Route Planning Algorithms for 3D Printing

A Project

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

in

Computer Science

University of Regina

By

Shulang Lei

Regina, Saskatchewan

August 2015

Abstract

General 3D printers use Fused Deposition Modeling (FDM) and Stereolithography (SLA) technologies to print 3D models. However, turning the nozzle on and off during FDM or SLA extruding will cause unwanted results. This project created an experimental 3D model slicer named Embodier that generates continuous extruding paths whenever possible. This enables 3D printers to draw out the printing layers accurately in a stable manner. Furthermore, the slicer partitions the outlines to tree structures for efficiency and applies flooding algorithm for water-tightness validation. Lastly, a 3D printing simulator is also created to visualize the printed paths in 3D for a more intuitive review of the Embodier slicer. The end result is that we have discovered that not only a single continuous-extruded-path slicer is possible, it can also be optimized for performance and robustness in practice.

Keywords: Continuous extruding path, Slicing, 3D printing, Flooding Algorithm, Space Partitioning.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Dr. Boting Yang, for his constant support, encouragement and consideration throughout my graduate studies. I greatly appreciate his advice and patience in helping me improve my Graph theory knowledge and writing skills. His expertise and insight is also invaluable for guiding me through this project when I was frustrated and stuck in my research.

Besides my supervisor, I would like to thank Dr. Samira Sadaoui and Dr. Howard Hamilton for being on my committee. I would like to express my gratitude to my committee, Dr. Boting Yang, Dr. Howard Hamilton and Dr. Samira Sadaoui for the time and efforts they put in reading and examining my report. I am particularly grateful for the valuable suggestions and comments Dr. Howard Hamilton made on my report writing. I also like to thank Dr. David Gerhard for sharing his experience in 3D printing, which helped me solved the bridging problem.

I also acknowledge the financial support from my employer, University of Regina, which makes this work possible.

Last but not least, I would like to thank my family: my wife, my parents, my parent-in-laws, and to my dear daughter for supporting me spiritually throughout writing this report and my life in general.

Contents

List of Figures

Chapter 1

Introduction

1.1 History of 3D Printing

Human have been making tools for millions years. Some of the methods involved are carving, sharpening, grinding, sawing, drilling, machining, etc. These methods are shaping objects by removing stuff: start with greater raw material and cut them into smaller desired shapes.

3D printing was originally utilized by the industry as "Additive Manufacturing". Instead of subtracting from the solid raw material, material is being successively layered by robot controlled printer head to build desired shapes.

1.1.1 SLA

3D printing was first invented by using the process called "Stereolithography" (SLA) by Hideo Kodama of Nagoya Municipal Industrial Research Institute [1] [2]. Ultraviolet light curable liquidized plastic, or liquid resin, is exposed to a moving ultraviolet laser that hardens the plastic layer by layer, and the object is finally constructed.

After 3D printing made its first appearance, the process is improved and made

actionable by 3D System Corporations [3] with their contribution in the software: the STL file format and the accompany slicing and infill strategies. Following that, metal and many more others are also being used as raw material with similar methods.

1.1.2 FDM

Later, an extruding method "Fused Deposition Modeling" (FDM) was developed by S. Scott Crump and commercialized by Stratasys [4]. Raw material such as plastic, metal, wood, cement and others are made into filaments. A heated nozzle is used to melt the filaments. A worm-drive then pushes the filament into the nozzle to extrude the melted material into layers, in order to reconstruct the object. By being low-cost and providing ease of use, FDM technology is currently popular among the hobbyist and consumer-oriented market.

1.2 Motivation for This Research

Current low-cost home 3D printers are priced between $500 to $2000 at the time of writing. Most of these printers are utilizing FDM and some are based on SLA. Their design concept is to melt the printing material (mostly plastic), and lay the melted plastic layer by layer to reconstruct the digital 3D object.

While these printers are starting to occupy the market, their ease of use, performance and reliability are still in question. While the hardware will undoubtedly improve in time, the improvements done in the software side can rapidly impact more users with a lower cost.

In this project, a combination of algorithms and data structures are applied to the routing planning processes for an open-sourced slicer named Embodier. The end results demonstrated by a 3D printer simulator show that wide range of improvements can be made using better software designs.

1.3 How 3D Printing Works

3D printing can be generally defined by following five major steps: 3D Modeling, Slicing, Filling, G-Code Generation and Printing.

1.3.1 3D Modeling

There are different ways to model an object that is 3D printable. Currently, the most popular way is using the polygon mesh format because of the abundances of the tool-sets available. In the polygon mesh format, a single 3D object has a volume that is bounded by 2-dimensional patches (surfaces). Because the minimum requirement to define a surfaces is three points, the minimum units for a polygon mesh object are triangles (see Figure 1.1).

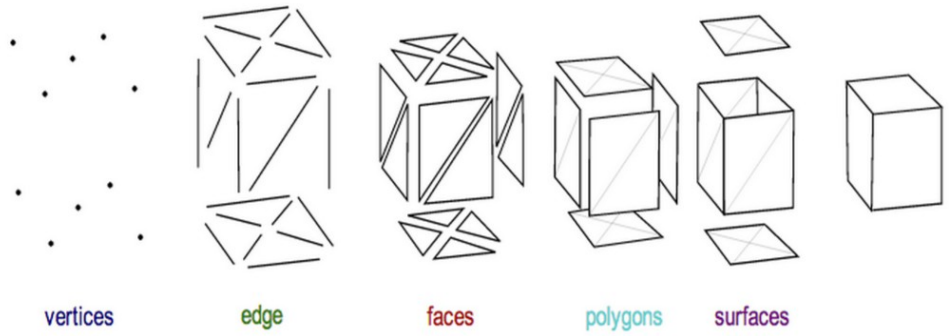


Figure 1.1: Polygon mesh composition

1.3.2 Water Tightness

A polygon mesh needs to have some volume to be printable. For example, an infinitely thin wall can not be printed. The way of checking for water tightness is to verify if the polygon mesh is “manifold”.

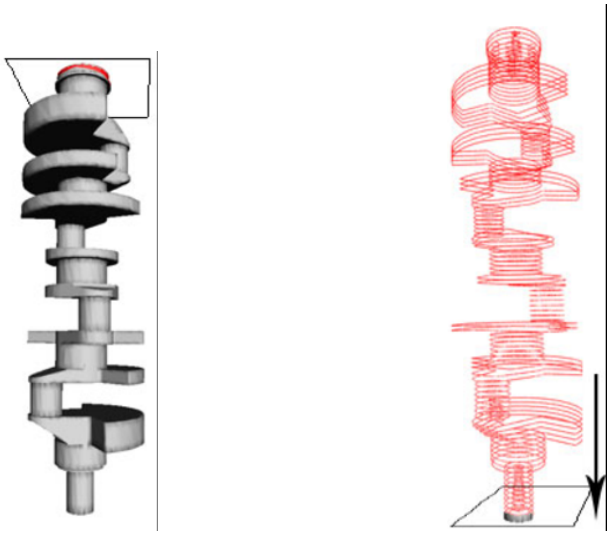
1.3.3 Manifold

A mesh is a manifold if each edge is incident to only one or two faces [6] (see Figure 1.2).

Figure 1.2: Left: Non-manifold. Right: Manifold.

1.3.4 Slicing

Slicing refers to the cutting of the polygon mesh into layers along the z-axis which is the vertical axis of the 3D printer (see Figure 1.3).



(a) Object to be sliced. (b) Object that is sliced.

Figure 1.3: Slicing a model

1.3.5 Filling

After slicing a 3D mesh into a series of 2D shapes, each outline (perimeter) needs to be filled with regular lattice (infill pattern).

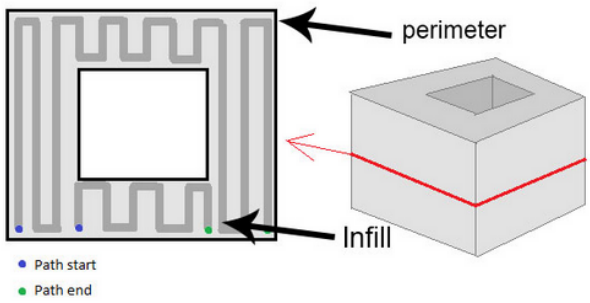


Figure 1.4: Filling a perimeter

1.3.6 G-Code Generation

G-Code is a language used to tell the printer head how to move and draw out the layer. Optimizing this aspect is the main focus of this project.

1.3.7 Printing

Printing is completed by heating the printer head to a point where the extruding plastic melts; next, the melted plastic is extruded along the path directed by G-Code. The final object is then constructed layer by layer.

1.4 Limitations of Home 3D Printers

The major limitations in current home 3D printing technologies are speed and reliability. It often takes hours to print a medium-sized object. In addition, it is not uncommon for the printing alignment to fail during the middle of the print. Therefore the two fundamental problems are:

1. Fast printing for bigger objects

2. Unsupervised long printing projects

These two problems are hard to solve with current home 3D printers. This is why 3D printing is still a hobbyist activity and rarely being utilized by non-tech-savvy

users. In addition, the cost for businesses that provides 3D printing service is still too high.

1.5 Improvement Approaches

There are already several efforts being made to improve 3D printing experience.

A low-cost optical fabrication printer, the Peachy Printer [8] uses laser to increase fault tolerance so that the printing path will not be blocked by faulty prints. Combined with liquid submersion to prevent gravity collapse, the outcome is very promising.

Another approach is to use better G-Code generation algorithms. Since printing paths, especially the filling paths can be arbitrary, we can make use of better algorithms. One example is "Skin-frame Method" [7]. It generates a frame structure right next to the "skin", the surface of the object. The material need is only 15.0% of a solid object (see Figure 1.5).

The "Skin-frame method" also attempts to improve the speed and reliability by applying better G-Code generation algorithms.

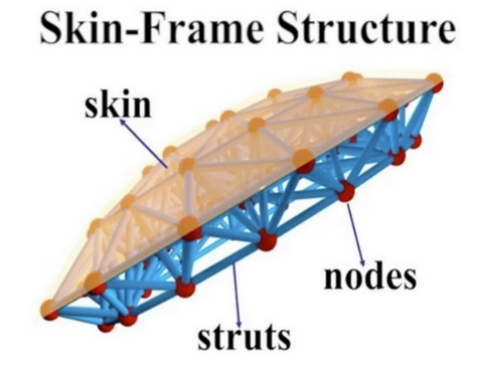


Figure 1.5: Skin frame structures

1.6 Problems This Project Addresses

A backlash happens when there are gaps in the biting surface between the gears: any reverse action in gear rotation will result a random pausing (see Figure 1.6). Pausing the extrusion during the movement of the printer head is done by reversing the worm drive to create a tiny suction that holds the melted plastic. Backlash happens in 3D printer whenever it pauses the extrusion during printing, resulting uncontrollable inaccuracies for timing.

The rationale of this project is to improve the generation of G-Code path such that a continuous path is being applied whenever possible rather than multiple-path extrusions that require the extrusion to be paused during movements. In addition, a continuous extrusion path improves speed and efficiency.

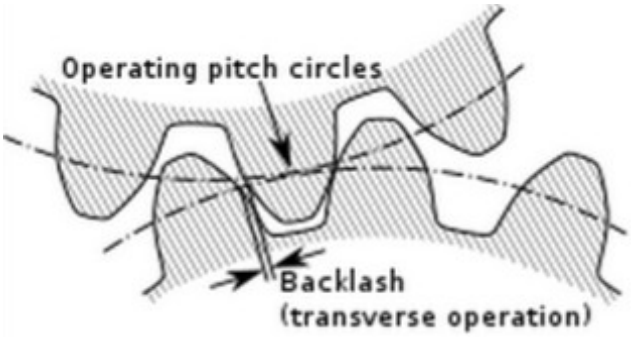


Figure 1.6: Backlash

Furthermore, even with stereolithography, coordinating the positions where the extrusion needs to be turn on-and-off is complicated.

The Peachy Printer is a first low-cost stereolithography printer that uses only the stereo-audio jack and microphone jack as the space controls. However, having to coordinate the on-and-off of the laser during extrusion of a single layer has delayed the project from release to pre-orders [9].

Another example of the the problem created by on-and-offs during extrusion is bridging (see Figure 1.7).

When the printer head tries to move across the long space in the center of the

object, the printer head has a probability of leaking plastic during non-print moves. Thus those "jumps" need to be eliminated as much as possible. If the printing path was generated by a better algorithm, one "jump" for each layer would be sufficient and the bridging would be easier to fix in this case.

Figure 1.7: Bridging

1.7 Continuous Single Path Extrusion

How can we generate a nice single continuous extrusion path? After slicing of the polygon mesh, we end up with a series of 2D layers, which can be considered as a 2D graph with fully connected nodes and edges. Certain nodes need to be traversed in order to support other layers that are built on top of them. However we are assuming that not all of the edges need to be traveled for the sake of easier single path generation. That means sacrificing the reconstruction accuracy of the digital 3D object for efficiency in the scope of this project. Therefore we can visualize the extrusion path as Figure 1.8.

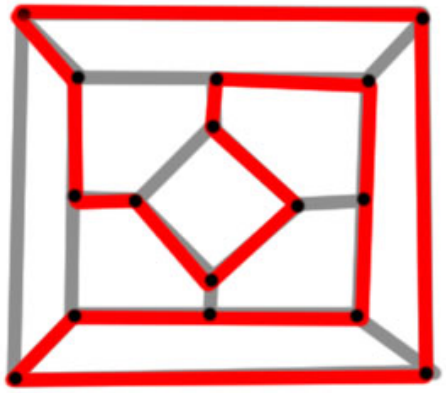


Figure 1.8: Single path extrusion

Since there guaranteed to be an edge between any two nodes (because all of the nodes are inside a two-dimensional Euclidean plane and the printer head can travel anywhere within the plane), we can simplify each layer to be a set of nodes and the main task is to get a single continuous path that visits them all at least once.

There are two flaws of this approach:

1. There will be missing edges, such as the outline that forms the "skin" of the object;

2. There will be surplus edges, such as the edges that cross the object in which there is a hollow space.

Fortunately, 3D printing materials are often as versatile as wood. These flaws can be overcome by performing simple actions: The first flaw can be fixed by applying plastic filler afterwards. Second flaw can be fixed by adding a small bridging structure across the space so that the path finding algorithm can be guided to pass through. Finally, simply cut the bridging structure out with a knife, a drill or a chisel.

Note that post-processing of the G-Code path to add new paths that fix these two flaws will be considered for future development. It is only for the duration sake of this project that we leave out these features.

Chapter 2

Route Planning Algorithms

We need a routing algorithm which given a set of nodes on a two-dimensional Euclidean plane will produce a nice single extrusion path so that the printer head can travel along with its melted plastic extruding at all times, whenever it is possible.

2.1 Traveling Salesman Solving Algorithms

Firstly, a typical traveling salesman problem can be described as follows: Given a list of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once. Let us see how a traveling salesman solver works on the Chinese cities [10]:

Figure 2.1: Chinese cities

After application of the Concorde TSP solver, we have the result within 0.029% Optimal [11] after 1.7 million seconds (472.222 hours) (see Figure 2.2).

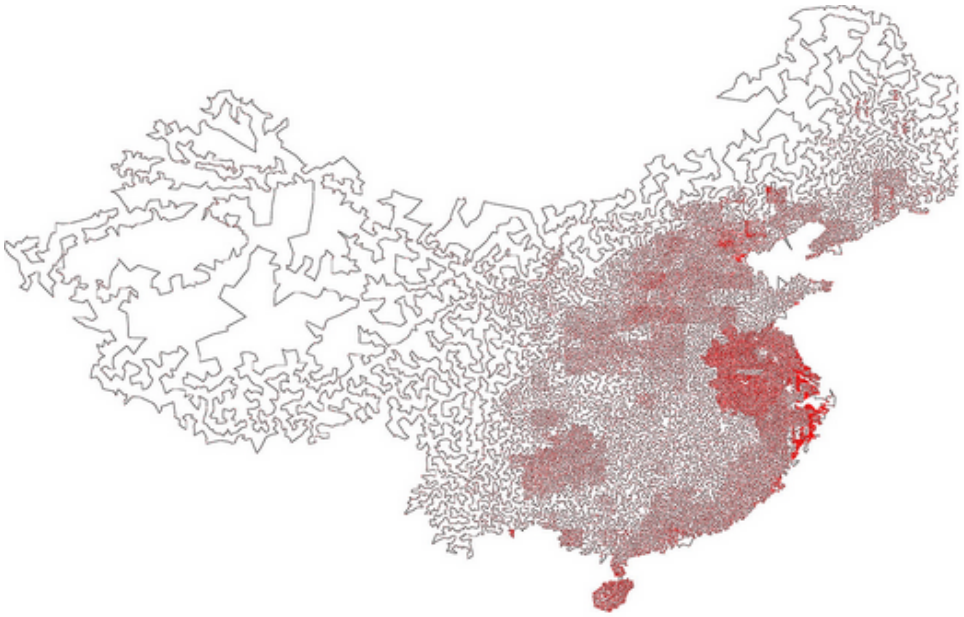


Figure 2.2: A single path tour produced by concorde TSP solver

Unfortunately, the TSP algorithm is not fast enough for our purpose. As well as the unacceptably long running time, there are other problems with the result generated from TSP solver.

The first problem is that the path generated by the TSP solver does not travel the outlines (perimeter) completely: this is because the algorithm is designed to get the shortest path possible.

The first problem of not traveling the edges enough is that the printed object will result in a very rough surface and will require massive filling job – not very practical.

The second problem of not traveling the edges enough is creating deep cracks from the surface. Looking at Figure 2.2 and it is obvious that there will be many cracks if we extrude the plastic that way.

Furthermore, TSP is an NP-complete problem and most of the exact algorithms are exponential time, resulting in the long running times.

Therefore we can conclude that for TSP solving algorithms, the advantages are:

- It generates path that traverses all nodes.
- It generates path that traverses only the minimum distance.

The disadvantages are:

- It generates paths that do not traverse all edges, which produces unsatisfied results.
- It may have a very slow run time.

We want a better algorithm which generates a path such that:

1. It will traverse all nodes.
2. It will traverse all edges.
3. Run-time of the algorithm needs to be fast.

2.2 Eulerian path/circuit Algorithms

The definition of an Eulerian path/circuit is promising for what we are searching for: a trail/circuit in a graph which visits every edge exactly once. Here we made up an example for demonstration as shown in Figure 2.3.

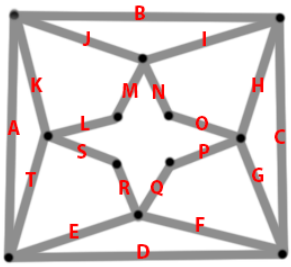


Figure 2.3: The Eulerian circuit is ABCDEFGHIJKLMNOPQRST (In Order)

It is also trivial that the path will travel every node in a graph that is fully connected. In addition, there exists an algorithm that finds the Eulerian Path/Circle in linear time.

Lastly, only a specific type of graphs have the Eulerian path/circuit. Fortunately, we have discovered a way to convert each layer to this type of graph. Let us see how this would work.

2.2.1 Eulerian Graph and Conversion

An Eulerian graph is a graph containing an Eulerian circuit. A graph is Eulerian if and only if it has no odd degree nodes. The degree of a node is the number of its neighbors.

Let this be the “even degree rule”. To convert a non-Eulerian graph to a Eulerian, we simply make some change that changes the degree of any node that violates the even degree rule.

There are two ways to change the degree of odd degree nodes to even:

1. Connect a pair of odd degree nodes as shown in Figure 2.4.

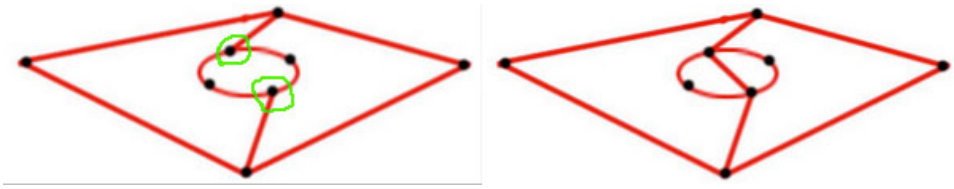


Figure 2.4: Connecting a pair of odd degree nodes

2. Remove the edge that connects a pair of odd degree nodes as shown in Figure 2.5.

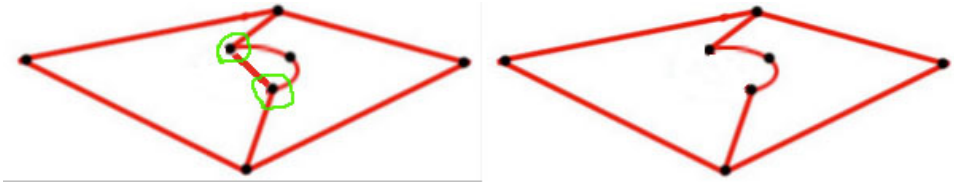


Figure 2.5: Disconnecting a pair of odd degree nodes by removing one edge

We can apply these two methods repeatedly until all odd nodes are removed. Be aware that there are different priorities and situations for each method to be applied. We will discuss the details in the implementation section.

