disadvantages to using the rational class:

- The memory footprint (in bytes) of each rational instance is much larger than a C++ primitive float point data type.
- The computation performance is 30-100 times slower than the conventional double data type [20].

To deal with the computation performance issue of rational points, the rational class provides an operation known as normalize. This operation helps to reduce the memory footprint and increase the computing performance. The normalize operation runs the GCD algorithm to find the largest common factor of the numerator and denominator. As explained by LEDA online documentation, a rational class is represented by a numerator and a denominator. Table 3 shows the major difference of the Slicer Algorithm running time when each rational point of polyhedron is normalized and non-normalized. When a set of rational points are not normalized, the Slicer Algorithm running time is tripled, as shown in Table 3.

Aside from the rational points, there are other computational geometric functionality of LEDA that provides:

1. To determine if a vertex is above, on or below the slicing plane.
2. To find the intersection point of the slicing plane and an edge.
3. To generate a randomized vertex (a vertex with a random x value, y value and a defined z value).

	Slicer Algorithm (seconds)
Normalized Rational Points	8.14
Non-normalized Rational Points	25.84

Table 3: A table representing the major difference between a normalized and non-normalized set of rational points when they are put in running time in the Slicer Algorithm. The dataset used to capture the slicer algorithm time is Assembly 2 - Rotor.

4. To sort a list of vertices.

Item 1 uses the d3_rat_plane::side_of operation. An instance of the d3_rat_plane is created first which represents the slicing plane of the Slicer Algorithm. The side_of operation returns -1, 0, 1 if a vertex is on the left side, on the plane, or right side of the plane. For our purposes, -1 value represents the vertex v below the plane, 0 for v on the plane and 1 for v above the plane. This operation is intensively used to classify if v is a begin, a mixed or an end vertex. Since the d3_rat_plane class is used, the returned value of side_of is always correct. The precision of the returned value is integral to avoiding marking the incorrect vertex class. See Figure 23 for an example of the consequences of incorrectly classifying a vertex.

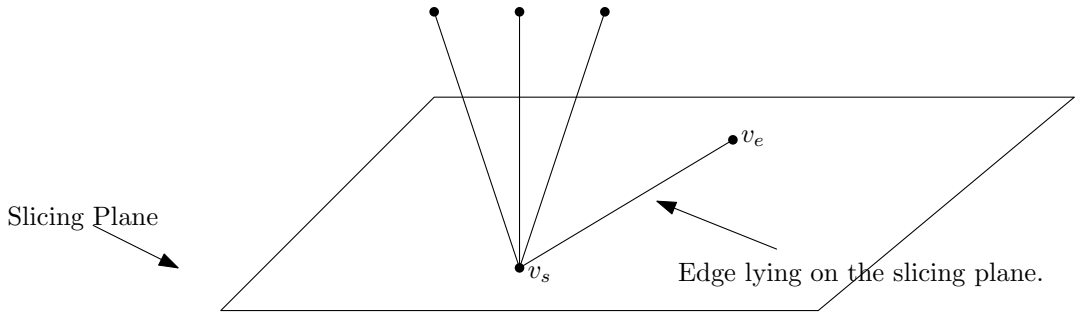


Figure 23: A begin vertex v_s with an edge (v_s, v_e) lying on the slicing plane. If the LEDA library was not used, there is a chance of v_e being below the plane due to a floating-point error. The Slicer Algorithm would mark v_s as a mixed vertex.

Item 2 is heavily used when a disk is being sliced at a particular z height. The d3_rat_plane class provides an operation called intersection. This operation takes an edge e and the d3_rat_point q as parameters, and returns a boolean value indicating whether or not e intersects the plane. If e does intersect the plane, the intersection point is stored in q.

For item 3 and 4, LEDA provides additional helper operations to randomize the values and sort an array. In the leda::array$< T >$ class, an operation called sort is used to sort any element of type T. For our purpose, we are sorting the list of vertices where each vertex is of the type d3_rat_point. Lastly, we use the random_source class which provides functionality to generating a vertex. This vertex is used for ray-tracing where the technique is integrated in the *Has* or *Inside* Disk Operation, described in Section 2.2.

3.2 Results of the Slicer Algorithm and the Polygon Nesting Tree Data Structure

This section covers the bench-marking performed on the Slicer Algorithm. We first begin by discussing the performance of the Slicer Algorithm on slicing different types of spheres to capture the total running time based on the number of vertices and the number of faces. We then analyze the running time on four sample polyhedra extracted from the 3DVIA website [21].