One apparent problem is that the two methods only remove a pair of odd degree nodes each time – what if there is odd number of odd degree nodes? According to the handshaking lemma, for a graph with node set N and edge set E [13]:

$$\sum_{n \in N} deg(n) = 2|E|$$

Therefore every finite undirected graph has an even number of nodes with odd degree. Thus this problem does not exist.

2.2.2 Hierholzer Algorithm

The oldest algorithm that finds the Eulerian circuit is called Fleury's algorithm. However, it is inefficient comparing to the Hierholzer Algorithm. The Hierholzer algorithm can find a Eulerian circuit very quickly and is easy to implement [12]:

1. Choose any starting node v, and follow a trail of edges from that node until returning to v.

2. Find another node v that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from v, following unused edges until returning to v, and join the tour formed in this way to the previous tour.

3. Repeat until all edges are included, the traversed tour is now an Eulerian circuit.

Let us look at an example of the Hierholzer algorithm:

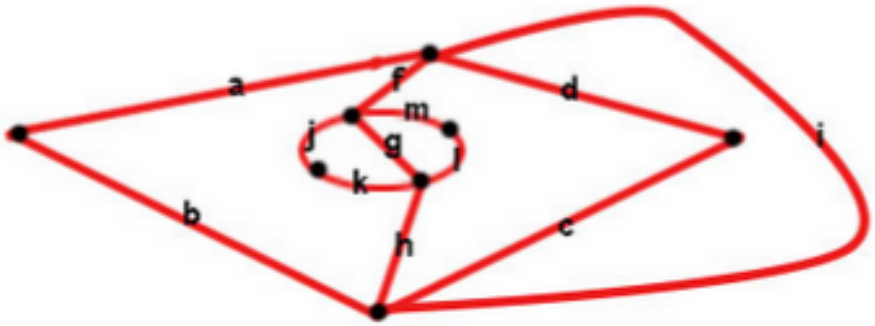


Figure 2.6: A graph without any odd degree nodes

1. First circuit: a,b,h,k,j,g,l,m,f.

2. Second circuit: d,c,i.

3. Joined two circuits: a,b,h,k,j,g,l,m,f + d,c,i.

4. No more edges left.

5. The final Eulerian circuit for the original graph is: a,b,h,k,j,g,l,m,f,d,c,i.

2.3 Design of Embodier Slicer

Concluding from the above analysis, we have a clear and detailed step by step implementation design for the Embodier slicer:

1. Parse the polygon mesh into a set of triangles.
2. Generate a series of planes along the z-axis.
3. Calculate the intersection of each plane and the whole set of triangles to get the outlines (perimeters).
4. Generate the infill pattern for each outline.
5. Transform each layer such that it is a Eulerian graph.
6. Use the Hierholzer algorithm to find a Eulerian circuit for each layer.
7. Generate G-Code from the Eulerian circuits.

Chapter 3

The Embodier Slicer

The Embodier slicer has the following steps in technical detail:

1. Parses binary and ASCII STL files into a set of triangles.
2. Slices polygon meshes with a list of planes.
3. Partitions the space of each layer using a quad tree.
4. Performs collision detection for each leaf of the quad tree, and stores the Boolean results in the collided nodes of the quad tree.
5. Performs ray-tracing water-tight check for each layer, and generates the initial flooding positions.
6. Applies a flooding algorithm to detect water-tight space, and stores the Boolean results in the flooded nodes of the quad tree.
7. Converts each layer to a Eulerian graph.
8. Applies the Hierholzer algorithm to the Eulerian graph to find the Eulerian circuit that traverses all edges and nodes.

9. Infills the quad tree partitions with selected user-defined patterns (not yet implemented).

10. Generates G-Code.

The Embodier slicer is implemented as a headless command line program. This makes future integration with a Web or Desktop interface straightforward.

The programming language of choice is Clojure, which is a functional programming language based on the JVM. The design of this language focuses on idiomatic functional semantics, multithreading and distributed computation models.

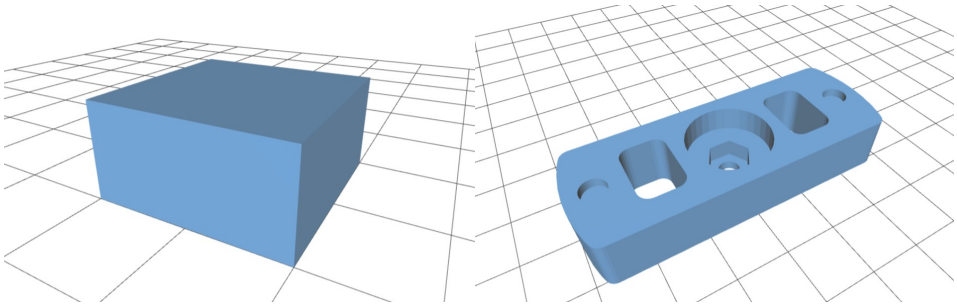
These elegant design of Clojure makes scaling up the performance and features using multiple servers or clusters possible. In addition, the JVM makes Operating System configuration across different platforms simple and easy.

3.1 Test Models

For our testing, we use two 3D models. These models are:

- A simple 20mm solid box (12 triangles)
  - embodier.stl.slicer/resources/stl/asc.stl .

- A complicated 3D printer hot-end part with holes in it (about 900 triangles)
  - embodier.stl.slicer/resources/stl/hotend_v2.stl .



(a) 20mm box (b) Rostock mini hot-end holder

These are ASCII STL files that contain polygon meshes represented in text.

3.2 Implementation

The functions are divided into these name-spaces:

1. Core – gets the parameters from the command line and connects to all other namespaces to execute the tasks.
2. File – parses binary and ASCII STL files using the GLOSS library.
3. Slice – uses plane-triangle intersection to slice the polygon mesh (in a collection of triangles form).
4. Tree – generates an breadth-first search quad tree space partitioned data structure using axis-aligned bounding box (AABB) collision detection.
5. Flood – uses flooding algorithm to find the inner area of a layer.
6. Eulerian – generate Eulerian circuit for layers.
7. Gcode – generates G-Code from result data structures in memory.
8. Draw – draws a variety of data structures for visualization, mostly for debugging reasons.
9. Util – provides some LISP macros to make debugging easier and to shorten some expressions.

3.2.1 Core

The major task for the core is to take care of the command line parameters[1]. Tools.cli library is used. The command line should be of the format:

[1]Source code is given in Appendix A.1.

embodier –help –stl [stl-file-name] –gcode [gcode-file-name]

The Parameters have the following meanings:

- help – provide the typical instructional help text for the program.
- stl – the next parameter gives the input STL file.
- gcode – the next parameter gives the output G-Code file.

Besides this, the "-main" function is the entry point of the whole program.

3.2.2 File

The File namespace reads the STL files[2]. It is implemented with the GLOSS library. There are two kinds of STL files: The binary STL and the ASCII STL. Gloss uses "Codec" to pattern match the file stream. This makes the file parsing fast and quick to develop.

The binary codec is "b-triangle". It represents a triangle as 12 little-endian floating numbers that contains an normal (3) and three vertices (9), followed by an unsigned integer.

The ASCII codec "a-triangle" is similar. It represents the triangle by string floating numbers and surrounded by text tags.

The "parse-stl" function parses the files and stores the triangles in a vector of maps for further processing.

[2]Source code is given in Appendix A.2.

3.2.3 Slice

The task of the Slice namespace is mainly to get the intersection between a set of triangles and a sequence of Z axis planes[3].

At first, a series of Z planes are generated by the function "gen-planes". These are a set of parallel planes that are separated from each other according to the nozzle diameter (printing resolution).

Then it calculates the intersection between each plane and the set of triangles with the "slice" function.

The "triangle-plane-inc" function finds the intersection subject to three conditions:

1. If the entire triangle sits on the plane, the function returns the whole triangle.
2. If one segment of the triangle sits on the plane, the function returns the segment line.
3. If a point of the triangle sits on the plane, the function returns the point.
4. If no intersection is found, the function returns nil.
5. Otherwise, the intersection of one side of the triangle and the plane is returned by the function.

Using our hot-end example (see Figure 3.1b), one of the slices appears as shown in Figure 3.2.

[3]Source code is given in Appendix A.3

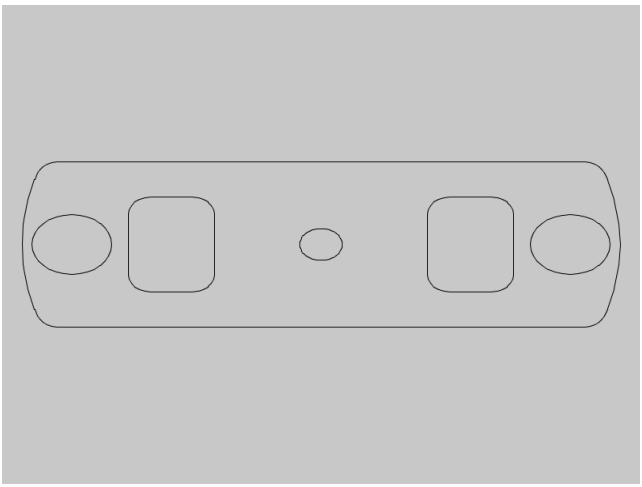


Figure 3.2: A slice of the hot-end model

Be aware that the slice is not as nice as it looks. It looks like a single nice path, but it actually has many segments of small but separated lines. Thus we need flooding algorithms to figure out where is the bounded inner area.

3.2.4 Tree

The Tree namespace implements the data structure and related methods for each layer[4].

When designing the flooding algorithm, the speed of the algorithm comes into consideration. If the flooding algorithm performs an exact simulation of how water floods a space from drop to drop, our algorithm would have to flood the scene pixel by pixel to achieve similar effects. This will be slow, and consume a large amount of memory.

The solution is to use **Space Partitioning**. If there is a big chunk of empty space, we want it to be represented by one single unit instead of many.

Also, we want the data structure storing these space units to be a **Tree**. So that the access time to any unit are reduced from O(N) to $O(\log N)$.

We are currently using a 4-way tree such that each node of the tree has exactly

[4]Source code is at Appendix A.4.

4 children. Thus it is a **Quad-tree**, which means we are quaternary partitioning the space.

Furthermore, we are using **Axis-Aligned Bounding Box**(AABB) to speed up the collision calculation. Therefore all of the area units here are square AABBs.

Lastly, instead of using a linked structure to store the tree, we use an **array** (vector) to store it in Breadth First Searching Order. The indexes of nodes will be arithmetically representing their positions, ancestors and descendants. As a result, most of the querying operations cost will be O(1) time.

Each node cell in the array contains a **Boolean value**. False means the node has not collided with any line, true means the node has collided with at least one line. All descendants of a False node has False values, thus the highest level False nodes are leaf nodes.

The quad-tree is generated by the "generate-tree" function. Basic explanation of the algorithm in steps are:

1. Get the bounding area of the whole slice as an AABB, and represented it at the root node.

2. Get the minimum bounding area representing by a leaf node, which it is a square with the width of the nozzle diameter – as the smallest unit that the printer can handle.

3. Let the root node width be R, the leaf node width be L, and calculate the size of the BFS order array with

$$\sum 4^0, 4^1, 4^2, ..., (\frac{R}{L})^2$$

4. Initialized the Array with nil values. Iterate through each element of the array and for each element:

(a) Calculate the bounding area by walking from root to current node. $O(\log_4 N)$ time.

(b) Test for collision between the bounding area and the slice: if collided, go down another level and test the children again; else when not collided, marked as a leaf node and no further test for descendants are needed. The result is stored as Boolean in the current array element.

Therefore tree generation is $O(N \log_4 N)$.

Apply these algorithm to our test case of simple box, the result is shown as Figure 3.3.

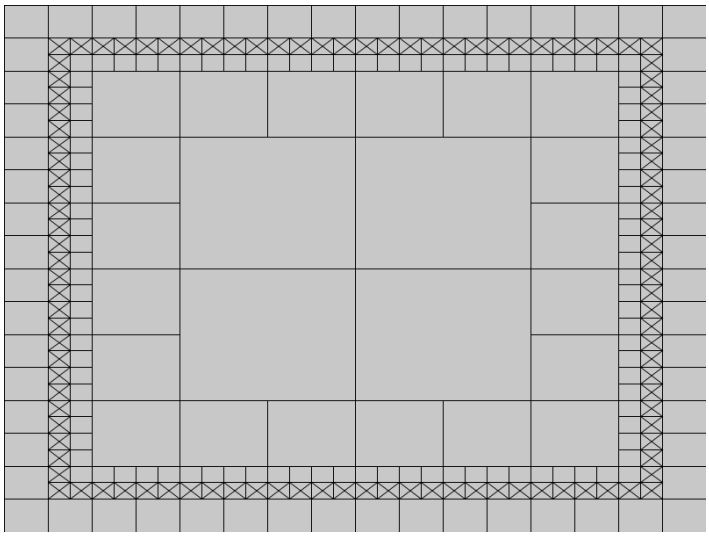


Figure 3.3: Quad-tree of one slice of 20mm box model

Apply these algorithm to our test case of hot-end model, the result is shown as Figure 3.4.

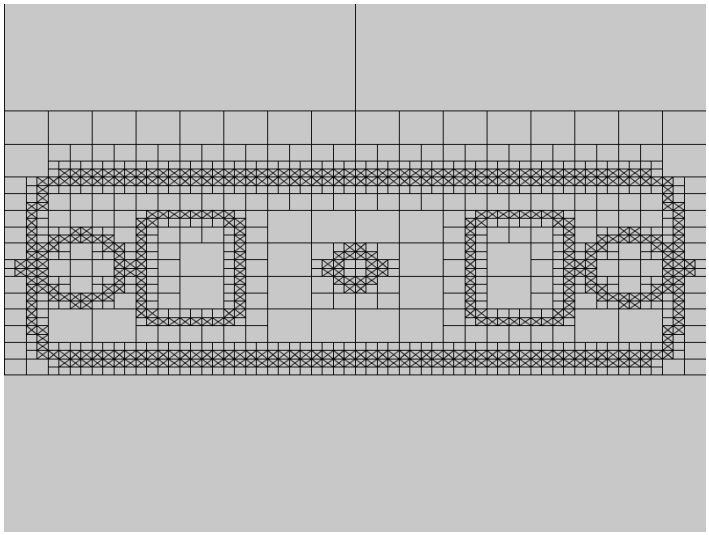


Figure 3.4: Quad-tree of one slice of hot-end model

Notice that the collided nodes are marked with crosses.

3.2.5 Flood

After our slices are converted to a quad tree in a BFS order array, they are ready for the flooding algorithm[5].

However, we can not simply flood from outside of the slice, because cases like the hot-end model (see Figure 3.1b) exists. If a slice is hollow, the flooding can not reach the inner spaces. Therefore, we need to flood from inside of the slice.

To find the initial flooding position, We use a method similar to ray-tracing: cast a series of parallel rays separated by the nozzle diameter into the slice. The actual algorithm uses vertical and horizontal rays. Figure only shows vertical rays for clearer picture.

For the 20mm box, ray intersections are shown as Figure 3.5.

[5]Source code is given in Appendix A.5.

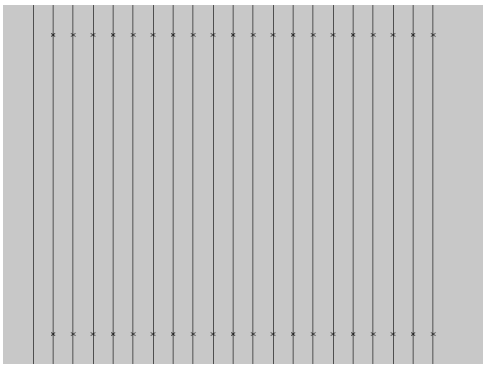


Figure 3.5: Rays intersecting with box

And for the hot-end, ray intersections are shown as Figure 3.6.

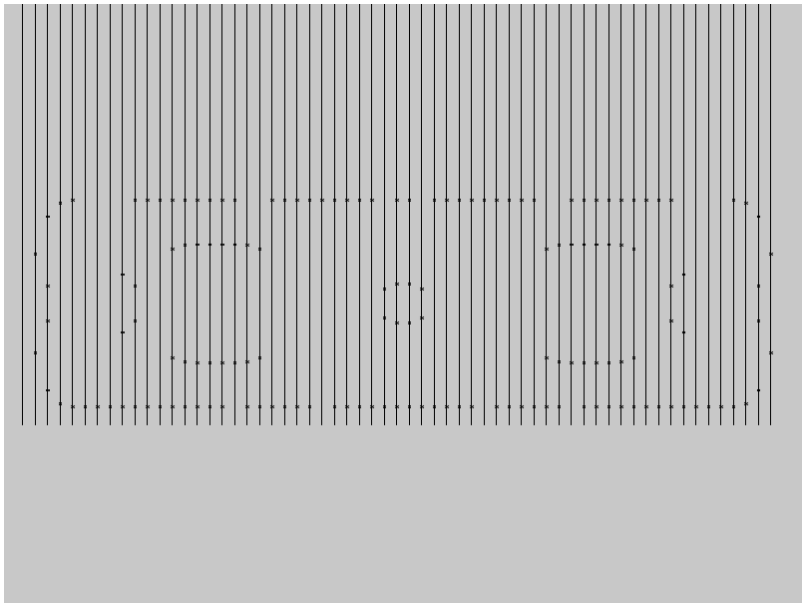


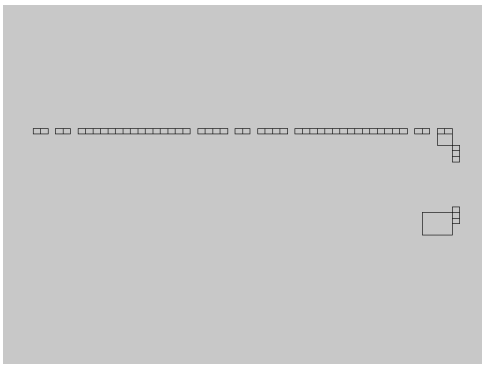
Figure 3.6: Rays intersecting with hot-end.

Using Figure 3.5 or 3.6 as examples, for each line of ray, the middle point of the first intersection and second intersection are being used as initial flooding position. Be aware that if more than two intersections exist, then the middle points of the third and fourth, fifth and sixth and so forth need to be included, in order to find the initial flooding positions for contained and disconnected shapes. Due to the extra complexity, contained and disconnected shapes are not fully supported in the current version. Finding the initial flooding points is done with the "find-contained-flooding-point" function.

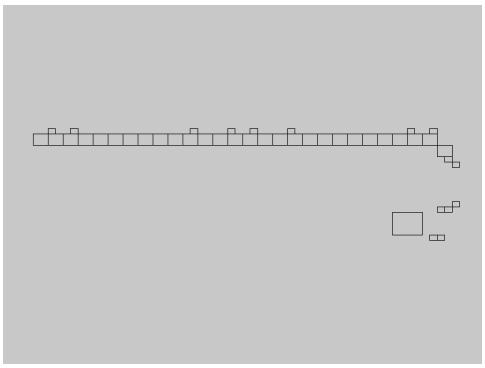
The flooding algorithm is implemented by the function “flood-node”. The steps of the algorithm are:

1. Use the “find-contained-flooding-point” function to find a set of initial flooding positions.
2. Find leaf nodes that contains initial flooding positions, marked as flooded.
3. Find the neighbor leaf nodes that are false (not collided), marked as flooded.
4. Repeat the above operation until no more neighbor leaf nodes are found. Return all of the flooded leaf nodes.

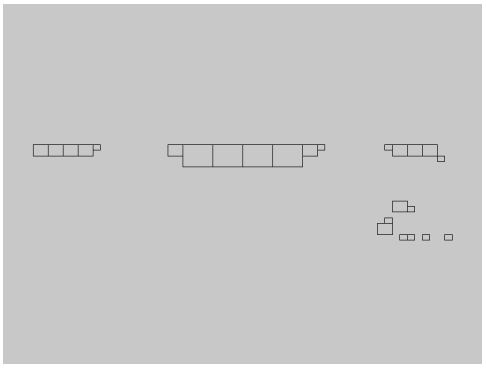
Let us run the flooding algorithm against one slice of the hot-end model as shown in Figure 3.8.



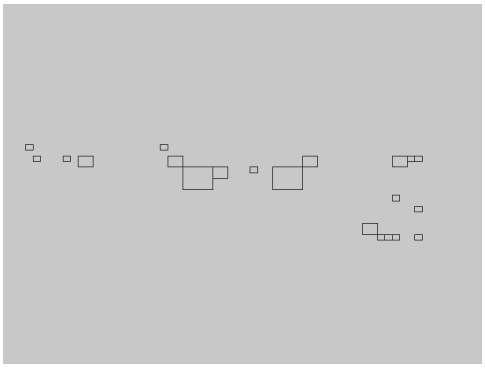
(a) Initial flooding nodes



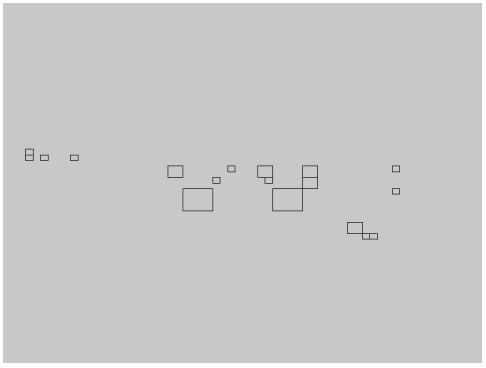
(b) Second iteration



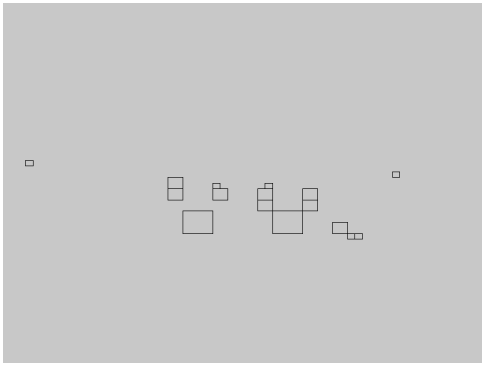
(c) Third iteration



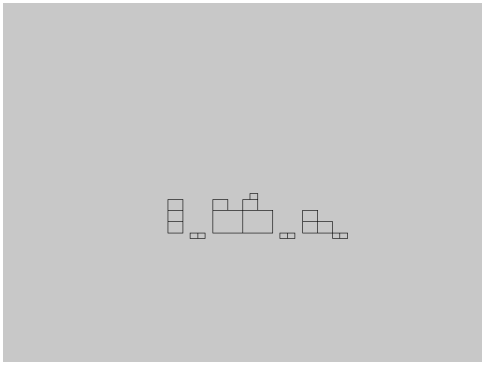
(d) Fourth iteration



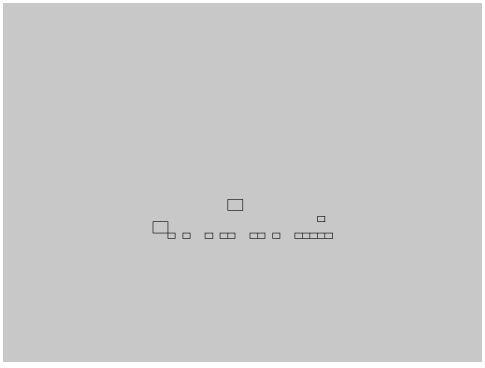
(e) Fifth iteration



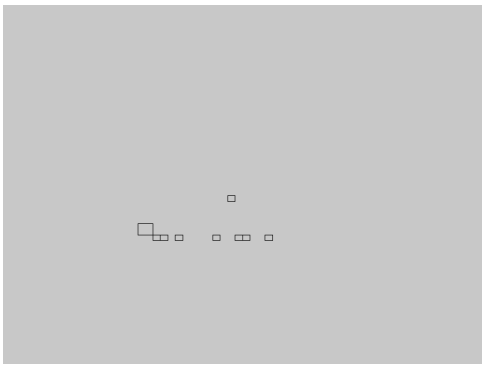
(f) Sixth iteration



(g) Seventh iteration



(h) Eighth iteration

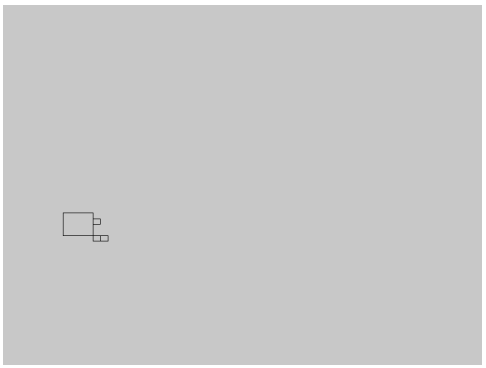


(a) Ninth iteration

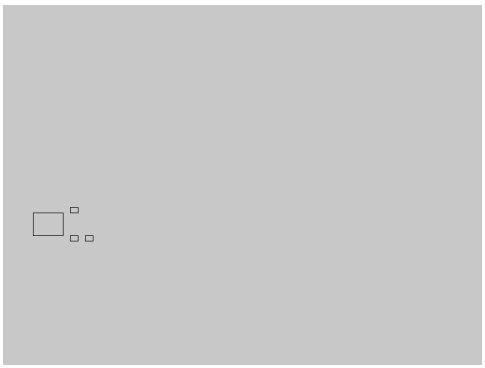
(b) Tenth iteration

(c) Eleventh iteration

(d) Twelfth iteration



(e) Thirteenth iteration



(f) Fourteenth iteration

(g) Fifteenth iteration

(h) Sixteenth iteration

Figure 3.8: Flooding algorithm iterations

And the final result is shown as Figure 3.9.

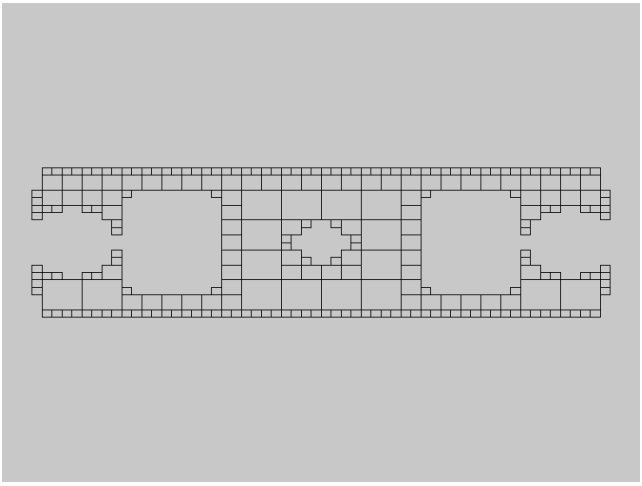


Figure 3.9: Flooded result of the hot-end

Combined with the collided leaf nodes, we have our final shape ready for infill as shown in Figure 3.10.

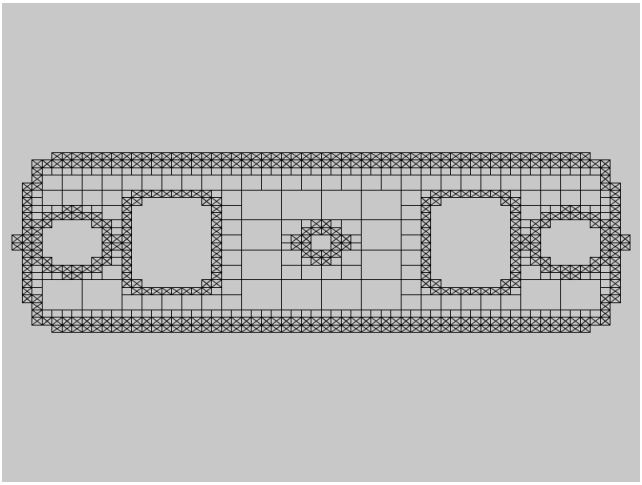


Figure 3.10: Flooded result of the hot-end with perimeter

3.2.6 Eulerian

The Eulerian namespace is the most important part of the program: it produces the Eulerian circuit[6]. After the flooding algorithm is applied, we have the shapes of the slices stored in quad-trees. Before they are ready to be searched for Eulerian

[6]Source code is given in Appendix A.6.

circuits, we need to convert the slices to be Eulerian graphs. For the purpose of this discussion, the following terminologies are used:

- Nodes – All of the leaf nodes of the quad-tree. Their geometrical positions are the center points of the bounding boxes of the leaf nodes.
- Edges – Geometrically adjacent leaf nodes have edges connecting them. An edge is a line that starts and ends with a pair of connected nodes from center to center. Non-adjacent leaf nodes do not have edges.

Figure 3.11 showing a visualization of the graph form of one slice from the hot-end model.

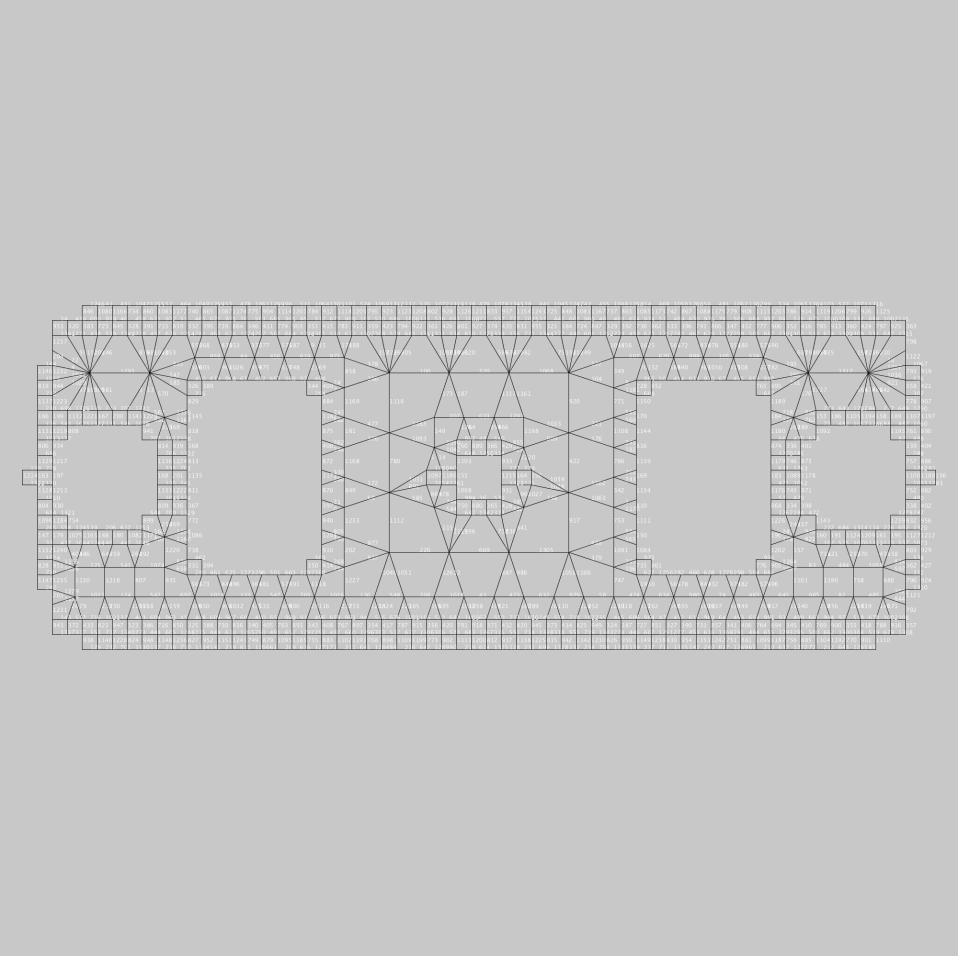


Figure 3.11: A slice of the hot-end in graph form

To convert a graph to Eulerian Graph, the target here is to change the degrees of the nodes with odd degrees. Remember we mentioned earlier that there are two ways to do so:

1. Remove the edges connecting a pair of odd degree nodes.

2. Connect a pair of non-neighboring odd degree nodes.

Let us consider which is appropriate here. The first method removes some of the edges. However, as we mentioned earlier, missing edges can be fixed by adding fillers after the printing is done. The second method adds edges. However, the additional edges will be connecting non-neighbor nodes. This is not appropriate for our application because such edge will cause the printer head to jump or the nozzle to cross printed edges. Thus the first method is better for our application. For extra long edges added by the second method, they can be rerouted into a purposely designed bridging area, so that removing then afterwards will be easier.

Therefore the algorithm we devised removes all of the edges that are connecting pairs of odd degree nodes in as many cases as possible, and then connects the remaining pairs of disconnected odd degree nodes. The functions doing these are "remove-odd-deg-nodes" and "connect-odd-deg-nodes" respectfully.

For the hot-end example, applying the remove-odd-deg-nodes function produces the set shown in Figure 3.12.

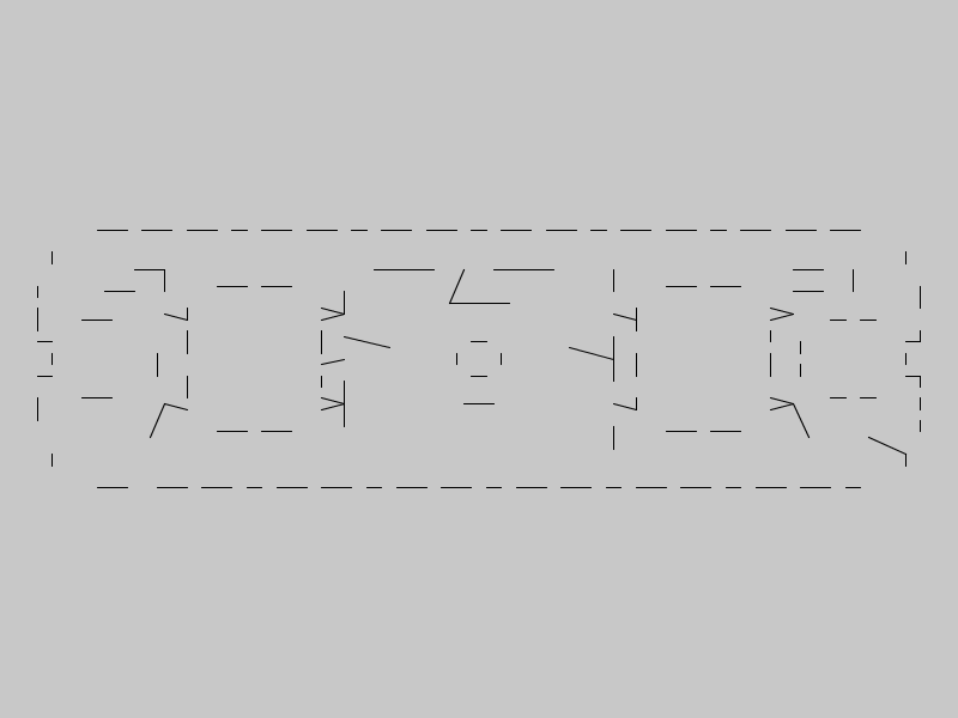


Figure 3.12: Removed edges

Applying the connect-odd-deg-nodes function to the hot-end example after the edges have been removed produces the set shown in Figure 3.13.

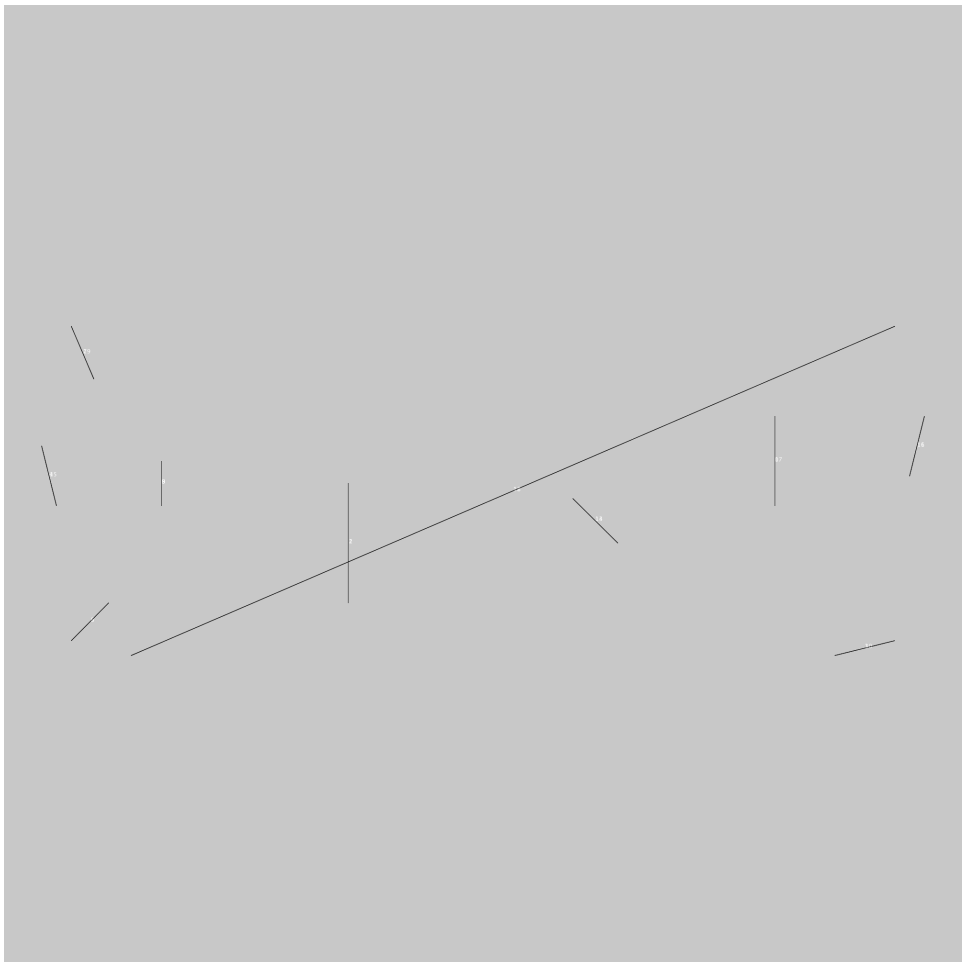


Figure 3.13: Connected edges

The function “convert-to-eulerian” applies these two sets into the graph and produces a Eulerian graph as Figure 3.14.

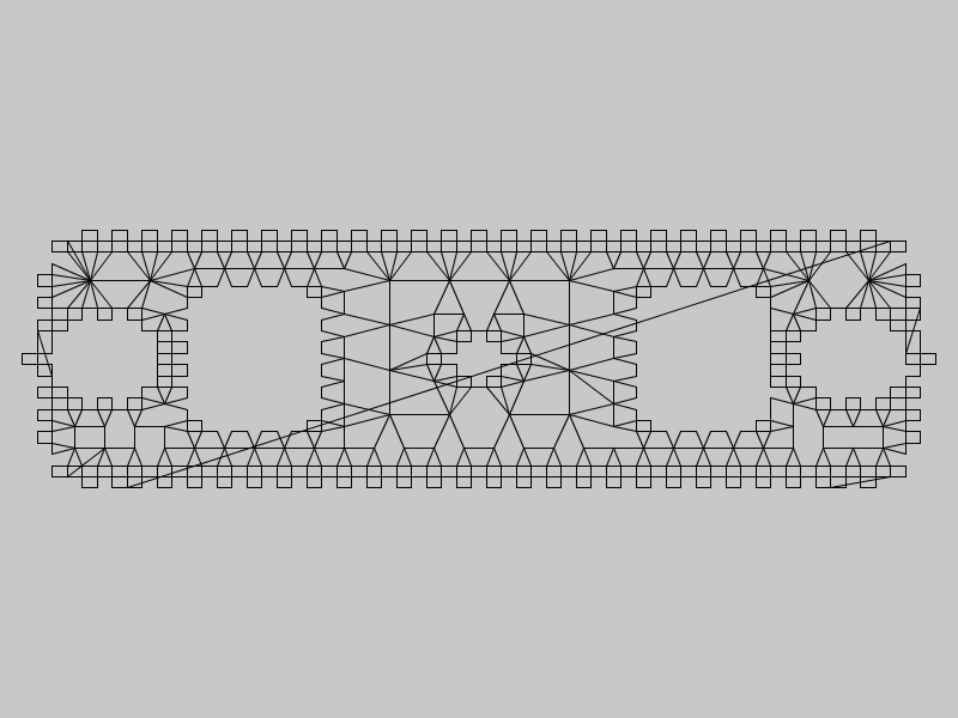


Figure 3.14: One slice of hot-end converted to a Eulerian graph

The next step is the implementation of the Hierholzer Algorithm. The basic idea of the Hierholzer algorithm is simple:

1. Randomly walk the nodes, mark each walked node, until your walk forms a loop – the "random-loop-walk" function does this.

2. Starting from any neighbor node of one of the walked nodes, repeat the random walk to form new loops.

3. When all nodes are marked "walked", combine all the walked loops into a single circuit – the "hierholzer" function does this.

Applying the Hierholzer algorithm to our hot-end base slice in Eulerian graph form, we obtained the graph shown in Figure 3.15. Although they are too small to see in the figure, the nodes are numbered according to their order in the Eulerian circuit.