The first part of the analysis involves measuring the performance of the Slicer Algorithm by slicing a sphere. A sphere is composed of stacks and slices. A stack of a sphere is a horizontal layer of the sphere with a fixed thickness. A slice of a sphere is a sub-volume of a sphere of azimuth thickness d (starting from θ and ending at $\theta + d$) and an inclination thickness of 180 degrees. Figure 24 shows a sphere composed of 9

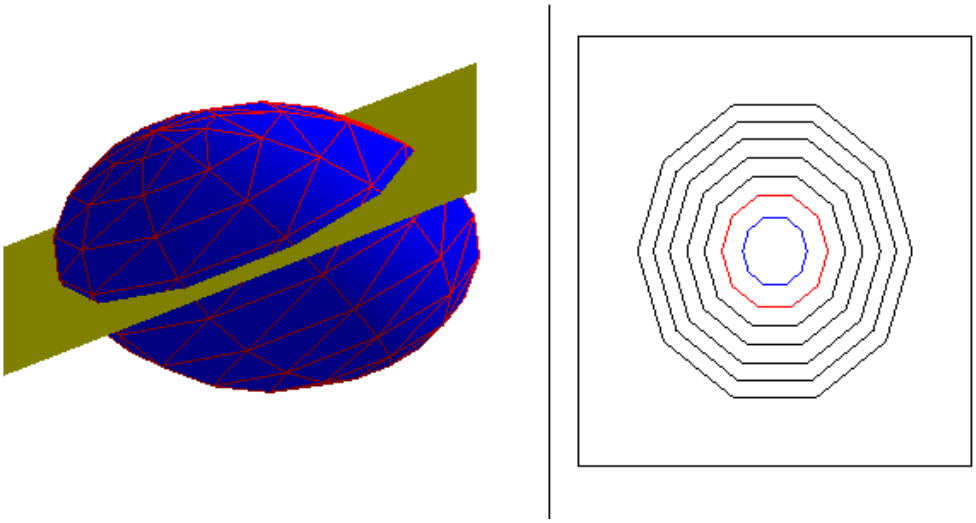


Figure 24: An example of a sphere containing another sphere, up to 9 levels with a resolution of 10. To the right of the sphere, there is a sliced layer containing 10 polygons. The square represents the bounding box of the polyhedron.

stacks and 9 slices. Since the number of stacks and slices are equal, we say that the sphere has a resolution of 10.

As shown in Figure 25, if we start with a low resolution of 5 and increase it to 30, the running time of the Slicer Algorithm is polynomial. The increase of the running time makes sense since every vertex, excluding the north and south vertex, is a mixed vertex case. In Section 2.4.2, the walking algorithm is executed to determine if a vertex is a split case or an update case. The walking algorithm is quite expensive since the algorithm has to process all the edges of the discovered vertex. The second expensive operation performed for all the vertex cases is the creation of a new disk. Creating a new disk is also expensive since we need to sort the edges in order to create faces, as explained in Section 2.2.

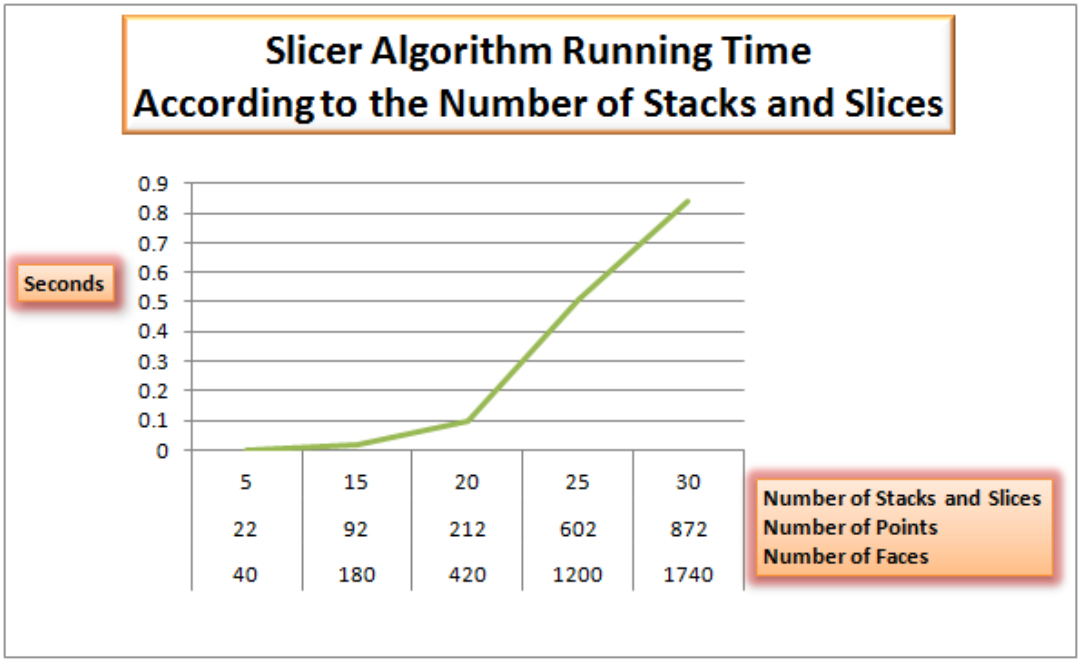


Figure 25: A graph showing the running time of the slicer algorithm where the resolution of the sphere is increased.