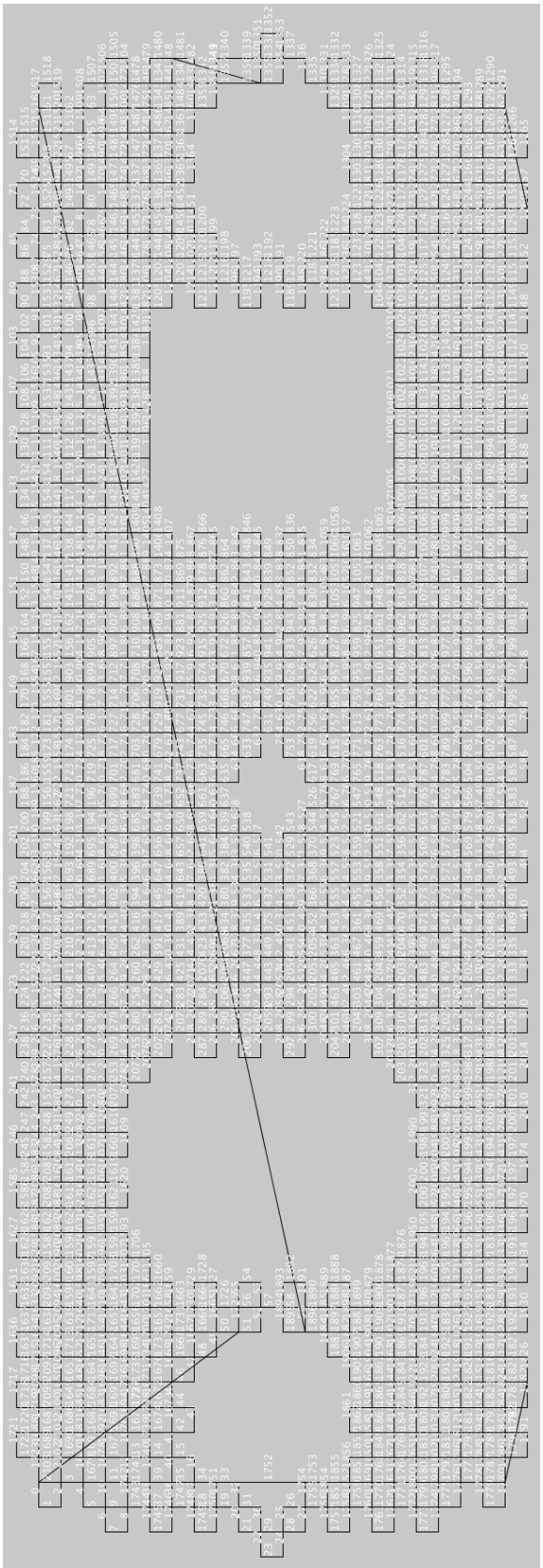


Figure 3.15: Eulerian circuit for the hot-end slice

After applying the same set of algorithms to the 20mm box, final result is shown as Figure 3.16.

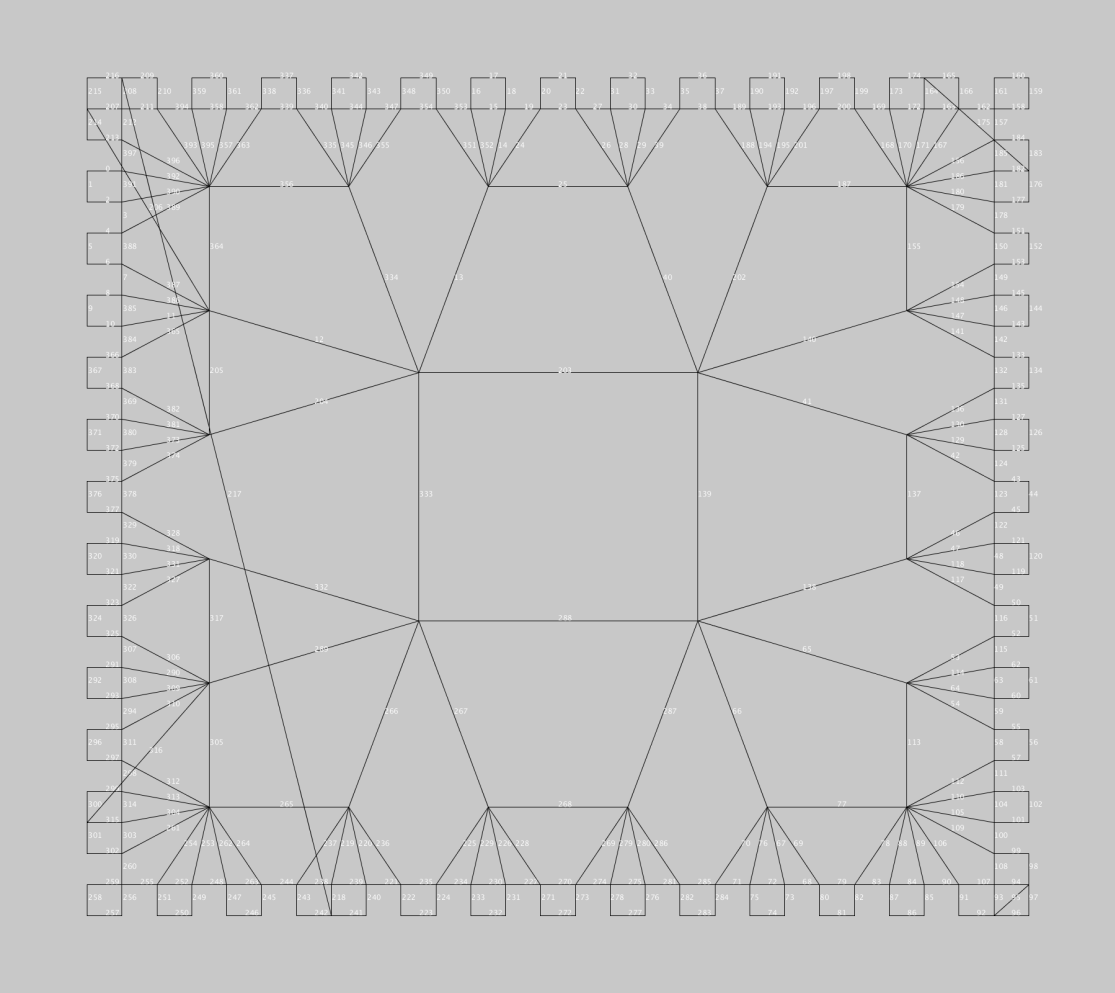


Figure 3.16: Eulerian circuit for the 20mm box slice

Now we are ready to generate the G-Code.

3.2.7 Gcode

The Gcode namespace generates the G-Code[7]. G-Code generation is simple. After we have a set of Eulerian circuits (one for each layer), our G-Code commands for a layer are just a set of mapping movement commands that describe a path from one node to the next in the same order as the Eulerian circuits are given.

[7]Source code is given in Appendix A.7

The function "slice-str" generates the G-Code of each circuit of the layers recursively:

1. Get the lowest positioned layer from a given list of layers.
2. With the "cut-point" representing the Z-axis position of this layer, translate it to a vertical movement G-Code command "G1 Z [cut-point]".
3. Calculate the distances between each node, stored in the list "point-distances".
4. With the given "current-e-height" representing the extruded length (initial to 0) from the previous layer, calculate the total extruded length for each node, stored in the list "extrusions". Replace "current-e-height" with the last value of the list.
5. Translate the node positions with their extruded lengths into a series of movement G-Code commands, each one as "G1 X [x-axis] Y [y-axis] E [extruded length]".
6. Combine the newly translated commands with the given "last-cmd" representing previously translated commands as the new "last-cmd".
7. Call this function recursively with the new "current-e-height", "last-cmd" and the remain list of layers. Until the list of layers is empty, then all the circuits are translated to G-Code commands.

The only tricky part is each of the extrusion axis G-Code command uses absolute distant to represent the extruded length of the plastic filament. As a result, each movement will increase the extruded length of the filament globally, to satisfy the requirement of the G-Code command.

Chapter 4

3D Printer Simulator

The best way to test the result of the Embodier slicer would be to execute the G-Code on an actual 3D printer. However, different 3D printers have different configurations. The melting point will be different depending on the plastic type, and there are different filament diameters, printer head nozzles sizes and others to be considered. To reduce the time duration of this project, we left out these factors and used a 3D printer simulator to validate the generated G-Code instead.

The simulator could served as a G-Code previewer for the Embodier slicer in the future.

4.1 Requirements of the 3D Printer Simulator

A G-Code program is simply a series of movement commands for all stepper motors of the 3D printer. These motors control the position, speed, and accuracy of the printing process. The motors are:

- X axis stepper motor – controls the left-and-right movement of the printer head.
- Y axis stepper motor – controls the forward-and-backward movement of the printer head.

- Z axis stepper motor – controls the height movement of the printer head. The height movement only needs to be accurately controlled for the up movement. Therefore some SLA printers do not achieve this with a stepper motor. For example, the Peachy printer simply uses liquid drops to raise the water-level, such that the height of the printing surface increases along with the water-level.

- E axis stepper motor – controls the extruding rate, pressure, pause of the printing material in the nozzle. SLA printers does not need this motor.

G-Code is basically an imperative style programming language that commands these motors. G1 is the movement command specifically. There are other commands that control the calibrations, speed, reset, etc. of the printer and we are leaving those out deliberately.

The simulator then does the following procedure:

1. Parses the G-Code and finds the movement commands.

2. Follows the movement commands and draws the printing path along the movements only if the E stepper motor is in the status of extruding printing materials.

3. The drawn paths are then output to WebGL Canvas of a supported browser.

The whole simulator will be compiled to pure html/javascript and requires no server or internet to execute.

4.2 Structure of the 3D Printer Simulator

The implementation of the simulator are separated into following namespaces:

- core – the entry point of the program, the joint of all other parts.

- fileapi – the parser of the G-Code that reads the X, Y, Z and E movements.

- canvasdraw – graphic engine that draws the parsed printing path to WebGL Canvas.
- webcomponents – the Web Interface widgets for file upload, view angles and layer controls.

4.3 Implementation

4.3.1 Core

The core is where all namespaces are connected together[1]. Two major libraries are being used as the framework here:

- Secretary – A client side routing library. You can have RESTFul style URL routing just within browser without making any request to a server.
- Reagent – A wrapper library for Facebook React. React uses functional reactive programmer approach to build Web UI components. Basic idea is that the UI components "react" to program status changes, saves huge amount of programmer efforts to setup event hooks and callbacks.

Three routes are setup. They are the upload route, the layers route and the root route where it is just an alias for the upload route. Upload route is where all of the file upload web components are active, and the layer route is where all of the WebGL and layer control web components made active.

When any of the routes being called by the browser from URL requesting, an atom that is being monitored by Reagent is then updated(by the "reset!" function), and the related web components are activated.

[1]Source code is given in Appendix B.1

4.3.2 Fileapi

G-Code files are text files. Thus the G-Code parsing of Fileapi is strictly string analyzing[2]. The main function of this namespace is "readFile". The threading macro in this function connected all other functions of the Fileapi namespace:
(reset! layers (-> raw-str s/split-lines filterG1 layered cmd-map collapseZ collapseXYE add-next))

1. "raw-str" is the direct string from file.
2. "split-line" makes a vector of lines.
3. "filterG1" removes any non-movement commands.
4. "layered" partitions the lines by Z movements.
5. "cmd-map" translate each movement command into a map of axis and values like {:x 1, :y 2, ...}.
6. "collapseZ" duplicates the Z axis values to all movement commands within the its layer.
7. "collapseXYE" duplicates the X, Y, E axis values to their following commands that is missing those values.
8. above two collapsing functions effectively convert the imperative information (where global status are everywhere) into stateless declarative information.
9. "add-next" function adds the "next" key value pair to each command base on extrusions. If there existed an extrusion movement, then {:next (next position)} are added into the command, else a nil value is added. This information helps

[2]Source code is given in Appendix B.2

the draw engine to draw the extruding lines from "this movement position" to "next movement position".

The final result is a series of stateless, self-explanatory and independent movement commands. Future optimization will be greatly benefit from this kind of data, since they can be easily parallelized.

Then "setOnLoad" is the event trigger point function that hooks the string parser whenever a file is loaded into the browser memory.

4.3.3 Webcomponents

The webcomponents namespace are a collection of HTML UI widgets[3]. They are either changing or react to the atoms that they are bound to. These atoms are the only status that we need to keep track of:

1. routes – determines which UI widgets to be turn on or off.
2. layers – stores the parsed movement commands once a G-Code file is loaded.
3. current-layer-num – Layer view controller changes this atom so that a detailed layer is drawn as lines while others layers are drawn as points.
4. req-id – Stores the ID so that the draw engine can cancel an animation frame request previously scheduled through a call to window.requestAnimationFrame(). This increases the performance after multiple G-Code sessions.

4.3.4 Canvasdraw

Canvasdraw is the CG engine that draws the 3D G-Code path into a WebGL canvas component[4]. This is the same as most of the game loops in game engines. The animation anonymous function in "show-layer" is the constantly running loop.

[3]Source code is given in Appendix B.3
[4]Source code is given in Appendix B.4

Notice that the animation loop is running outside of the Reagent framework. That means the animation will not react to atom changes automatically. So we have to dereferences the atom in the "update-scene" function and this function is triggered by an "on-mouse-out" event.

Since the animation loop itself is not controlled by Reagent, as it is getting more complicated, variable states will eventually out of control. Re-factor this loop to follow an Entity component system style is most likely happening in the future development. In the end, the whole stack would be functional and data centric, and free of objects and states.

4.4 Example Using Slic3r

Slic3r is a popular open-sourced slicer. Let us uses Slic3r to test if the simulator shows the correct printing path. The model that we used is a 20mm box from thingiverse as shown in Figure 4.1.

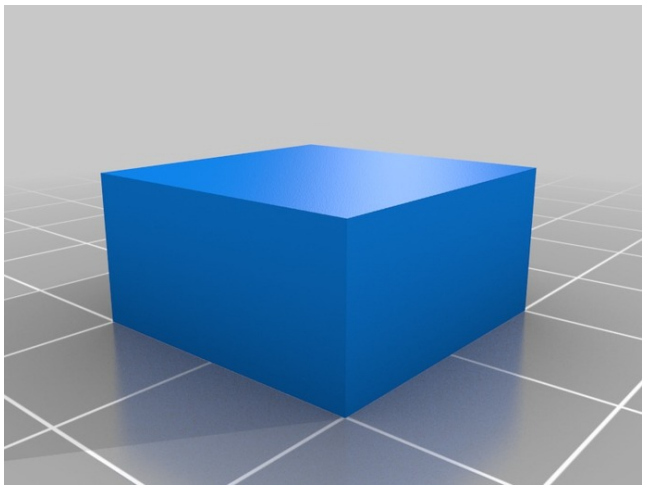


Figure 4.1: 20mm box from thingiverse

And Figure 4.2 shows what is drawn onto the canvas when the current layer is set to 3:

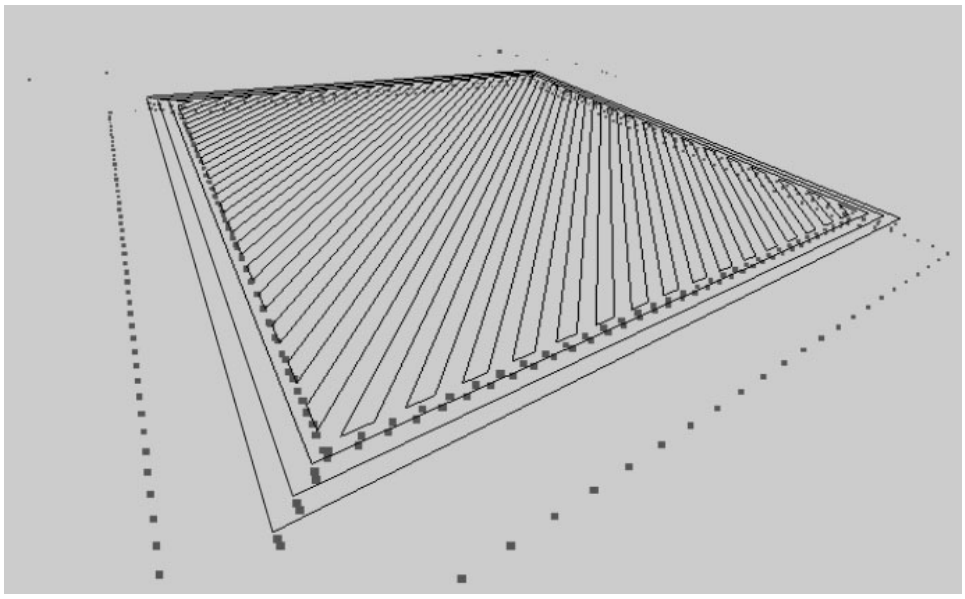


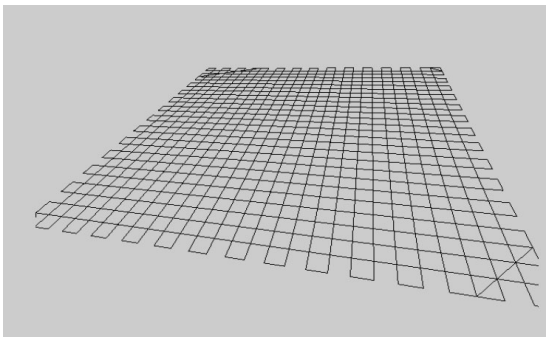
Figure 4.2: G-Code path of 3rd layer

The simulator only draws the path of the current layer. The node positions of the underlying layers below the set current layer are only drawn as vertices, for the sake of better visibility. As shown in Figure 4.2, we can see that the current layer requires at least three separated paths to be drawn: two for the perimeter, one or more for the infill. This appears to be showing the printing path of G-Code generated by Slic3r correctly.

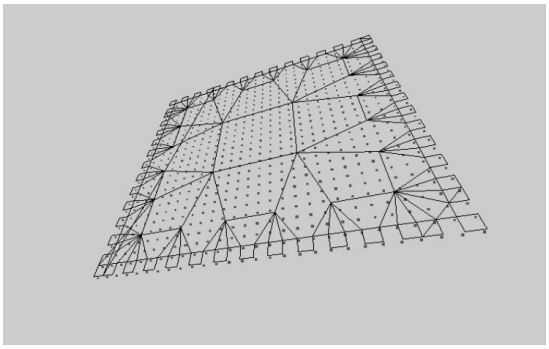
4.5 Validation of Embodier Slicer

With the G-Code simulator, we can now preview the end result of the G-Code generated by the Embodier slicer.

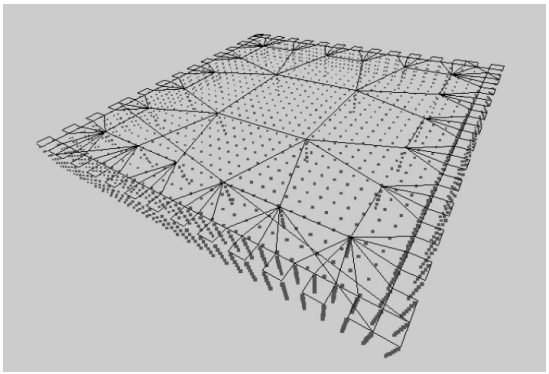
Let us try the 20mm box model first: the complete slicing process took 6 minutes on a 2012 Mac-book Air with 1.7 Ghz i5 CPU. The result is shown as Figure 4.3.



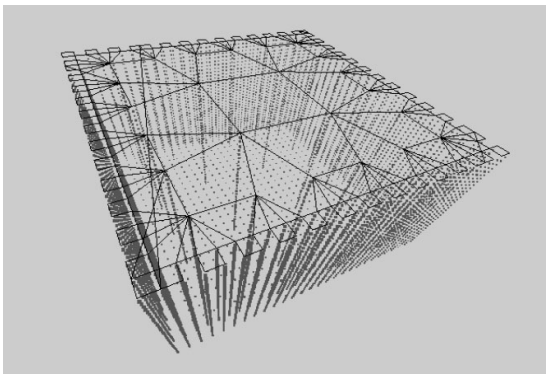
(a) First layer, the bottom layer



(b) Second layer



(c) Tenth layer



(d) Thirty-third layer, the last layer

Figure 4.3: G-Code path for the 20mm box.

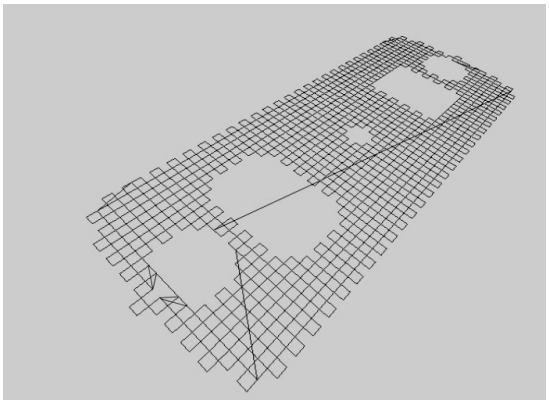
For the first layer in Figure 4.3a, the generated path is a single and continuous one. Being the bottom layer, it forms the surface of one side of the box. The infill pattern is dense and traverses all parts, leaving no gap as desire.

For the second and the tenth layer (Figure 4.3b, 4.3c), we can see that the properties of quad tree leads the generated path of the center contain area to have low density pattern. And then the path along the perimeter became dense again, such that the surface of the box will not have gaps. Not only the quad tree data structure gives us the efficient in speed and memory, but it also gives us a desire structure that saves plastic consumption and maintains continuous surfaces.

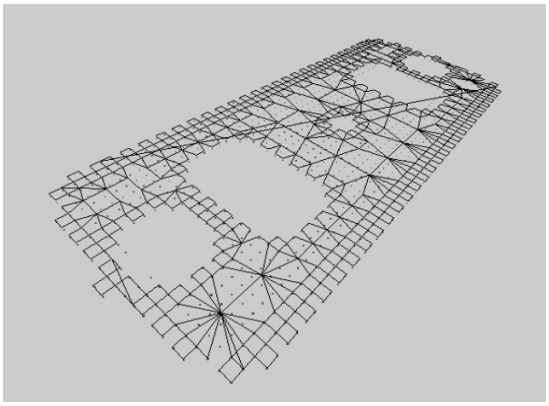
Lastly, for the last layer as shown in Figure 4.3d, we found a problem instantly – as the last layer forms the top surface of the box, the generated path pattern needs to have high density such that it traverses all parts. However, the current generated path is very similar to the paths of the middle layers like the second and the tenth. Obviously, this path will leave huge open spaces in the surface of the box.

The cause of the problem is a bug in the slicing planes generation such that the last plane generated is slightly lower than the top surface. This minor error remains to be fixed by the newer version of the Embodier slicer.

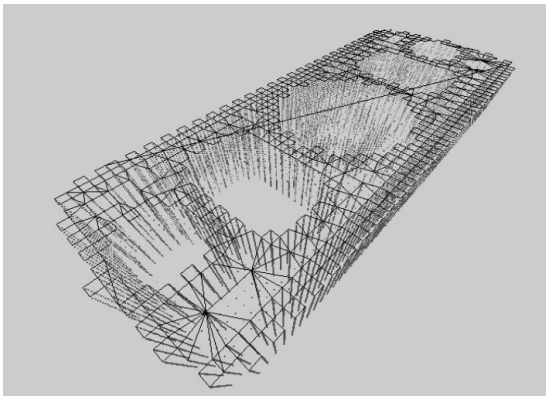
And let us try to preview the G-Code for the hot-end model: the complete slicing process took 30 minutes on a 2012 Mac-book Air with 1.7 Ghz i5 CPU. The result is shown as Figure 4.4.



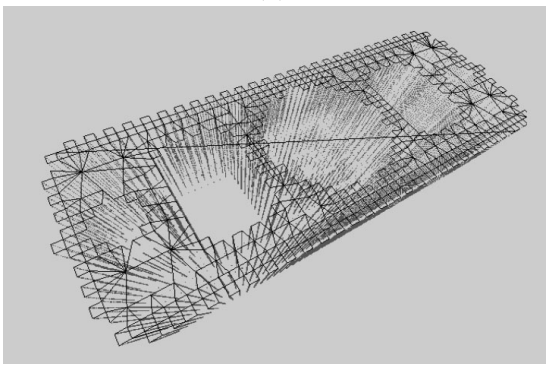
(a) First layer, the bottom layer



(b) Second layer



(c) Tenth layer



(d) Thirty-fifth layer, the last layer

Figure 4.4: G-Code path for the hot-end model.

For the first layer in Figure 4.4a, the generated path is a single and continuous one as expected. However, unlike the 20 mm box, the hot-end model has hollow spaces. The conversion of layer into Eulerian graph leads the generated path to be running across the hollow spaces, and this will required further processing after the object is printed. In the future version, a better algorithm that reroutes the crossing segment of the path could produce better results.

For the preview of the second and tenth layer (Figure 4.4b, 4.4c), a continuous single extrusion path, the loose pattern for the center contain area and the dense pattern for the path along the perimeter are observed in the graph as expected. However, the generated path is also running across the hollow spaces, and leads to further processing of the object after printed.

Lastly, for the last layer as shown in Figure 4.4d, besides the crossing path on the hollow spaces, we also found the problem caused by the Embodier slicer that it did not correctly generates the last slicing plane – it leads to the open spaces that should have been closed by a high density path pattern that traverses all parts. The next version from the version of this report would have this minor error fixed.

Chapter 5

Conclusion

The Embodier slicer still has lots of improvements to be added. For the Eulerian graph conversion, we could improve it by generating shorter jumps, rerouting longer jumps to avoid hollow spaces etc. Also, instead of using quad trees, the parity of the trees could be increased from 4 to 32 or 64 (depending on the memory address widths supported by the operating system and the hardware). With the increased parity, the depth of the trees will decrease dramatically, such that it will yield significant faster access times. (from $O(\log_4 N)$ to $O(\log_{32} N)$). Lastly, each partitioned space could be filled with more detailed infill patterns.

The Embodier slicer provides following major improvements using the single extrusion approach for 3D print model slicers:

1. Reliability - eliminate sudden change of gear movements.

2. Speed - less traversal distance for the printer head.

The Embodier slicer proves that Space Partitioning, Flooding Algorithm, Eulerian Graph and Hierholzer Algorithm can be jointly applied to 3D printer slicing and infilling procedures.

Although there are still more works to make the Embodier slicer to generate usable G-Code for a specific model of 3D printers, the slicer successfully generates continuous

G-Code paths in an efficient manner, without any hardware modification required. The Embodier slicer provides a basis for further development of a printing slicer, for 3D printers that generate continuous extrusion paths.

Appendices

Appendix A

Embodier Slicer

A.1 Source Code of "slicer.core"

```
(ns slicer.core
  (:require [clojure.tools.cli :refer [parse-opts]]
            [slicer.file :as f]
            [slicer.slice :as s]
            [slicer.gcode :as g])
  (:gen-class))


;; ##cli specs
;; declare the specs for the cmd args
(def cli-options
  [["-h" "--help"]
   ["-s" "--stl STL-file" "embodier -s [stl-file-name]." :id
     ↪ :stl :validate [#(re-find #".+\.stl" %) "Must be a
     ↪ binary STL file."]]
```

```
    ["-g" "--gcode Gcode-file" "embodier -g [gcode-file-name].
      ↪ " :id :gcode :default "out.gcode"]
    ]
  )

(def help-txt "To slice STL, simply run: \"embodier -s [
  ↪ stl-file-name] -g [gcode-file-name]\" ")

;; ## the exit
(defn exit [status & msg]
  (when msg (println msg))
  (System/exit status))

;; ## Main function, entry point of the command
;; first destructed the map from tools.cli, making it more
  ↪ readable.
;; If there are no errors and it's not a help request,
  ↪ process with file IO
(defn -main [& args]
  (let [{opts :options args :arguments summary :summary errs
    ↪ :errors}
        (parse-opts args cli-options)]
    (when (not (empty? errs))
      (doseq [err errs]
        (println err))
      (exit 1))
    (when (:help opts)
```

```
    (println help-txt)
    (exit 0 summary))
  ;(when (:stl opts)
    ;(println (prn-str (f/parse-stl (:stl opts)))))
  (when (and (:gcode opts) (:stl opts))
    (g/write-gcode (:gcode opts)
                   (-> (s/slice (:triangles (f/parse-stl (:
                     ↪ stl opts)))
                               (s/gen-planes (:min (s/
                                 ↪ find-min-max :z (:
                                 ↪ triangles (f/
                                 ↪ parse-stl (:stl opts)
                                 ↪ )))) (:max (s/
                                 ↪ find-min-max :z (:
                                 ↪ triangles (f/
                                 ↪ parse-stl (:stl opts)
                                 ↪ )))) 0.3 :z)
                               :z)
                       s/rm-nil
                       s/tri-compressor
                       g/gcode)))
  ))
```

A.2 Source Code of "slicer.file"

```
(ns slicer.file
  (:use gloss.core
```

```
        gloss.io
        clojure.java.io))

(def delimiters ["\r" "\r\n" "\n" \newline])

;; ## binary triangle frame

(defcodec b-triangle
  (ordered-map
   :normal [:float32-le :float32-le :float32-le]
   :vertex-1 [:float32-le :float32-le :float32-le]
   :vertex-2 [:float32-le :float32-le :float32-le]
   :vertex-3 [:float32-le :float32-le :float32-le]
   :attribute :uint16))

;; ## ascii triangle frame

(defcodec a-triangle
  (ordered-map
   :_ (string :utf-8 :delimiters ["normal "])
   :normal [(string-float :utf-8 :delimiters [\space])
            (string-float :utf-8 :delimiters [\space])
            (string-float :utf-8 :delimiters delimiters)]
   :_ (string :utf-8 :delimiters ["vertex "])
   :vertex-1 [(string-float :utf-8 :delimiters [\space])
              (string-float :utf-8 :delimiters [\space])
              (string-float :utf-8 :delimiters delimiters)]
```

```
   :_ (string :utf-8 :delimiters ["vertex "])
   :vertex-2 [(string-float :utf-8 :delimiters [\space])
              (string-float :utf-8 :delimiters [\space])
              (string-float :utf-8 :delimiters delimiters)]
   :_ (string :utf-8 :delimiters ["vertex "])
   :vertex-3 [(string-float :utf-8 :delimiters [\space])
              (string-float :utf-8 :delimiters [\space])
              (string-float :utf-8 :delimiters delimiters)]
   :_ (string :utf-8 :delimiters ["endfacet"])
   :_ (string :utf-8 :delimiters delimiters)
   )
  )

;; ## binary stl frame

;; [header triangles]
;; another way to interpret header is (vec (repeat 80 :byte))
(defcodec b-stl
  (ordered-map
   :header (string :utf-8 :length 80)
   :triangles (repeated b-triangle :prefix :uint32-le)))



;; ## Ascii stl frame decode

;; The number of appearance of "normal" will be exactly how
   ↪ many triangles
```

```
;; Gloss documentation gives the following method:
;;:triangles (repeated a-triangle :delmiters ["endsolid"]) .
;; But this does not work.
;; Thus I have to count the words manually
(defn adecode [buffer]
  (let [n (count (re-seq #"normal" (String. buffer)))
        a-stl (compile-frame
               (ordered-map
                :header (string :utf-8 :delimiters delimiters
                  ↪ )
                :triangles (vec (repeat n a-triangle))
                :_ (string :utf-8 :delimiters delimiters)))]
    (decode a-stl buffer false)))


;; ## parse file

;; 115 111 108 105 100 are the magic numbers for "solid"
;; At first it keeps giving incifient bytes errors.
;; Then I tested the codec and finds out that it's the
  ↪ endianess that is messing with me.
(defn parse-stl [stl-file]
  (let [length (.length (file stl-file))
        buffer (byte-array length)]
    (.read (input-stream stl-file) buffer)
    (if (= '(115 111 108 105 100) (first (split-at 5 buffer))
      ↪ )
```

```
      (adecode buffer)
      (decode b-stl buffer))))
```

A.3 Source code of "slicer.slice"

```
(ns slicer.slice)


(defn dot*
  "
␣␣Dot␣Product
␣␣This␣is␣a␣simple␣version␣of␣dot␣product␣just␣for␣3-tuples"
  [[x1 y1 z1 :as p1]
   [x2 y2 z2 :as p2]]
  {:pre [(float? x1) (float? y1) (float? z1)
         (float? x2) (float? y2) (float? z2)]}
  (+
   (* x1 x2)
   (* y1 y2)
   (* z1 z2)))


(defn norm
  "
␣␣Norm␣of␣a␣vector
␣␣"
  [[x y z :as v]]
  (Math/sqrt (dot* v v)))
```

```
(defn point-plane
  "
  Distance from point to plane
  Returns the distance.
  If result is positive, point is at the side of the normal,
  if the result is negative, point is at the other side of
    ↪ the normal.
  "
  [[x0 y0 z0 :as point]
   [[x2 y2 z2 :as normal] [x1 y1 z1 :as position] :as plane]]
  {:pre [(float? x1) (float? y1) (float? z1)
         (float? x2) (float? y2) (float? z2)
         (float? x0) (float? y0) (float? z0)]}
  (/
    (dot* normal (map - point position))
    (norm normal)))


(defn plane-line-inc
  "
  Ray/Segment and Plane intersection
  line is represented by start-point and end-point
  plane is represented by normal direction vector (from
    ↪ original) and the position point
  returns the intersection point if there is one,
  returns nil when not having any,
  returns the line when the line is on the plane.
```

```
␣␣"
  [[[x1 y1 z1 :as start-point] [x2 y2 z2 :as end-point] :as
    ↪ line]
   [[x3 y3 z3 :as normal] [x4 y4 z4 :as position] :as plane]]
  {:pre [(float? x1) (float? y1) (float? z1)
         (float? x2) (float? y2) (float? z2)
         (float? x3) (float? y3) (float? z3)
         (float? x4) (float? y4) (float? z4)]}
  (let [d (dot* normal (map - end-point start-point))]
    (cond (and (zero? (point-plane start-point plane)) (zero?
      ↪ (point-plane end-point plane))) line
          (zero? d) nil
          :else (let [r (/ (dot* normal (map - position
            ↪ start-point)) d)]
                  (if (and (<= r 1) (>= r 0))
                    (vec (map + start-point (map * [r r r] (
                      ↪ map - end-point start-point))))
                    nil)))))



(defn triangle-plane-inc
  "
␣␣Triangle␣and␣Plane␣intersection
␣␣If␣the␣entire␣triangle␣sits␣on␣the␣plane,␣returns␣the␣whole
  ↪ ␣triangle;
```

```
  If one segment of the triangle sits on the plane, that's
    ↪ what you got;
  If a point of the triangle sits there, you have it;
  If not intersecting, nil;
  Otherwise, you have the intersection
  "
   [[[x1 y1 z1 :as p1]
     [x2 y2 z2 :as p2]
     [x3 y3 z3 :as p3] :as triangle]
    [[x4 y4 z4 :as normal]
     [x5 y5 z5 :as position] :as plane]]
   {:pre [(float? x1) (float? y1) (float? z1)
          (float? x2) (float? y2) (float? z2)
          (float? x3) (float? y3) (float? z3)
          (float? x5) (float? y5) (float? z5)
          (float? x4) (float? y4) (float? z4)]}
   (let [ds (map #(point-plane % plane) triangle)]
     (cond
       (= 3 (count (filter zero? ds)))
       triangle
       (= 2 (count (filter zero? ds)))
       (vec (filter #(zero? (point-plane % plane)) triangle))
       (= 1 (count (filter zero? ds)))
       (if (neg? (apply * (filter (complement zero?) ds)))
         [(first (filter #(zero? (point-plane % plane))
           ↪ triangle))
          (plane-line-inc
```

```
            (vec (filter #(not (zero? (point-plane % plane)))
              ↪ triangle))
            plane)]
          (vec (first (filter #(zero? (point-plane % plane))
            ↪ triangle))))
        :else
        (if (or (empty? (filter neg? ds)) (empty? (filter pos?
          ↪ ds)))
          nil
          (vec (filter (complement nil?)
                 [(plane-line-inc [p1 p2] plane)
                  (plane-line-inc [p2 p3] plane)
                  (plane-line-inc [p3 p1] plane)]))
          ))))

(defn slicing-plane
  "
  Given a x/y/z axis value and the axis keyword :x/:y/:z,
    ↪ returns the plane [[x y z :as normal] [x y z :as plane
    ↪ ]]
  "
  [a b]
  {:pre [(float? a)
         (keyword? b)]}
  (cond
   (= b :x) [[1.0 0.0 0.0] [a 0.0 0.0]]
   (= b :y) [[0.0 1.0 0.0] [0.0 a 0.0]]
```

```
   (= b :z) [[0.0 0.0 1.0] [0.0 0.0 a]]))


(defn gen-planes
  "
␣␣Generate␣a␣series␣of␣slicing-planes␣from␣'start'␣to␣'end'␣
    ↪ each␣'step'␣along␣the␣provided␣'axis'
␣␣"
  [start end step axis]
  {:pre [(number? start)
         (number? end)
         (number? step)
         (keyword? axis)]}
  (vec
   (for [i (range (bigdec start) (bigdec (+ end step)) (
     ↪ bigdec step))]
     (slicing-plane (double i) axis))))


(defn find-min-max
  "finds␣the␣highest␣and␣lowest␣point␣in␣axis␣of␣a␣collection
    ↪ ␣of␣triangles␣along␣provided␣axis"
  [axis
   [{[x1 y1 z1 :as p1] :vertex-1 [x2 y2 z2 :as p2] :vertex-2
     ↪ [x3 y3 z3 :as p3] :vertex-3} & tris :as triangles]]
  {:pre [(number? x1) (number? x2) (number? x3) (number? y1)
    ↪ (number? y2) (number? y3) (number? z1) (number? z2) (
    ↪ number? z3) (keyword? axis)]}
  {:min
```

```
(reduce (fn [min-num tri]
          (cond
            (= axis :x)
            (min
              min-num
              (min (first (:vertex-1 tri))
                   (first (:vertex-2 tri))
                   (first (:vertex-3 tri))))
            (= axis :y)
            (min
              min-num
              (min (second (:vertex-1 tri))
                   (second (:vertex-2 tri))
                   (second (:vertex-3 tri))))
            (= axis :z)
            (min
              min-num
              (min (last (:vertex-1 tri))
                   (last (:vertex-2 tri))
                   (last (:vertex-3 tri))))))
        (cond (= axis :x) (min x1 x2 x3)
              (= axis :y) (min y1 y2 y3)
              (= axis :z) (min z1 z2 z3))
        tris)
:max
(reduce (fn [max-num tri]
          (cond
```

```
                (= axis :x)
                (max
                  max-num
                  (max (first (:vertex-1 tri))
                       (first (:vertex-2 tri))
                       (first (:vertex-3 tri))))
                (= axis :y)
                (max
                  max-num
                  (max (second (:vertex-1 tri))
                       (second (:vertex-2 tri))
                       (second (:vertex-3 tri))))
                (= axis :z)
                (max
                  max-num
                  (max (last (:vertex-1 tri))
                       (last (:vertex-2 tri))
                       (last (:vertex-3 tri))))))
            (cond (= axis :x) (max x1 x2 x3)
                  (= axis :y) (max y1 y2 y3)
                  (= axis :z) (max z1 z2 z3))
            tris)}
  )

(defn triangle-map2vector
  "convert triangle map to vector"
  [{v3 :vertex-3, v2 :vertex-2, v1 :vertex-1, :as triangle}]
```

```
  [v1 v2 v3])

(defn slice
  "slice every triangle of the model with every plane along
    ↪ the axis"
  [triangles planes axis]
  {:pre [(map? (first triangles))
         (map? (last triangles))
         (vector? triangles)
         (vector? planes)
         (vector? (first planes))
         (keyword? axis)]
   :post [(seq? %)
          (map? (first %))]}
  (for [triangle triangles
        plane planes]
    (let [result (triangle-plane-inc (triangle-map2vector
      ↪ triangle) plane)]
      {;:axis axis
       :cut-point (cond (= axis :x) (first (second plane))
                        (= axis :y) (second (second plane))
                        (= axis :z) (last (second plane)))
       ;:plane plane
       ;:triangle triangle
       :result result})))

(defn rm-nil
```

```
  "remove nil results"
  [results]
  {:pre [(seq? results)
         (map? (first results))]
   :post [(seq? %)
          (map? (first %))]}
  (filter
    (fn [result]
      ((complement nil?) (:result result))) results))

(defn tri-compressor
  "transpose results to collections of triangles based on
    ↪ their cut-points"
  [[{;axis :axis
     cut-point :cut-point
     ;plane :plane
     ;triangle :triangle
     result :result} & more :as results]]
  (let [cuts (-> (group-by :cut-point (vec results)) sort)]
    (for [cut cuts]
      {:cut-point (first cut)
       :result (for [line (second cut)]
                 (:result line))}))))
```