Another interesting performed experiment is the ability of a sphere to contain another sphere. Again, as shown in Figure 25, when the plane is sliced near the center of the sphere, there are 9 polygons. The smallest polygon is at level 9 of the Polygon Nesting Tree. This experiment is performed to determine the Slicer Algorithm running time when a sphere is contained within another sphere. In Figure 26, a graph shows the total running time of the Slicer Algorithm based on the number of tree levels. The running time is polynomial as the number of tree levels is increased. If we increase the resolution level to 15 and 20, the graph line in Figure 27 and 28 is still the same as Figure 26 except the y-axis values of each graph is increased by a polynomial value.

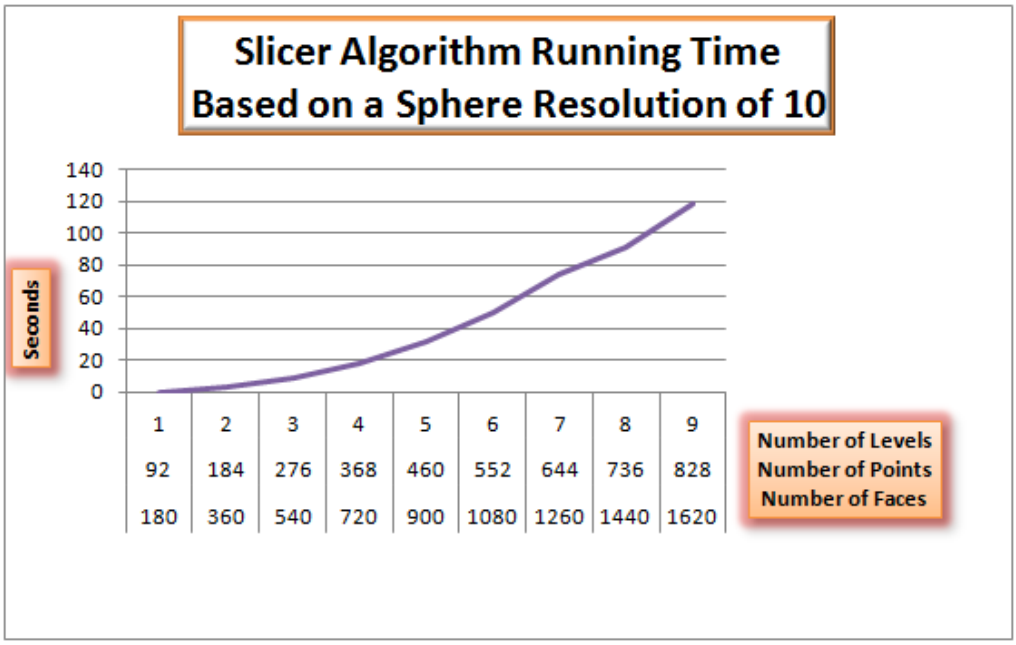


Figure 26: A graph showing the running time of the slicer algorithm where the resolution of each sphere is frozen at 10 and contains a sphere within a sphere.

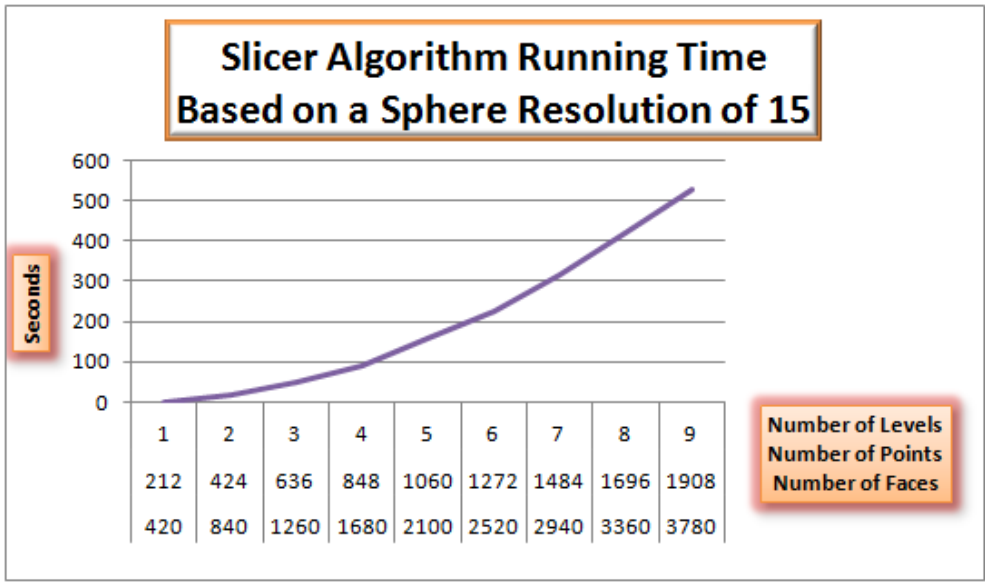


Figure 27: A graph showing the running time of the slicer algorithm where the resolution of each sphere is frozen at 15 and contains a sphere within a sphere.

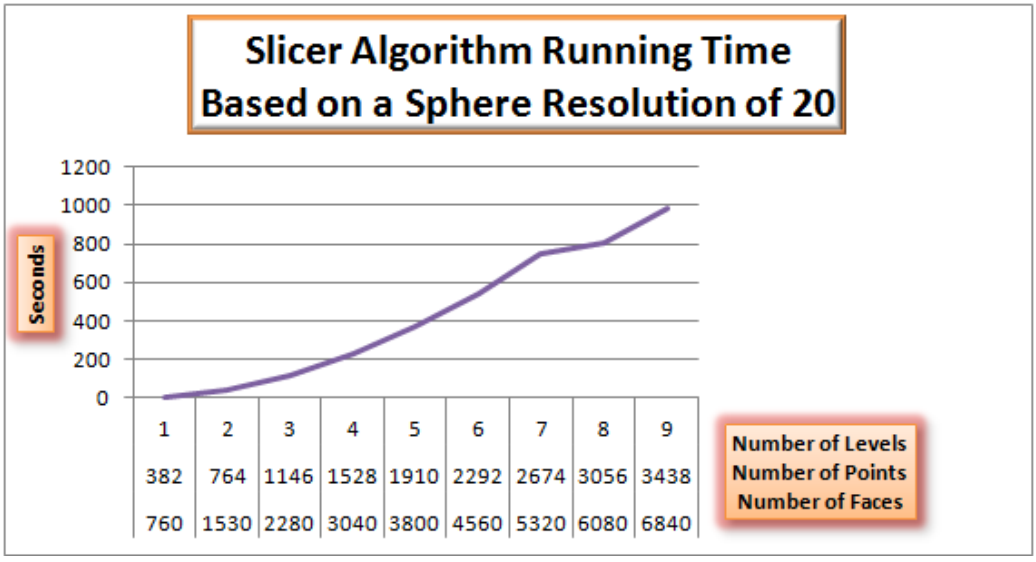


Figure 28: A graph showing the running time of the slicer algorithm where the resolution of each sphere is frozen at 20 and contains a sphere within a sphere.

Lastly, another interesting experiment related to the spheres that was performed involved containing several spheres within a single sphere. In Figure 29, the root sphere contains four smaller spheres. Each smaller sphere contains 4 even smaller spheres. This experiment was executed to capture the running time. Interestingly, in Figure 30, the graph line is the same as that found in Figures 26, 27 and 28.