A.4 Source code of "slicer.tree"

```
(ns slicer.tree
```

```
  (:require [clojure.core.match :refer [match]])
  (:use slicer.util))

(def tree-arity 4);changing this will affect the performance
  ↪ of this algorithm. aware that some functions (
  ↪ split-aabb ...) are not arity changable

(defn line-box-inc
  "check if a line start and end by two points is intersected
    ↪ with an AABB box. Imprative since performance is
    ↪ important"
  ([[x1 y1 :as line-start] [x2 y2 :as line-end]
    [min-x min-y max-x max-y :as aabb]]
   (line-box-inc [x1 y1] [x2 y2] [min-x min-y] [max-x max-y])
    )
  ([[x1 y1 :as line-start] [x2 y2 :as line-end]
   [min-x min-y :as box-min] [max-x max-y :as box-max]]
    (let [m (atom 0.0)
          x (atom 0.0)
          y (atom 0.0)]
      (cond
       (or
         (and (< x1 min-x) (< x2 min-x))
         (and (< y1 min-y) (< y2 min-y))
         (and (> x1 max-x) (> x2 max-x))
         (and (> y1 max-y) (> y2 max-y))) false
       (= x2 x1) true
```

```
        (do
          (reset! m (/ (- y2 y1) (- x2 x1)))
          (reset! y (+ (* @m (- min-x x1)) y1))
          (and (s>= @y min-y 0.000001) (s<= @y max-y 0.000001)
            ↪ )) true
        (do
          (reset! y (+ (* @m (- max-x x1)) y1))
          (and (s>= @y min-y 0.000001) (s<= @y max-y 0.000001)
            ↪ )) true
        (do
          (reset! x (+ (/ (- min-y y1) @m) x1))
          (and (s>= @x min-x 0.000001) (s<= @x max-x 0.000001)
            ↪ )) true
        (do
          (reset! x (+ (/ (- max-y y1) @m) x1))
          (and (s>= @x min-x 0.000001) (s<= @x max-x 0.000001)
            ↪ )) true
        :else false
        ))))


(defn tri-box-inc
  "check if a triangle intersects with an AABB box."
  ([[x1 y1 :as tri-1] [x2 y2 :as tri-2] [x3 y3 :as tri-3]
    [min-x min-y max-x max-y :as aabb]]
   (tri-box-inc [x1 y1] [x2 y2] [x3 y3] [min-x min-y] [max-x
     ↪ max-y])
   )
```

```
([[x1 y1 :as tri-1] [x2 y2 :as tri-2] [x3 y3 :as tri-3]
 [min-x min-y :as box-min] [max-x max-y :as box-max]]
  (cond
   (or
     (and (< x1 min-x) (< x2 min-x) (< x3 min-x))
     (and (< y1 min-y) (< y2 min-y) (< y3 min-y))
     (and (> x1 max-x) (> x2 max-x) (> x3 max-x))
     (and (> y1 max-y) (> y2 max-y) (> y3 max-y))) false ;
       ↪ tri completely outside
   (and
    (and (< x1 min-x) (> x1 max-x) (< y1 min-y) (> y1 max-y
      ↪ ))
    (and (< x2 min-x) (> x2 max-x) (< y2 min-y) (> y2 max-y
      ↪ ))
    (and (< x3 min-x) (> x3 max-x) (< y3 min-y) (> y3 max-y
      ↪ ))) true ;tri completely inside box
   (or
    (line-box-inc [x1 y1] [x2 y2] [min-x min-y] [max-x
      ↪ max-y])
    (line-box-inc [x2 y2] [x3 y3] [min-x min-y] [max-x
      ↪ max-y])
    (line-box-inc [x3 y3] [x1 y1] [min-x min-y] [max-x
      ↪ max-y])) true ; edge intersecting
   (and
    (< (min x1 x2 x3) min-x)
    (> (max x1 x2 x3) max-x)
    (< (min y1 y2 y3) min-y)
```

```
      (> (max y1 y2 y3) max-y)) true ;box completely inside
         ↪ tri
      :else false
      )))

(defn point-box-inc
  "check if a point is inside an AABB"
  ([[x1 y1 :as point]
    [min-x min-y :as box-min]
    [max-x max-y :as box-max]]
   (and (s>= x1 min-x 0.000001) (s<= x1 max-x 0.000001)
        (s>= y1 min-y 0.000001) (s<= y1 max-y 0.000001)))
  ([point [min-x min-y max-x max-y :as aabb]]
   (point-box-inc point [min-x min-y] [max-x max-y])))

(defn line-line-inc
  "check if two lines intersects"
  [[x1 y1 :as start-1] [x2 y2 :as end-1] [x3 y3 :as start-2]
     ↪ [x4 y4 :as end-2]]
  (let [aabb1
        [(min x1 x2) (min y1 y2)
         (max x1 x2) (max y1 y2)]
        aabb2
        [(min x3 x4) (min y3 y4)
         (max x3 x4) (max y3 y4)]
        a1 (- y2 y1)
        b1 (- x1 x2)
```

```
        c1 (+ (* a1 x1) (* b1 y1))
        a2 (- y4 y3)
        b2 (- x3 x4)
        c2 (+ (* a2 x3) (* b2 y3))
        det (- (* a1 b2) (* a2 b1))]
    (if (zero? det)
      nil
      (let [x (/ (- (* b2 c1) (* b1 c2)) det)
            y (/ (- (* a1 c2) (* a2 c1)) det)
            intersect? (and (point-box-inc [x y] aabb1)
                            (point-box-inc [x y] aabb2))]
        (if intersect?
          [(double x) (double y)]
          nil)))))

;(line-line-inc [0 10] [0 -10] [0 0] [1 0])
;(line-line-inc [-1 0] [1 0] [0 -1] [0 1])
;(line-line-inc [-1 0] [1 0] [-1 1] [1 1])
;(line-line-inc [-1 0] [1 0] [2 -1] [2 1])
;(line-line-inc [-1 0] [2 0] [2 -1] [2 1])
;(line-line-inc [-1 0] [2 0] [2 3] [2 1])
;(line-line-inc [0 0] [2 0] [0 1] [2 1])

(defn point-point-distant
  "distance between two points"
  [[x1 y1] [x2 y2]]
  (let [dx (Math/abs (- x2 x1))
```

```
        dy (Math/abs (- y2 y1))]
    (Math/sqrt (+ (* dx dx) (* dy dy)))))

;(point-point-distant [0 0] [3 4])



(defn distant-closer-to-point [[x1 y1 :as p1]]
  "give a point, returns a function taht takes two points,
  and returns if the distance between two points are closer
   ↪ to the first point"
  (fn [[x2 y2 :as p2] [x3 y3 :as p3]]
    (match [p2 p3]
           [[x2 y2 ] [x3 y3]]
           (let [delta-x1 (Math/abs (- x2 x1))
                 delta-y1 (Math/abs (- y2 y1))
                 delta-x2 (Math/abs (- x3 x1))
                 delta-y2 (Math/abs (- y3 y1))]
             (< (+ delta-x1 delta-y1) (+ delta-x2 delta-y2)))
           [p2 nil] false
           [nil p3] true
           :else false)))

;(reduce into (sorted-set-by (distant-point [0 0]))
;        [[[3 4] [4 4] nil nil [1 1] nil [2 2]]
;         nil
;         [[4 3] [3 3]]])
```

```
(defn line-slice-inc
  "check segment of line and slice intersection.
  returns first intersections in order of their distance to
    ↪ start point"
  [[[sx1 sy1 :as start] [ex2 ey2 :as end] :as line] a-slice]
  (vec (filter (complement nil?) (reduce into (sorted-set-by
    ↪ (distant-closer-to-point start))
    (for [geo a-slice]
      (match [geo]
             [[[x1 y1 & [z1]][x2 y2 & [z2]][x3 y3 & [z3]]]] ;
               ↪ triangle
             [(line-line-inc start end [x1 y1] [x2 y2])
              (line-line-inc start end [x2 y2] [x3 y3])
              (line-line-inc start end [x3 y3] [x1 y1])]
             [[[x1 y1 & [z1]][x2 y2 & [z2]]]] ;line
             [(line-line-inc start end [x1 y1] [x2 y2])]
             :else nil ))))))

;(line-slice-inc [[32 4.5] [-32 4.5]]
;                [[[28 10] [28 0]]
;                 [[29 10] [29 0]]])
;
;(line-line-inc [32 4.5] [-32 4.5] [28 3.5] [28 4.5])

;(line-slice-inc [[0 0] [10 0]]
;                [[[1 1 1] [1 -1 1]]
;                 [[-1 -1 1] [1 1 1] [1 -1 1]]
```

```
;                    [[1  1]]])

;(line-line-inc [0 0] [10 0] [1 1] [1 -1])
;(line-line-inc [-1 0] [10 0] [-1 1] [1 -1])
;(line-line-inc [-1 0] [10 0] [-1 -1] [1 1])


(defn slice-box-inc
  "check if a slice is intersecting an AABB"
  [[min-x min-y max-x max-y :as aabb-box] a-slice]
  (reduce #(or %1 %2) false
    (for [geo a-slice]
      (match [geo]
             [[[x1 y1 z1][x2 y2 z2][x3 y3 z3]]]
               (tri-box-inc [x1 y1] [x2 y2] [x3 y3] [min-x
                 ↪ min-y] [max-x max-y])
             [[[x1 y1 z1][x2 y2 z2]]]
               (line-box-inc [x1 y1] [x2 y2] [min-x min-y] [
                 ↪ max-x max-y])
             [[x1 y1 z1]]
               (point-box-inc [x1 y1] [min-x min-y] [max-x
                 ↪ max-y])
             :else false))))

(defn aabb-tri
  "get aabb from triangle"
  [[[x1 y1 _] [x2 y2 _] [x3 y3 _] :as tri]]
```

```
  [(min x1 x2 x3) (min y1 y2 y3) (max x1 x2 x3) (max y1 y2 y3
    ↪ )])

(defn aabb-line
  "get aabb from line"
  [[[x1 y1 _] [x2 y2 _] :as line]]
  [(min x1 x2) (min y1 y2) (max x1 x2) (max y1 y2)])

(defn aabb-slice
  "get aabb box from a list of geometries"
  [a-slice & [border]]
  (loop [geos a-slice
         min-x 0
         min-y 0
         max-x 0
         max-y 0]
    (if (<= (count geos) 0)
      (if (nil? border)
                [min-x min-y max-x max-y]
                [(- min-x border) (- min-y border) (+ max-x
                  ↪ border) (+ max-y border)])
      (match [(first geos)]
             [[[x1 y1 _] [x2 y2 _] [x3 y3 _]]] ;triangle
               (let [[mx my maxx mayy] (aabb-tri (first geos)
                 ↪ )]
                 (recur (rest geos) (min min-x mx) (min min-y
                   ↪  my) (max max-x maxx) (max max-y mayy)
```

```
                    ↪ ))
                [[[x1 y1 _] [x2 y2 _]]] ;line
                  (let [[mx my maxx mayy] (aabb-line (first geos
                    ↪ ))]
                    (recur (rest geos) (min min-x mx) (min min-y
                      ↪ my) (max max-x maxx) (max max-y mayy)
                      ↪ ))
                [[x1 y1 _]] ;point
                  (recur (rest geos) (min min-x x1) (min min-y
                    ↪ y1) (max max-x x1) (max max-y y1))
                :else
                  (if (nil? border)
                    [min-x min-y max-x max-y]
                    [(- min-x border) (- min-y border) (+ max-x
                      ↪ border) (+ max-y border)])
                      ))))


(defn smaller-than-nozzle?
  "is the current aabb is smaller than the nozzle"
  [[min-x min-y max-x max-y :as aabb] nozzle-diameter]
  (cond
   (<= (- max-x min-x) nozzle-diameter) true
   (<= (- max-y min-y) nozzle-diameter) true
   :else false
   )
  )
```

```
(defn make-square
  "make an aabb BOX square / width = height"
  [[min-x min-y max-x max-y :as aabb]]
  (let [delta-x (Math/abs (- max-x min-x))
        delta-y (Math/abs (- max-y min-y))]
    (if (> delta-x delta-y)
      [min-x min-y max-x (+ delta-x min-y)]
      [min-x min-y (+ delta-y min-x) max-y])))


(defn split-aabb
  "split aabb box to four smaller boxes"
  ([aabb pos]
  {:pre [(keyword? pos)]}
   (let [aabbs (split-aabb aabb)]
     (case pos
       :upper-left (first aabbs)
       :upper-right (second aabbs)
       :lower-left (nth aabbs 2)
       :lower-right (nth aabbs 3)
       :else nil)))
  ([[min-x min-y max-x max-y :as aabb]]
   (let [delta-x (/ (Math/abs (- max-x min-x)) 2)
         delta-y (/ (Math/abs (- max-y min-y)) 2)]
     [[min-x (+ min-y delta-y) (- max-x delta-x) max-y]
      [(+ min-x delta-x) (+ min-y delta-y) max-x  max-y]
      [min-x min-y (- max-x delta-x) (- max-y delta-y)]
      [(+ min-x delta-x) min-y max-x (- max-y delta-y)]]))))
```

```
;;follwing three functions are deprecated due to potential
   ↪ StackOverflowErrors. commented out as referencing
   ↪ matters.
(comment
(declare make-node)

(defn make-leaf
  [aabb a-slice pos nozzle-diameter]
  (let [aabb-node (split-aabb aabb pos)
        toosmall? (smaller-than-nozzle? aabb-node
           ↪ nozzle-diameter)
        intersects? (slice-box-inc aabb-node a-slice)]
    (cond (and toosmall? intersects?) [:leaf aabb-node
       ↪ intersects?]
          (and toosmall? (not intersects?)) [:emptyleaf
             ↪ aabb-node intersects?]
          (and (not toosmall?) (not intersects?)) [:emptyleaf
             ↪  aabb-node intersects?]
          (and (not toosmall?) intersects?) (make-node [(case
             ↪  pos
         :upper-left :floodingleafA
         :upper-right :floodingleafB
         :lower-left :floodingleafC
         :lower-right :floodingleafD)]
      aabb-node a-slice nozzle-diameter)
          :else [:error aabb-node]
```

```
          )))

;;this is a non-tail call recusive function. Need core.async
   ↪ optimization later
;;a better way is to use a set of BFS nodes of the whole tree
   ↪  down to the smallest node
;;then use pmap to check AABB collision and assigned the true
   ↪ /false value
;;way easier and fasters, no conrecursion too.
(defn make-node
  [tree aabb a-slice nozzle-diameter]
  (let [m-leaf #(identity
                  [:node [(make-leaf (split-aabb aabb %)
                    ↪ a-slice :upper-left nozzle-diameter)
                          (make-leaf (split-aabb aabb %)
                            ↪ a-slice :upper-right
                            ↪ nozzle-diameter)
                          (make-leaf (split-aabb aabb %)
                            ↪ a-slice :lower-left
                            ↪ nozzle-diameter)
                          (make-leaf (split-aabb aabb %)
                            ↪ a-slice :lower-right
                            ↪ nozzle-diameter)]
                   aabb (slice-box-inc aabb a-slice)])]
    (match [(first tree)]
           [:floodingleafA]
            (m-leaf :upper-left)
```

```
            [:floodingleafB]
             (m-leaf :upper-right)
            [:floodingleafC]
             (m-leaf :lower-left)
            [:floodingleafD]
             (m-leaf :lower-right))
  ))

(defn make-tree
  "tree construction from a layer of slice"
  [a-slice nozzle-diameter]
  {:pre [(seq? a-slice)
         (number? nozzle-diameter)]}
  (let [aabb (-> (aabb-slice a-slice)
                 make-square)
        aabbs (split-aabb aabb)
        tree [:node
              (make-node [:floodingleafA] (first aabbs)
                ↪ a-slice nozzle-diameter)
              (make-node [:floodingleafB] (second aabbs)
                ↪ a-slice nozzle-diameter)
              (make-node [:floodingleafC] (nth aabbs 2)
                ↪ a-slice nozzle-diameter)
              (make-node [:floodingleafD] (nth aabbs 3)
                ↪ a-slice nozzle-diameter)
              aabb
              (slice-box-inc aabb a-slice)]]
```

```
    tree))
)


(defn tree-nodes-count
  "the totoal number of nodes from height for K-arity based
    ↪ tree"
  [height base]
  (/ (dec (Math/pow base height)) (dec base)))



(defn height
  "given tree or its leafs count, return its height"
  [t b]
  (cond
   (or (seq? t) (vector? t)); if given a tree, return its
     ↪ height
   (let [d (dec b)
         c (count t)
         a (Math/log (inc (* c d)))]
     (int (Math/ceil (/ a (Math/log b)))))
   (number? t); if given a leaf count, return its height
   (let [d (Math/log 4)
         c (Math/log t)]
    (int (inc (Math/ceil (/ c d)))))
   :else nil))


;(height [1] 4)
```

```
;(height (vec (range 5)) 4)
;(height (vec (range 21)) 4)
;(height (vec (range 85)) 4)
;(height 4 4)
;(height 16 4)
;(height 65 4)
;(Math/log 4)



(defn index-to-hrp
  "given tree arity base and the index to one of its node,
    ↪ return height and level position across the same
    ↪ level. OLogN time."
  [ind base]
  {:pre [(pos? base) (not (neg? ind))]
   }
  (loop [h 1]
    (if (>= (tree-nodes-count h base) (inc ind))
      (let [i (int (- ind (tree-nodes-count (dec h) base)))]
      {:height h
       :row-index i})
      (recur (inc h))
      )
    )
)


;repl tests
```

```
;(clojure.repl/doc case)
;(case 2
;  1 :1
;  2 :2)
;(mod 4 4)
;(index-to-hrp 4 4)

(defn aabb-walk
  "given base, height and row-index, returns a walk from the
    ↪ root to the node. OLogN time."
  [b h r]
  {:pre [(< r (Math/pow b h))]}
  (let [div #(quot % b)
        divs #(iterate div %)
        pos #(mod % tree-arity)
        aabb-walk (map #(identity {:position
                                   (case (mod %1 tree-arity)
                                     ↪ 0 :upper-left 1 :
                                     ↪ upper-right 2 :
                                     ↪ lower-left 3 :
                                     ↪ lower-right)
                                   :height %2})
                       (take h (divs r)) ;reverse walking in
                         ↪ row indexes
                       (reverse (range 1 (inc h))))
        ]
    aabb-walk
```

```
    ))

;(aabb-walk 4 1 0)
;(aabb-walk 4 2 0)
;(aabb-walk 4 4 16)
;(mod 15 4)

(defn hr-to-aabb
  "given aabb, base, height and row index, return AABB box in
    ↪ OLogN time."
  [aabb b h r]
  (if (= 1 h)
    aabb
    (loop [walkings (reverse (drop-last (aabb-walk b h r)))
           p (:position (first walkings))
           current-aabb (split-aabb aabb p)]
      (let [next-walkings (rest walkings)]
      (if (empty? next-walkings)
        current-aabb
        (recur next-walkings (:position (first next-walkings)
          ↪ ) (split-aabb current-aabb (:position (first
          ↪ next-walkings)))))))))

;(hr-to-aabb [-10 -10 10 10] 4 2 0)
;(hr-to-aabb [-10 -10 10 10] 4 3 15)

;(quot (quot 63 4) 4)
```

```
;(def tt1 #(quot % 4))
;(def tt2 #(iterate tt1 %))
;(def tt3 #(nth (tt2 %1) (- %2 2)))
;(tt3 16 4)
;(take 3 (tt2 30))
;(iterate #(quot % 4) 63)
;(iterate #(quot % 4) 15)
;(iterate #(quot % 4) 3)


(defn index-to-aabb
  "given aabb, base and index, returns aabb. OLogN time."
  [aabb b i]
  (let [hrp (index-to-hrp i b)]
    (hr-to-aabb aabb b (:height hrp) (:row-index hrp))))


(defn index-to-center
  "given aabb, base and index, returns center point. OLogN
    ↪ time."
  [aabb b i]
  (let [[min-x min-y max-x max-y] (index-to-aabb aabb b i)
        dx (/ (- max-x min-x) 2)
        dy (/ (- max-y min-y) 2)]
    [(+ min-x dx) (+ min-y dy)]))


(defn tree-leaf-size
  "given tree or its leafs count and root AABB box, return
    ↪ its minimum AABB width"
```

```
  [t aabb]
  (let [last-index (if (number? t)
                       (dec t)
                       (dec (count t)))
        [minx miny maxx maxy] (index-to-aabb aabb tree-arity
          ↪ last-index)]
    (min (- maxx minx) (- maxy miny))))

(defn parent
  "given a node index, returns parent index"
  [i]
  {:pre [(integer? i)]}
  (cond
   (= i 0) nil
   (< i 5) 0
   :else (let [hr (index-to-hrp i tree-arity)
               grandparent-node-count (tree-nodes-count (- (:
                 ↪ height hr) 2) tree-arity)
               parent-row-index (Math/floor (/ (:row-index hr
                 ↪ ) tree-arity))]
           (int (+ grandparent-node-count parent-row-index)))
             ↪ )))

;(parent 1)
;(parent 5)
;(parent 21)
;(parent 85)
```

```
;(parent 341)


(defn children
  "given a node index, returns children indexes:
  [child-upper-left child upper-right child-lower-left
   ↪ child-lower-right]"
  [i]
  {:pre [(integer? i)]}
  (let [hr (index-to-hrp i tree-arity)
        a (-> (->> hr
                   :height)
              (tree-nodes-count tree-arity))
        e (-> hr
              :row-index
              inc
              (* 4)
              (+ a)
              dec
              int)]
    [(- e 3) (- e 2) (dec e) e]))


;(children 0)
;(children 1)
;(children 2)
;(children 3)
;(children 4)
;(children 5)
```

```
(defn generate-tree
  "generate tree down to the lowest level in BFS order. O(
    ↪ NLogN) time"
  [a-slice nozzle-diameter & [border]]
  {:pre [(seq? a-slice)
         (number? nozzle-diameter)]}
  (let [[min-x min-y max-x max-y :as aabb] (if (nil? border)
  (-> a-slice aabb-slice make-square)
  (-> a-slice (aabb-slice border) make-square))
  diff-x (- max-x min-x)
  leaf-num (let [round-up (/ diff-x nozzle-diameter)]
  (int (Math/pow round-up 2)))
  tree-height (height leaf-num tree-arity)
  node-count (tree-nodes-count tree-height tree-arity)
  result (atom (vec (repeat node-count nil)))]
    (doseq [ind (range node-count)]
      (cond
       (= ind 0);get collsion test for root node
       (swap! result assoc ind
              (-> (index-to-aabb aabb tree-arity ind)
                  (slice-box-inc a-slice)))
       (true? (nth @result (parent ind)));only get collision
         ↪ test for nodes which parent is collided
       (swap! result assoc ind
              (-> (index-to-aabb aabb tree-arity ind)
                  (slice-box-inc a-slice))))))
```

```
    @result
    ))

(defn adjacent
  "given indexes of two nodes, returns if their AABB boxes
    ↪ are adjacent to each other"
  [n1 n2 aabb]
  {:pre [(integer? n1) (integer? n2)]}
  (let [[min-x1 min-y1 max-x1 max-y1 :as n1-aabb] (
    ↪ index-to-aabb aabb tree-arity n1)
        [min-x2 min-y2 max-x2 max-y2 :as n2-aabb] (
          ↪ index-to-aabb aabb tree-arity n2)]
    (cond
     (= min-x1 max-x2) true
     (= min-x2 max-x1) true
     (= min-y1 max-y2) true
     (= min-y2 max-y1) true
     :else false)))

(defn aabb-inc
  "given two aabb returns if they are intersecting with each
    ↪ other"
  [[min-x1 min-y1 max-x1 max-y1 :as aabb1] [min-x2 min-y2
    ↪ max-x2 max-y2 :as aabb2]]
  (cond
      (and
       (or (and (>= min-x2 min-x1) (<= min-x2 max-x1))
```

```
            (and (>= max-x2 min-x1) (<= max-x2 max-x1)))
        (or (and (>= min-y2 min-y1) (<= min-y2 max-y1))
            (and (>= max-y2 min-y1) (<= max-y2 max-y1))))
          ↪ true
    :else false))

(defn node-inc
  "given indexes of two nodes, returns if their AABB boxes
    ↪ are intersecting with each other"
  [n1 n2 aabb]
  {:pre [(integer? n1) (integer? n2)]}
  (let [n1-aabb (index-to-aabb aabb tree-arity n1)
        n2-aabb (index-to-aabb aabb tree-arity n2)]
    (aabb-inc n1-aabb n2-aabb)))

(defn false-ancestor
  "given a tree and an index of an nil node. returns its
    ↪ ancestor which is a false node"
  [t i]
  {:pre [(-> t (nth i) nil?)]}
  (loop [parent-node (parent i)]
    (cond
      (false? (nth t parent-node)) parent-node
      :else (recur (parent parent-node))
      )
    )
  )
```

```
;(false-ancestor [false nil nil nil false nil nil nil nil nil
  ↪  nil nil nil nil nil nil nil nil nil nil nil nil] 17)

(defn leafs
  "given a tree, returns its leafs"
  [t]
  (let [last-index (-> t count dec)
        hr (index-to-hrp last-index tree-arity)
        last-row-start-index (-> hr :height dec (
          ↪ tree-nodes-count tree-arity) int)]
    (->
     (for [i (range last-row-start-index (inc last-index))]
       (if (nil? (nth t i))
         (let [anc (false-ancestor t i)] [anc (nth t anc)])
         [i (nth t i)]))
     set
     vec)))

;(leafs [true false false false true nil nil nil nil nil nil
  ↪ nil nil nil nil nil nil true false false false])
;(leafs (range 5))
;(leafs (range 21))
;(leafs (range 85))

(defn center-aabb
```

```
  "give aabb, put it to the center so that [0 0 20 20]
    ↪ becomes [-10 -10 10 10]"
  [[min-x min-y max-x max-y :as aabb]]
  (let [delta-x (-> (- max-x min-x) (/ 2) (+ min-x))
        delta-y (-> (- max-y min-y) (/ 2) (+ min-y))]
    [(- min-x delta-x) (- min-y delta-y) (- max-x delta-x)
      ↪ (- max-y delta-y)]))

;(center-aabb [-1 -1 5 5])
;(center-aabb [-11 -11 -5 -5])
;(center-aabb [11 11 15 15])

(defn point-leaf
  "given a point, returns the leaf node where it sits on"
  [p t aabb]
  {:pre [
         ;(do (debugger p "P:") true)
         (number? (first p)) (number? (second p)) (vector? t)
           ↪ (number? (first aabb))
         ]}
  (if (point-box-inc p aabb)
    (let [child-nodes (children 0)
          child-aabbs (map (fn [i] (index-to-aabb aabb
            ↪ tree-arity i)) child-nodes)
          sits-in-node (match [(point-box-inc p (first
            ↪ child-aabbs))
```

```
                                (point-box-inc p (nth
                                  ↪ child-aabbs 1))
                                (point-box-inc p (nth
                                  ↪ child-aabbs 2))
                                (point-box-inc p (nth
                                  ↪ child-aabbs 3))]
                               [true _ _ _] (first child-nodes
                                 ↪ )
                               [_ true _ _] (nth child-nodes
                                 ↪ 1)
                               [_ _ true _] (nth child-nodes
                                 ↪ 2)
                               [_ _ _ true] (nth child-nodes
                                 ↪ 3))]
    (loop [node sits-in-node]
      (cond
        ;out of bounce
        (-> (children node)
            last
            (>= (count t)))
        node
        ;false node is a leaf node
        (false? (nth t node))
        node
        ;true node needs check deeper level
        (nth t node)
        (let [c-nodes (children node)
```

```
                c-aabbs (map (fn [i] (index-to-aabb aabb
                  ↪ tree-arity i)) c-nodes)]
            (recur (match [(point-box-inc p (first c-aabbs))
                           (point-box-inc p (nth c-aabbs 1))
                           (point-box-inc p (nth c-aabbs 2))
                           (point-box-inc p (nth c-aabbs 3))]
                          [true _ _ _] (first c-nodes)
                          [_ true _ _] (nth c-nodes 1)
                          [_ _ true _] (nth c-nodes 2)
                          [_ _ _ true] (nth c-nodes 3))))
          ;you shouldn't be here
          :else :warning-intruder-alert! )))
    nil)
)
```