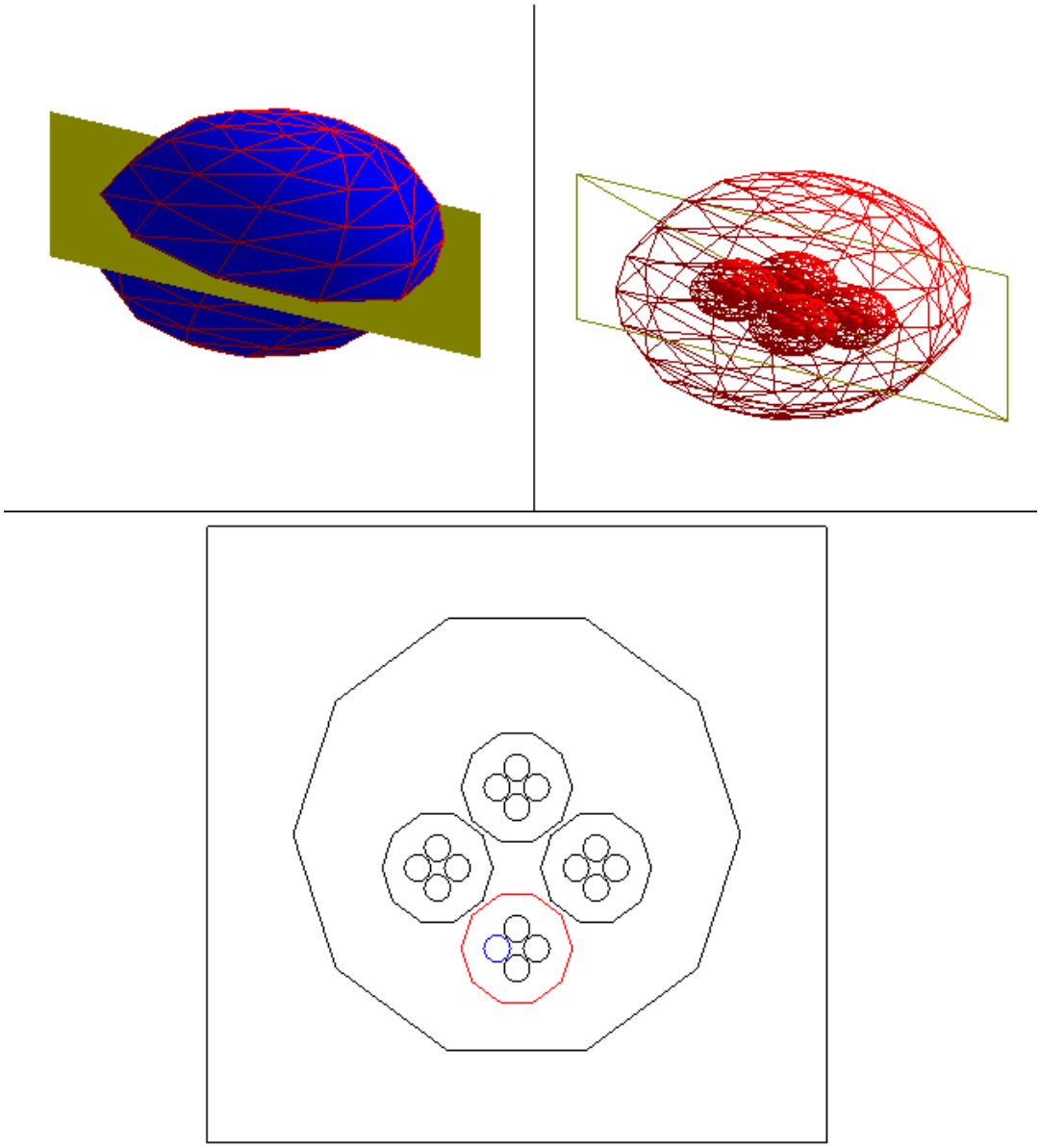


Figure 29: An illustration of four spheres being contained within one large sphere. In this example, the number of levels in the Polygon Nesting Tree is 3.

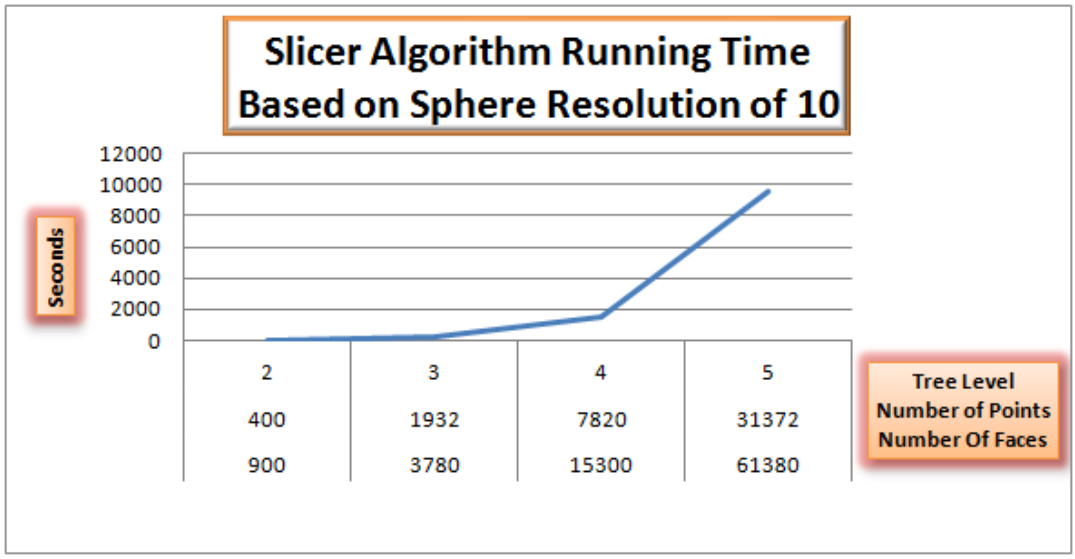


Figure 30: A graph showing the running time of the slicer algorithm. Each sphere contains 4 additional spheres.

Lastly, we captured the Slicer Algorithm running time of each sample dataset found in the 3DVIA website [21]. As shown in Appendix A, most of the polyhedra are represented by a small Polygon Nesting Tree height. Most of the time, a disk is created, updated several times and is then destroyed. The *Sample Bottle* polyhedron has a higher running time because when the slicer plane began slicing the bottle, it started with two disks. Later, the slicer plane merged the two disks into one and then split again at a higher z-coordinate. For a visualization of each dataset shown in Table 4, please refer to Appendix A.

Dataset Name	Number of Points	Number of Faces	Running Time (Seconds)
68 VISTAS junio 2001	19	34	0.01
Stone Crusher	408	812	0.33
Assembly 2 - Rotor	5632	11260	8.19
Sample Bottle	8352	16700	276.51

Table 4: Demonstration of the results of the Slicer Algorithm on a set of datasets from 3DVIA datasets. Refer to Appendix A for a visualization of each dataset.

Chapter 4

Conclusion and Future Work

We have developed a Polygon Nesting Tree data structure $\mathcal{T}$ with a slicer algorithm that accepts a polyhedron as an input, and outputs a set of queried polygon nesting trees at specific interval heights of the polyhedron. An idea for constructing $\mathcal{T}$ is to use a disk to keep track of the edges while the slicer plane traverses the polyhedral object. For every vertex that we process, we create, update, merge or split a disk and then perform the update on $\mathcal{T}$. $\mathcal{T}$ uses the fat-node persistent technique to record every change in updates. The technique allows us to query $\mathcal{T}$ to produce a nesting polygon tree at height h of the polyhedron. The nesting polygon tree is generated by locating the closest version i in $\mathcal{T}$ that represents h and traverses $\mathcal{T}$ at i. During the traversal, we build the polygon nesting tree. Each node stores a polygon which was produced through the slicing of the disk at height h at version i or less.

Chapter 3 explains the implementation details of the developed Plane Slicer Viewer application. The application uses an essential C++ library called LEDA [12], which provides tools to perform geometric computations. Using the application, we were able to obtain results based by experimenting with the Polygon Nesting Tree height, increasing the resolution of a polyhedron and running a set of polyhedra obtained from 3dvia [21].

4.1 Future Work

4.1.1 Parallel Computing to Build a Polygon Nesting Tree Data Structure

The developed slicing algorithm used to build $\mathcal{T}$ is designed for sequential computing. Since most computing devices contain several CPUs and faster memory access, the algorithm could be modified to construct $\mathcal{T}$ by slicing the polyhedron into multiple sections and then sending each section to a CPU. Each section would build a part of $\mathcal{T}$. Once all the sections have been processed, they are all merged into a single $\mathcal{T}$. Experiments are needed to determine the total time spent building $\mathcal{T}$. Also there is a need to develop an algorithm that would determine the correct number to partition the polyhedron. There is also a need for an algorithm that would coordinate the sections that need completion and the sections that need processing. Furthermore, a merging algorithm is needed to merge all the smaller parts of $\mathcal{T}$ to a completed $\mathcal{T}$.

4.1.2 Decomposing each polygon face into a set of triangles

One of the requirements mentioned in chapter 2 is for the face to contain only three vertices. The slicer algorithm can be modified to process a face of n vertices. One quick way to adapt to this new requirement is to triangulate the polygon. The triangulation of a polygon would of course increase the computational workload. However this work can be accomplished during the parallel computing stage. Otherwise, more cases needed to be handled in the slicer plane algorithm to avoid creating unnecessary disks.

List of References

[1] S. McMains. "Layered manufacturing technologies." *Communications of the ACM* **48**(6), 50–56 (2005).

[2] J. Leno. "Jay leno's 3d printer replaces rusty old parts." "`http://www.popularmechanics.com/cars/jay-leno/technology/4320759`" (2009).

[3] S. Devados and J. O'Rourke. *Discrete and Computational Geometry*, page 1. Princeton University Press, first edition. ISBN 0691145539 (2011).

[4] A. Beutelspacher and U. Rosenbaum. *Projective Geometry: From Foundations to Applications*, pages 165–172. Cambridge University Press, first edition (1998).

[5] J. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. "Making data structures persistent." *Journal of Computer And System Sciences* **38**, 86–124 (1989).

[6] C. Bajaj and T. Dey. "Polygon nesting and robustness." *CSD-TR-918* (1989).

[7] P. Arruda. *Plane Sweep Topological Hierarchy - Sorting Algorithm.* Honours project, Carleton University (2008).

[8] S. Choi and K. Kwok. "A topological hierarchy-sorting algorithm for layered manufacturing." *Rapid Portotyping Journal* **10**(2), 98–113 (2004).

[9] 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," SMA '99, pages 285–295. ACM, New York, NY, USA. ISBN 1-58113-080-5 (1999).

[10] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. "Classroom examples of robustness problems in geometric computations." *Proc. 12th Europ. Sympos. Alg* **vol. 3221**, 702–713 (2004).

[11] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in JAVA*, pages 612, 615. John Wiley & Sons, Inc., first edition. ISBN 0471193089 (1998).

[12] K. Mehlhorn and S. Naher. *LEDA: A Platform for Combinatorial and Geometric Computing*, pages 103, 104. Cambridge University Press, first edition (1999).

[13] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry, Algorithms and Applications*, pages 29–33, 47. Springer, third edition (1998).

[14] G. Barequet and M. Sharir. "Filling gaps in the boundary of a polyhedron." *Computer Aided Geometric Design* **12**, 207–229 (1993).

[15] R. Pagh and F. Rodler. "Cuckoo hashing." In F. Heide, editor, "Algorithms ESA 2001," volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer Berlin Heidelberg. ISBN 978-3-540-42493-2 (2001).