A.5 Source code of "slicer.flood"

```
(ns slicer.flood
  (:require [clojure.core.match :refer [match]]
            [slicer.tree :as tree])
  (:use slicer.util))



(defn flooding-aabb-gen
  [[min-x min-y max-x max-y :as aabb]]
  (let [w (- max-x min-x)
        h (- max-y min-y)]
```

```
  [[(- min-x w) min-y min-x max-y] ;left box
   [min-x max-y max-x (+ max-y h)] ;upper box
   [max-x min-y (+ max-x w) max-y] ;right box
   [min-x (- min-y h) max-x min-y]])) ;lower box

(defn slow-flood
  "given a tree and aabb generated from the slice,
  flood the tree from outside and return the nodes that are
  ↪ flooded.
  Perfect flood compare to fast-flood, and no border is
  ↪ required."
  [t aabb]
  (let [flooding-aabbs (atom (flooding-aabb-gen aabb))
        leafs (tree/leafs t)
        flooded-nodes (atom #{})]
    (loop [flooded-count (count @flooded-nodes)]
      (doseq [[leaf collided] leafs
              flooding-aabb @flooding-aabbs]
        (let [leaf-aabb (tree/index-to-aabb aabb tree/
        ↪ tree-arity leaf)]
          (match [(tree/aabb-inc leaf-aabb flooding-aabb)
                  (contains? @flooded-nodes leaf)
                  collided]
                 [true false false] ;when the leaf is
                 ↪ intersecting with the flooding nodes;
                 ↪ and not flooded before; and is not
                 ↪ part of the slice matters; flood it
```

```
                    ↪ now
                  (swap! flooded-nodes conj leaf)
                  :else :do-not-care)))
      (reset! flooding-aabbs
              (->> @flooded-nodes
                   vec
                   (map #(tree/index-to-aabb aabb tree/
                     ↪ tree-arity %))
                   vec))
      (if (<= (count @flooded-nodes) flooded-count) ;when
        ↪ flooded nodes are not increasing. water has reach
        ↪  all parts
        (vec @flooded-nodes)
        (recur (count @flooded-nodes)))))))

;since aabb can be smaller than nozzle-diameter, this functin
  ↪  needs extra care

;(range 0.5 1 0.5)
(defn aabb-flood-points
  "generate flooding points for interseciton check from aabb"
  [[min-x min-y max-x max-y :as aabb] leaf-size]
  (let [hl (/ leaf-size 2)
        x-points (range (+ min-x hl)  max-x hl)
        y-points (range (+ min-y hl)  max-y hl)
        left-points (map #(identity [(- min-x hl) %])
          ↪ y-points)
```

```
        right-points (map #(identity [(+ max-x hl) %])
          ↪ y-points)
        up-points (map #(identity [% (+ max-y hl)]) x-points)
        low-points (map #(identity [% (- min-y hl)]) x-points
          ↪ )]
    (-> left-points
        (into right-points)
        (into up-points)
        (into low-points))))

;(aabb-flood-points [0 0 1 1] 1) => ([1/2 -1/2] [1/2 3/2]
  ↪ [3/2 1/2] [-1/2 1/2])
;(aabb-flood-points [0 0 2 2] 1)

;(defn aabb-flood-points
;  "generate flooding points for interseciton check from aabb
  ↪ "
;  [[min-x min-y max-x max-y :as aabb] leaf-size]
;  (let [[mx my :as mid-point] [(-> (- max-x min-x)
;                                   (/ 2)
;                                   (+ min-x))
;                               (-> (- max-y min-y)
;                                   (/ 2)
;                                   (+ min-y))]
;        dx (min leaf-size (- max-x min-x))
;        dy (min leaf-size (- max-y min-y))]
;    [;left column
```

```
;                [(- min-x (/ dx 2)) (- my (/ dy 2))]
;                [(- min-x (/ dx 2)) (+ my (/ dy 2))]
;                ;right column
;                [(+ max-x (/ dx 2)) (- my (/ dy 2))]
;                [(+ max-x (/ dx 2)) (+ my (/ dy 2))]
;                ;upper row
;                [(- mx (/ dx 2)) (+ max-y (/ dy 2))]
;                [(+ mx (/ dx 2)) (+ max-y (/ dy 2))]
;                ;lower row
;                [(- mx (/ dx 2)) (- min-y (/ dy 2))]
;                [(+ mx (/ dx 2)) (- min-y (/ dy 2))]]
;                ))



;(count (range -10 10 0.1))
;(aabb-flood-points [-10 -10 10 10] 0.1)
;(aabb-flood-points [-0.5 -0.5 0.5 0.5] 1)
;(into #{} [1 2 3])

;(def debugging (atom []))
;(def tree-debug (atom nil))
;(def aabb-debug (atom nil))
;(count @debugging)
;(nth @debugging 1)
;(slicer.draw/gui-main @debugging @tree-debug @aabb-debug (
  ↪ str "resources/pic/fdd-1.png"))
```

```
(defn flood-node
  "geometrically flood the nodes that are intersecting with
    ↪ the given aabbs,
  given collision of either true or false.
  the second aabb is the root aabb box for the tree t.
  returns:
  #{leaf-index ...}"
  [aabbs-or-points aabb t leaf-size collision]
  (let [init-flood-points (match [(count (first
    ↪ aabbs-or-points))]
  [2] aabbs-or-points
  [4] (->> aabbs-or-points
  (map (fn [ab] (aabb-flood-points ab leaf-size)))
  (reduce into #{})
  vec))
        init-flooded-leafs (->> init-flood-points
                                (map (fn [p] (tree/point-leaf
                                  ↪  p t aabb)))
                                (filter (complement nil?))
                                (filter (fn [i] (= collision
                                  ↪ (nth t i)))))
        flooded-set (atom (set init-flooded-leafs))]
    (loop [flooding-leafs init-flooded-leafs]
      (let [flood-points (->> flooding-leafs
                              (map (fn [i] (tree/
                                ↪ index-to-aabb aabb tree/
                                ↪ tree-arity i)))
```

```
                                (map (fn [ab] (
                                  ↪ aabb-flood-points ab
                                  ↪ leaf-size)))
                                (reduce into #{})
                                (filter (complement nil?)))
                flooded-leafs (->> flood-points
                                (map (fn [p] (tree/point-leaf
                                  ↪ p t aabb))) ;;get leafs
                                  ↪ for the points
                                (filter (complement nil?))
                                (filter (fn [node] (not (
                                  ↪ contains? @flooded-set
                                  ↪ node)))) ;; remove the
                                  ↪ flooded ones
                                (filter (fn [i] (= collision (
                                  ↪ nth t i)))) ;;filtered
                                  ↪ according to collision
                                  ↪ boolean
                                )]
              (swap! flooded-set into flooded-leafs)
              (if (empty? flooded-leafs) ;; if the new flooded set
                ↪ is not grew, flooding is done
                @flooded-set
                (recur flooded-leafs))))))


;(keys (zipmap (map (comp keyword str) [1 2 3]) (repeat nil))
  ↪ )
```

```
;(vec (reduce into #{} '([1 2 3] [4 5 6])))
;(map inc #{1 2 3})



;(contains? #{1 2 3} 4)
;(conj #{} 1)
;(set nil)
;(set [0 1 2 3 4])
;(trampoline (flood [true false true true true])) => 1
;(trampoline (flood [true true true true false])) => 4
;(tree/leafs [true false true true true])

;(let [a [1 2 3]
;      b (atom a)
;      _ (swap! b rest)]
;  [a @b]
;  )

; Now to look for contained space, I use line-slice
  ↪ intersection.
; Shoot a list of parallel lines along the longer axis where
  ↪ the slice sits.
; If the distance between first and second intersection
  ↪ points is larger than the smallest node (nozzle
  ↪ diameter)
; then we have a flooding point to start.
;
```

```
; But what if the the model has no contained spaced after
  ↪ made as quad-tree,
; then a simple outer flood with all the collided nodes will
  ↪ left nothing.
;
; Out most flood should be done first to see if there is
  ↪ contained space.
; then line intersection checks
; then the inner flooding.
;
; In case of line intersection checks finds no flooding point
  ↪ ,
; while out most flood + collided nodes = all leafs,
; the lines are not generated good enough.


;(map vector
;  (map vector [1 2 3] [3 4 5])
;  (map vector [1 2 3] [3 4 5]))
;
;(into [1 2 3] [4 5 6])



(defn mid-point
  "given two points, return middle point"
  [[x1 y1 :as p1] [x2 y2 :as p2]]
  (let [dx (/ (- x2 x1) 2)
        dy (/ (- y2 y1) 2)]
```

```
    [(double (+ x1 dx)) (double (+ y1 dy))]))

;(mid-point [0 0] [2 2])
;(mid-point [0 0] [-2 -2])
;(mid-point [1 0] [-2 -2])

(defn line-slice-flood-point
  "give a line segment, a slice, returns the flood point or
    ↪ nil if none exist.
  --*x*----------*--x*---"
  [line a-slice leaf-size]
  (let [intersections (tree/line-slice-inc line a-slice)]
    (cond
      (empty? intersections)
      nil
      (even? (count intersections)) ;there are some missing
        ↪ intersecitons, even? will filter those out.
      (let [[x1 y1 :as p1] (first intersections)
            [x2 y2 :as p2] (second intersections)]
        (cond
          (or ;if a min-node aabb can fit in first two points
            (> (Math/abs (- x2 x1)) leaf-size)
            (> (Math/abs (- y2 y1)) leaf-size))
          (mid-point p1 p2) ;(move-point-towards-point p1 p2
            ↪ (* 1.1 leaf-size)) ;return the middle point
          :else nil ))
      :else
```

```
      nil)))

(defn find-contained-flooding-point
  "generate a list of parallel lines from AABB of a slice,
  find the point that is contained in the slice"
  [a-slice leaf-size [min-x min-y max-x max-y :as aabb]]
  (let [x-points (range (+ min-x (* leaf-size 1.5)) (- max-x
    ↪ (* leaf-size 1.5)) (/ leaf-size 2))
        x-start-points (map vector x-points (repeat max-y))
        x-end-points (map vector x-points (repeat min-y))
        x-lines (map vector x-start-points x-end-points)
        y-points (range (+ min-y (* leaf-size 1.5)) (- max-y
          ↪ (* leaf-size 1.5)) (/ leaf-size 2))
        y-start-points (map vector (repeat max-x) y-points)
        y-end-points (map vector (repeat min-x) y-points)
        y-lines (map vector y-start-points y-end-points)
        lines (into x-lines y-lines)]
    (loop [ind 0
           results []]
      (if (>= ind (count lines))
        results
        (let [flood-point (line-slice-flood-point (nth lines
          ↪ ind) a-slice leaf-size)]
          (if (not (nil? flood-point))
            (recur (inc ind) (conj results flood-point))
            (recur (inc ind) results)
            ))))))
```

```
;(find-contained-flooding-point [[[10 10 1] [10 -10 1]]
;                                [[-10 -10 1] [10 10 1] [10
   ↪ -10 1]]
;                                [[1 1]]]
;                                1)
;
;(find-contained-flooding-point [[[1 1 1] [1 -1 1]]
;                                [[-1 -1 1] [1 1 1] [1 -1 1]]
;                                [[1 1]]]
;                               1)

(defn fast-flood
  "above flood is so slow, why not a new one.
  faster, but sacrificed some accuracy.
  This will not flood a slice correctly if tree is not
   ↪ generated with a border.
  border needs to be at least two times of the nozzle size"
  [t aabb a-slice]
  (let [
        ;outer-aabbs (flooding-aabb-gen aabb)
        ;_ (debugger outer-aabbs "outer aabbs:")
        ;outer-nodes (flood-node outer-aabbs aabb t leaf-size
           ↪ false)
        ;debug-nodes (->> contained-points (map (fn [p] (tree
           ↪ /point-leaf p t aabb))) (filter (complement nil
           ↪ ?)))
```

```
        leaf-size (tree/tree-leaf-size t aabb)
        edges (filter (fn [i] (nth t i)) (map first (tree/
          ↪ leafs t)))
        contained-points (find-contained-flooding-point
          ↪ a-slice leaf-size aabb)
        contained-nodes (if (empty? contained-points)
                            nil
                            (flood-node contained-points aabb t
                              ↪ leaf-size false))]
    (into edges contained-nodes)
    ))
```