[16] C. A. R. Hoare. "Quicksort." *The Computer Journal* **5**, 10–16 (1962).

[17] "12 standard specification for additive manufacturing file format (amf) version 1.1." "`http://www.astm.org/Standards/F2915.htm`". DOI: 10.1520/F2915-12, ICS Code: 35.240.50 (2012).

[18] B. G. Baumgart. "Winged edge polyhedron representation." Technical report, Stanford, CA, USA (1972).

[19] Shreiner, Sellers, Kessenich, and Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, pages 115–127. Addison-Wesley Professional, eighth edition. ISBN 978-0-321-77303-6 (2013).

[20] "Rational numbers." "`http://www.algorithmic-solutions.info/leda_guide/number_types/rational.html`". Algorithmic Solutions Software GmbH (2006).

[21] "3dvia community." "`http://www.3dvia.com`" (2013).

Appendix A

Visualization of Polyhedra

This Appendix contains four sample polyhedra obtained from the 3dvia [21] community. The results of the Slicer Algorithm running time is shown in Table 4. Each figure is separated by two sides. The left side, the polyhedron being sliced at a particular height, and at the right side, the Polygon Nesting Tree is being generated by the slicer plane.

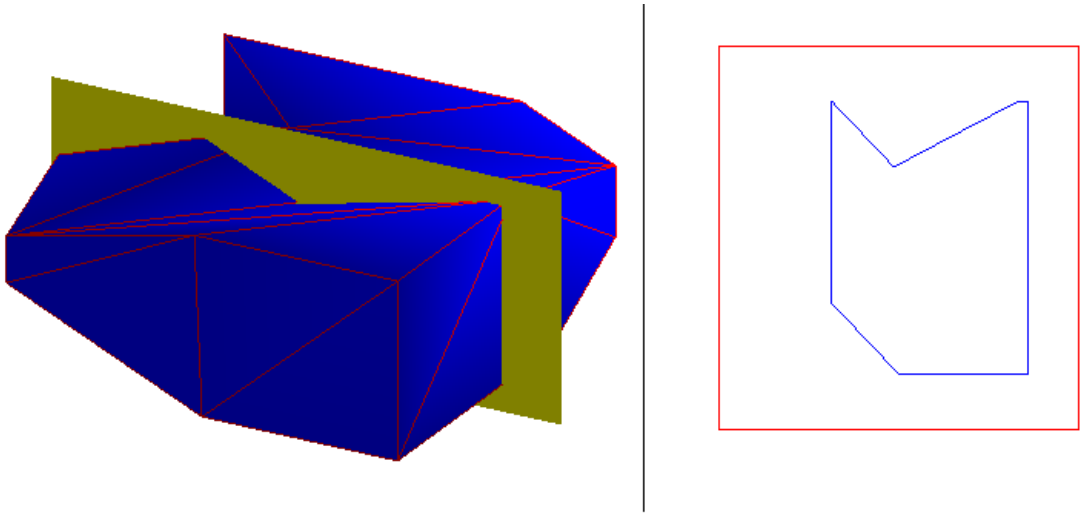


Figure 31: VISTAS junio 2001 Polyhedron

The polyhedron shown in Figure 31 contains a dent which is detected by the slicer plane. The size of the dent is originally as small, but it progressively increases when

the slicer plane is moved to a higher z-coordinate. The polyhedron is represented by one disk.

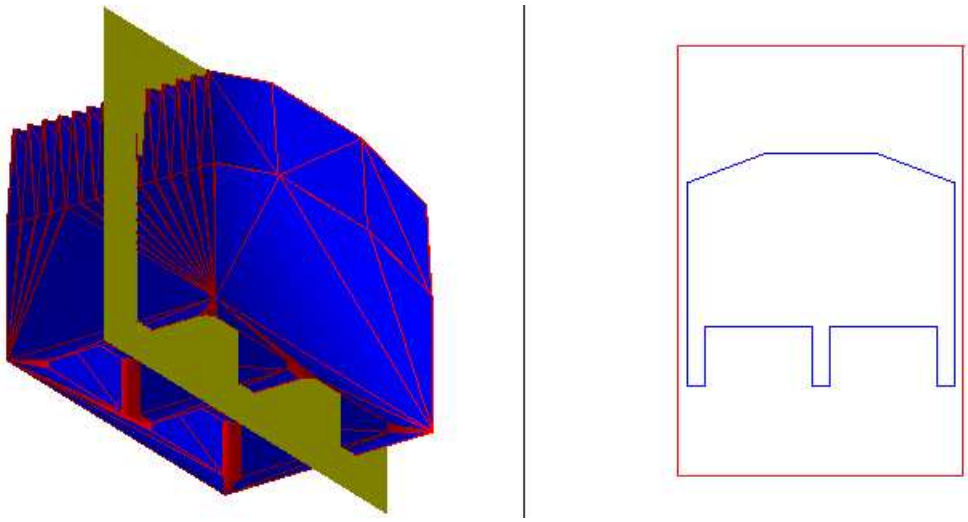


Figure 32: Stone Crusher Polyhedron

The polyhedron shown in Figure 32 contains several dents at the top and bottom. The polyhedron is represented by a disk which is updated 406 times during the execution of the Slicer Algorithm.

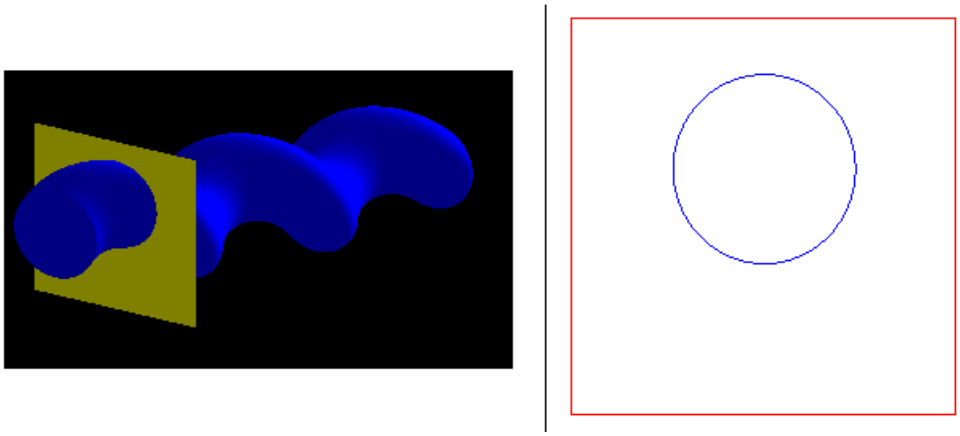


Figure 33: Assembly 2 - Rotor Polyhedron

The polyhedron portrayed in Figure 33 is an interesting object used for automobiles. During the execution of the Slicer Algorithm, a disk was created, and updated 5630 times. Fascinatingly, when the rotor is sliced at a particular height, it is always remains a circle of the same radius with a displaced center.

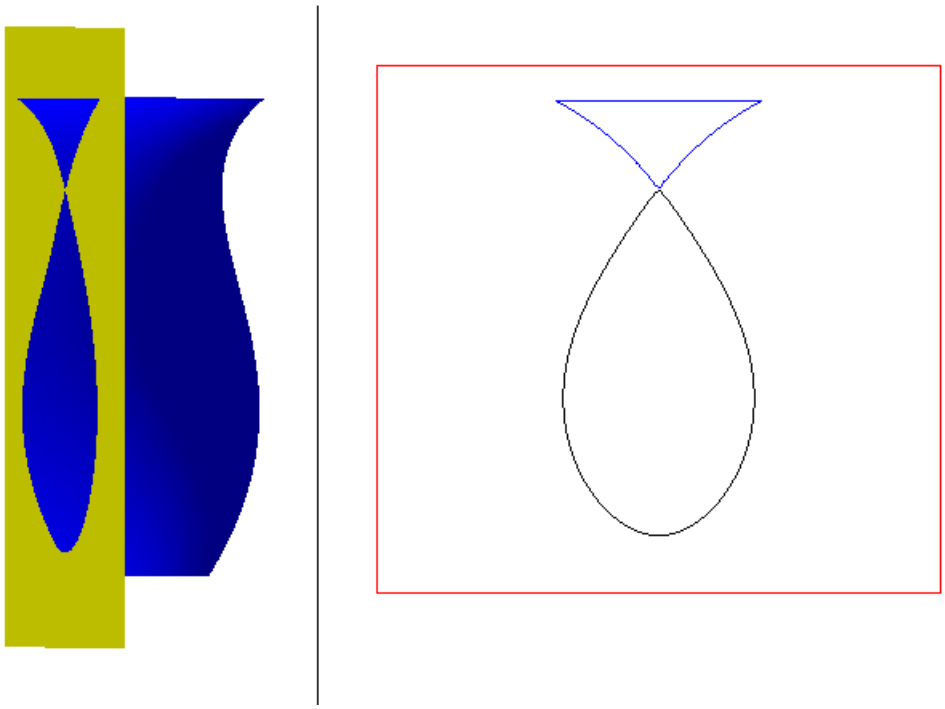


Figure 34: Sample Bottle Polyhedron

Finally, the polyhedron shown in Figure 34 is a compelling interesting object for the Slicer Algorithm. When the slicer plane starts at the lowest z value, two disks are created. Eventually, the two disks are merged into one disk. When the slicer plane reaches to 3/4 of the height of the object, the merged disk is again split into two disks. Overall, the Slicer Algorithm accomplished four feats. It created two disks, split one disk into two disks, and merged two disks into one disk. Finally, it updated the handled disks in the Persistent Polygon Nesting Tree data structure a total of 8346 times.