A.6 Source code of "slicer.eulerian"

```
(ns slicer.eulerian
  (:require [clojure.core.match :refer [match]]
            [clojure.set :as s]
            [slicer.tree :as tree]
            [slicer.flood :as flood])
  (:use slicer.util))


;exmple of an neighbour-set:
;{:neg {:1 #{2 3 4}
;       :2 #{4 1}
;       :3 #{4 1}
;       :4 #{1 2 3}}
; :pos {:1 #{2 3 4}
```

```
;         :2 #{4 1}
;         :3 #{4 1}
;         :4 #{1 2 3}}}

(defn neighbours
  "give the a list of nodes and a leaf node and a neighbour
    ↪ set (neg and pos), returns its neighbours within the
    ↪ list of nodes"
  [node nodes t aabb & [neighbour-set]]
  (let [node-aabb (tree/index-to-aabb aabb tree/tree-arity
    ↪ node)
        leaf-size (tree/tree-leaf-size t aabb)
        neighbour-points (flood/aabb-flood-points node-aabb
          ↪ leaf-size)
        results (->> neighbour-points
                     ;find leaf of each poine
                     (map #(tree/point-leaf % t aabb))
                     ;remove ones that are not in the flooded
                       ↪ nodes
                     (filter #(contains? (set nodes) %))
                     set
                     vec
                     )]
    (if (nil? neighbour-set)
      results
      (->> results
```

```
            (filter (fn [n] ;remove ones that are in the neg
              ↪ neighbour-set
                (let [non-neighbours ((keyword (str node)) (:
                  ↪ neg neighbour-set))]
                  (not (contains? non-neighbours n)))))
            (into ((keyword (str node)) (:pos neighbour-set)))
              ↪  ;add ones in pos neighbour-set
            vec
            ))))

;(contains? (set [1 2 3]) 4)
;(contains? (set [1 2 3]) 3)
;((keyword (str 3)) {:3 #{1 2 3}})

(defn bodd?
  "bigger odd"
  [a]
  (if (> a 2)
    (odd? a)
    false))

;this is too slow and not correct
;(defn remove-odd-deg-nodes
;  "remove adjacent nodes edge that connects nodes of both
  ↪ odd degree."
;  [nodes-with-odd-degrees t aabb pre-set]
;  (let [result-set (atom pre-set)]
```

```
;     (doseq [node-from nodes-with-odd-degrees]
;       (doseq [node-to (neighbours node-from
   ↪ nodes-with-odd-degrees t aabb @result-set)] ;neighbours
   ↪  of odd degrees
;           (swap! result-set update-in
;                  [:neg (keyword (str node-from))]
;                  conj node-to)
;           (swap! result-set update-in
;                  [:neg (keyword (str node-to))]
;                  conj node-from)))
;     @result-set))

(defn first-odd-node
  "find the first node of the searching-nodes that has odd
     ↪ degrees within nodes."
  [searching-nodes nodes t aabb pre-set]
  (loop [ind 0]
    (cond (>= ind (count searching-nodes))
          nil
          (bodd? (count (neighbours (nth searching-nodes ind)
             ↪  nodes t aabb pre-set)))
          (nth searching-nodes ind)
          :else (recur (inc ind)))))

(defn remove-odd-deg-nodes
  "remove adjacent odd-nodes edge that connects odd-nodes of
     ↪ both odd degree."
```

```
[odd-nodes nodes t aabb pre-set searched-nodes]
(let [node-odd-deg (first (s/difference odd-nodes
  ↪ searched-nodes))
      neighbour-node-odd-deg (if (not (nil? node-odd-deg))
                                 (first-odd-node (neighbours
                                   ↪  node-odd-deg nodes t
                                   ↪  aabb pre-set) nodes
                                   ↪ t aabb pre-set)
                                 nil)]
  ;(debugger (count odd-nodes) "counting odd-nodes")
  ;(debugger (count searched-nodes) "counting
    ↪ searched-odd-nodes")
  (match [(nil? node-odd-deg) (nil? neighbour-node-odd-deg)
    ↪ ]
         [false false]; two adjacent odd-nodes with odd
           ↪ degrees are found
         (recur odd-nodes
                nodes t aabb
                (-> pre-set
                  (update-in [:neg (keyword (str
                    ↪ node-odd-deg))] conj
                    ↪ neighbour-node-odd-deg)
                  (update-in [:neg (keyword (str
                    ↪ neighbour-node-odd-deg))] conj
                    ↪ node-odd-deg))
                (conj searched-nodes node-odd-deg
                  ↪ neighbour-node-odd-deg))
```

```
            [false true]; one node of odd degree without
               ↪ neighbour of odd degrees are found, and not
               ↪ all odd-nodes are searched.
            (recur odd-nodes nodes t aabb pre-set (conj
               ↪ searched-nodes node-odd-deg))
            :else
            pre-set
            )))

;(assoc-in {:neg {:1 #{2}}}
;          [:neg :1]
;          (conj (:1 (:neg {:neg {:1 #{2}}})) 3))

(defn min-index [v]
  (first (apply min-key second (map-indexed vector v))))

(defn connect-odd-deg-nodes
  "connect the pair of nodes of odd degrees in a heruistic
     ↪ fashion:
  links each node to its closest non-neighbour"
  [odd-nodes nodes current-node t aabb pre-set]
  (let [neighbour-nodes (neighbours current-node nodes t aabb
     ↪  pre-set)
        searching-odd-nodes (vec (disj (s/difference
           ↪ odd-nodes neighbour-nodes) current-node))
        searching-points (map #(tree/index-to-center aabb
           ↪ tree/tree-arity %) searching-odd-nodes)
```

```
      searching-distances (map #(tree/point-point-distant
                                  (tree/index-to-aabb aabb
                                    ↪ tree/tree-arity
                                    ↪ current-node) %)
                                searching-points)
      min-node (if (not (empty? searching-distances))
                 (nth searching-odd-nodes (min-index
                   ↪ searching-distances))
                 nil)
      next-node (if (>= (count searching-odd-nodes) 2)
                  (first (disj (set searching-odd-nodes)
                    ↪ min-node))
                  nil)]
  (cond
    (number? next-node)
    (recur (disj odd-nodes current-node min-node)
           nodes
           next-node
           t aabb
           (-> pre-set
             (update-in [:pos (keyword (str current-node))]
               ↪  conj min-node)
             (update-in [:pos (keyword (str min-node))]
               ↪ conj current-node)))
    (number? min-node)
    (-> pre-set
```

```
                (update-in [:pos (keyword (str current-node))]
                  ↪ conj min-node)
                (update-in [:pos (keyword (str min-node))]
                  ↪ conj current-node))
        :else
        pre-set)))

;(disj #{1 2 3} 3 2 1)
;(min-key (range 4) [4 4 4 1])
;(min-index [1 2 3 0 4])

(defn convert-to-eulerian
  "given a flooded leaf nodes,
  returns a neighbour-set that will convert the graph that
  ↪ has eularian path"
  [nodes t aabb]
  (let [init-set (zipmap
                   (map (fn [n] (keyword (str n))) nodes)
                   (repeat #{}))
        pre-set {:neg init-set :pos init-set}
        odd-nodes (set (filter (fn [n] (bodd? (count (
          ↪ neighbours n nodes t aabb)))) nodes))
        neg-set (remove-odd-deg-nodes odd-nodes nodes t aabb
          ↪ pre-set #{})
        odd-nodes2 (set (filter (fn [n] (odd? (count (
          ↪ neighbours n nodes t aabb neg-set)))) nodes))
```

```
        final-set (connect-odd-deg-nodes odd-nodes2 nodes (
          ↪ first odd-nodes2) t aabb neg-set)
        ;odd-nodes3 (filter (fn [n] (odd? (count (neighbours
          ↪ n nodes t aabb final-set)))) nodes)
        ]
    final-set
    ;(vec odd-nodes3)
    )
  )

(defn all-edges
  "returns all edges of the flooded node"
  [nodes t aabb & [fix-set]]
  (let [result (atom #{})]
    (doseq [node nodes]
      (doseq [neighbour (if (nil? fix-set)
                          (neighbours node nodes t aabb)
                          (neighbours node nodes t aabb
                            ↪ fix-set))]
        (swap! result conj #{node neighbour})))
    @result))


;(= #{1 2} #{2 1})
;(conj #{} #{1 2} #{2 1})
```

```
;(zipmap [:1 :2 :3] (repeat #{}))
;(assoc {:neg {} :pos {}} :neg {:1 [2 3]})
;(assoc {:neg {:1 [4]} :pos {}} :neg {:1 [2 3]})

(defn random-loop-walk
  [start-node unwalked-edges & [init-node walked-edges]]
  {:pre [(set? unwalked-edges)]}
  (let [the-walked-edges (if (nil? walked-edges) []
    ↪ walked-edges)
        the-init-node (if (nil? init-node) start-node
          ↪ init-node)
        step-edge (first (s/select #(contains? % start-node)
          ↪ unwalked-edges))
        ]
    (cond
      ;walked to an end without anymore step-edge, gives an
        ↪ error
      (nil? step-edge) (throw (Exception. "loop walked failed
        ↪ , graph is not eulerian."))
      ;walked to original position with a loop, walked is
        ↪ finished.
      (= (first (disj step-edge start-node)) the-init-node) (
        ↪ conj the-walked-edges step-edge)
      :else
      (recur (first (disj step-edge start-node))
             (disj unwalked-edges step-edge)
```

```
              [the-init-node (conj the-walked-edges step-edge)
                ↪ ])))))

;(first (disj #{1 2} 1))

;(into
;  (random-loop-walk 1 #{#{1 2} #{6 7} #{5 7} #{7 1} #{2 3}
  ↪ #{4 3} #{4 5} #{6 5}})
;  (random-loop-walk 1 #{#{1 2} #{6 7} #{5 7} #{7 1} #{2 3}
  ↪ #{4 3} #{4 5} #{6 5}}))

;(disj #{#{1 2} #{2 3}} #{2 3})

(defn get-start-node
  [walked-edges unwalked-edges]
  {:pre [(vector? walked-edges)]}
  (if (empty? walked-edges)
    [(first (first unwalked-edges)) 0]
    (let [[edge node]
          (first
            (for [x walked-edges
                  y unwalked-edges
                  :when (not (empty? (s/intersection x y)))]
              [x (first (s/intersection x y))]))
          ;error: this index is not where it should be
            ↪ inserting the new loop.
          ; need search again for the position
```

```
; sometimes place after, sometimes place before
index  (.indexOf walked-edges edge)
;_ (debugger walked-edges "walked-edges")
;_ (debugger (first (s/intersection (first
   ↪ walked-edges) (last walked-edges))) "first
   ↪ and last:")
;_ (debugger (first (s/intersection (nth
   ↪ walked-edges index) (nth walked-edges (inc
   ↪ index)))) "this and after:")
insert-index (cond
               ;first and last node has the start
                  ↪ node
               (=
                 (first (s/intersection (first
                    ↪ walked-edges) (last
                    ↪ walked-edges)))
                 node)
               0
               ;current and one after has the start
                  ↪  node
               (=
                 (first (s/intersection (nth
                    ↪ walked-edges index) (nth
                    ↪ walked-edges (inc index))))
                 node)
               (inc index) ;insert between current
                  ↪ and one after
```

```
                        ;current and one before has the
                           ↪ start node
                        (=
                          (first (s/intersection (nth
                             ↪ walked-edges index) (nth
                             ↪ walked-edges (dec index))))
                          node)
                        index ;insert between current and
                           ↪ one after
                        :else (throw (Exception. "loop␣
                           ↪ insertion␣failed"))
                        )
            ]
        [node insert-index])))

;(get-start-node [#{1 2} #{2 3}] #{#{3 4} #{4 1} #{4 5}})
;(get-start-node [#{6 2} #{2 3}] #{#{3 4} #{4 1} #{4 5}})
;(get-start-node [#{6 2} #{2 3}] #{#{7 4} #{4 1} #{4 5}})
;(get-start-node [] #{#{7 4} #{4 1} #{4 5}})

;(s/select #(contains? % 3) #{#{1 2} #{3 4}} )

;(for [x #{#{1 2} #{2 3}}
;      y #{#{3 4} #{4 1} #{4 5}}
;      :when (not (empty? (s/intersection x y)))]
;  (s/intersection x y))
```

```
;(s/select #(contains? % 1) #{#{2 3} #{1 2}}) => #{#{1 2}}
;(.indexOf [ #{1 2} #{2 3} #{3 4} ] #{1 2})
;(.indexOf [ #{1 2} #{2 3} #{3 4} ] #{2 0}) => -1
;(.indexOf [ #{1 2} #{2 3} #{3 4} ] nil) => -1
;[1 2 3 4 5] => [4 5 1 2 3]
;(subvec [#{1 2} #{2 3} #{3 4}] 0 1)
;(subvec [#{1 2} #{2 3} #{3 4}] 1 3)

(defn shift-edges
  "shift the edges of walked-edges so that the start-node is
    ↪ the last"
  [start-node walked-edges]
  (let [end-edge (first (s/select #(contains? % start-node) (
    ↪ set walked-edges)))
        index-end-edge (inc (.indexOf walked-edges end-edge))
        first-segment (subvec walked-edges 0 index-end-edge)
        last-segment (subvec walked-edges index-end-edge (
          ↪ count walked-edges))]
    (into last-segment first-segment)))

;(shift-edges 1 [#{1 2} #{2 3} #{3 4} #{4 1}])
;(shift-edges 2 [#{1 2} #{2 3} #{3 4} #{4 1}])
;(shift-edges 3 [#{1 2} #{2 3} #{3 4} #{4 1}])
;(shift-edges 4 [#{1 2} #{2 3} #{3 4} #{4 1}])
;(shift-edges 5 [#{1 2} #{2 3} #{3 4} #{4 1}])

(defn hierholzer
```

```
  "recursively␣randomly␣walk␣the␣flooded␣nodes␣until␣all␣
     ↪ edges␣are␣walked,
␣␣returns␣the␣walking␣path"
  [all-edges nodes walked-edges]
  (if (= (count walked-edges) (count all-edges));if all edges
     ↪  are walked
    walked-edges
    (let [
          unwalked-edges (s/difference all-edges (set
             ↪ walked-edges))
          [start-node index-start-node] (get-start-node
             ↪ walked-edges unwalked-edges)
          first-seg (subvec walked-edges 0 index-start-node)
          new-loop (random-loop-walk start-node
             ↪ unwalked-edges)
          last-seg (subvec walked-edges index-start-node (
             ↪ count walked-edges))
          new-walked-edges (into (into first-seg new-loop)
             ↪ last-seg)
          ;_ (debugger new-loop "new-loop:")
          ;_ (debugger start-node "start-node:")
          ;_ (debugger index-start-node "index-start-node:")
          ]
      (recur all-edges nodes new-walked-edges))))

;(into [1] [2 3]) => [1 2 3]
```

```
(defn edge-to-lines
  [edges aabb]
  (for [edge edges]
    [(tree/index-to-center aabb tree/tree-arity (first edge))
      ↪ (tree/index-to-center aabb tree/tree-arity (second
      ↪ edge))]))

(defn edge-to-node-path [edge-path]
  (let [edge-path-s (conj (vec (rest edge-path)) (first
    ↪ edge-path)) ;shift the path
        intersections (map (fn [a b] (first (s/intersection a
          ↪ b))) edge-path edge-path-s)
        result (filter (complement nil?) intersections)
        first-node (first (s/difference (first edge-path) (
          ↪ sorted-set (first result))))
        last-node (first (s/difference (last edge-path) (
          ↪ sorted-set (last result))))
        ]
    (cond
      (= 1 (count edge-path)) (vec (first edge-path)) ;single
        ↪ edge
      (= (last result) first-node) (into [first-node] result)
        ↪ ;if a loop
      (not= (last result) first-node) (into (into [first-node
        ↪ ] result) [last-node]) ;if a path
      :else (throw (Exception. "node path failed"))
      )
```

```
    ))


(defn edge-to-points
  [edges aabb]
  (let [nodes (edge-to-node-path edges)]
    (for [node nodes]
      (tree/index-to-center aabb tree/tree-arity node))))


;(conj [1 2] 3)
;(map #(identity [%1 %2]) [1 2 3] [2 3])
;(edge-to-node-path [#{1 2} #{3 2} #{3 4} #{4 5} #{5 1} ])
;(edge-to-node-path [#{1 2} #{3 2} #{3 4}])
;(edge-to-node-path [#{1 2} #{3 2}])
;(edge-to-node-path [#{1 2} #{1 2}])
;(edge-to-node-path [#{1 2}])
;(s/intersection #{1 2} #{2 3})
;(conj (vec (rest [1 2])) 1)
```

A.7 Source code of "slicer.gcode"

```
(ns slicer.gcode
  (:require [slicer.util :refer :all]
            [slicer.tree :as tree]
            [slicer.flood :as flood]
            [slicer.eulerian :as eulerian])
  (:use clojure.java.io))
```

```
(def header-str
  (str "; generated by embodier 0.0.1" \newline))

(defn point-str [p e]
  (str "G1 X" (first p) " Y" (second p) " E" e \newline))

(defn sum-lst
  "[1 2 3] => [1 3 6]"
  [a]
  (vec (reverse (take (count a) (map #(reduce + %) (iterate
    ↪ drop-last a))))))
;(reduce + [1 2 3 4])
;(def a [10 1 1 1 1 5])
;(sum-lst a)

(defn slice-str
  [cuts last-cmd last-e-height]
  (if (empty? cuts)
    last-cmd
    (let [cut (first cuts)
          init-cmd (str "G1 Z" (:cut-point cut) \newline)
          slice (:result cut)
          ;_ (debugger slice "slice:")
          tree (tree/generate-tree slice 1 2)
          aabb (-> slice (tree/aabb-slice 2) tree/make-square
            ↪ )
          flooded-leafs (flood/fast-flood tree aabb slice)
```

```
          fixing-set (eulerian/convert-to-eulerian
            ↪ flooded-leafs tree aabb)
          edges (eulerian/all-edges flooded-leafs tree aabb
            ↪ fixing-set)
          edge-path (eulerian/hierholzer edges flooded-leafs
            ↪ [])
          points (eulerian/edge-to-points edge-path aabb)
          point-distants (into [last-e-height]
                               (map tree/point-point-distant
                                 ↪ (drop-last points) (rest
                                 ↪  points)))
          extrusions (sum-lst point-distants)
          current-e-height (last extrusions)]
      #(slice-str (rest cuts)
                  (str last-cmd
                       init-cmd
                       (apply str (map point-str points
                         ↪ extrusions)))
                  current-e-height))))


;this funciton is just a placepo for watertight, outline,
  ↪ infill, traversal and extruding accumulation
(defn gcode
  [cuts]
  (trampoline slice-str cuts header-str 0))


(defn write-gcode
```

```
  [gcode-file gcode-str]
  (with-open [g (writer (file gcode-file))]
    (.write g gcode-str)))
```

Appendix B

Printer Simulator

B.1 Source code of “embodier.core”

```
(ns embodier.core
  (:require
    [embodier.webcomponents :as web]
    [embodier.canvasdraw :as draw]
    [goog.events :as events]
    [secretary.core :as secretary :include-macros true :refer
      ↪ [defroute]]
    [reagent.core :as reagent])
  (:import [goog History]
           [goog.history EventType]))

(reagent/render-component [web/app] (.-body js/document))

(secretary/set-config! :prefix "#")
```

```
(defroute upload "/gcode" []
  (reset! web/routes (assoc web/default :gcode-file true)))

(defroute layers "/layers" []
  (reset! web/routes (assoc web/default :layer-view true)))

(defroute "/" []
  (reset! web/routes (assoc web/default :gcode-file true)))

(def history (History.))

(events/listen history EventType.NAVIGATE
  (fn [e] (secretary/dispatch! (.-token e))))

(.setEnabled history true)
```

B.2 Source code of "embodier.fileapi"

```
(ns embodier.fileapi
  (:require
    [clojure.string :as s]))

(defn abs [a]
  (if (< a 0)
    (- 0 a)
    a))
```

```
(defn d [log & logs]
  (.log js/console "========debug========:" (apply print-str
    ↪ log logs)))

(defn reverse-layerscmd
  "reverse each layers cmds"
  [layers-cmds]
  (for [cmds layers-cmds]
    (reverse cmds)))

(defn collapseY
  "stores Y into each command"
  [layers-cmds]
  (for [cmds layers-cmds]
    (loop [resultcmds nil
           counter 0
           last-y 0]
      (if (= counter (count cmds))
        resultcmds
        (recur
          (if (nil? (:y (nth cmds counter)))
            (cons (assoc (nth cmds counter) :y last-y)
              ↪ resultcmds)
            (cons (nth cmds counter) resultcmds))
          (inc counter)
          (if (nil? (:y (nth cmds counter)))
            last-y
```

```
            (:y (nth cmds counter)))))))))


(defn collapseX
  "stores X into each command"
  [layers-cmds]
  (for [cmds layers-cmds]
    (loop [resultcmds nil
           counter 0
           last-x 0]
      (if (= counter (count cmds))
        resultcmds
        (recur
          (if (nil? (:x (nth cmds counter)))
            (cons (assoc (nth cmds counter) :x last-x)
              ↪ resultcmds)
            (cons (nth cmds counter) resultcmds))
          (inc counter)
          (if (nil? (:x (nth cmds counter)))
            last-x
            (:x (nth cmds counter)))))))))


(defn collapseE
  "stores extrusions into each command"
  [layers-cmds]
  (for [cmds layers-cmds]
    (loop [resultcmds nil
           counter 0
```

```
            last-extrusion 0]
      (if (= counter (count cmds))
        resultcmds
        (recur
          (if (nil? (:e (nth cmds counter)))
            (cons (assoc (nth cmds counter) :e last-extrusion
              ↪ ) resultcmds)
            (cons (nth cmds counter) resultcmds))
          (inc counter)
          (if (nil? (:e (nth cmds counter)))
            last-extrusion
            (:e (nth cmds counter)))))))))


(defn collapseXYE
  "stores XY into each command"
  [layers-cmds]
  (for [cmds layers-cmds]
    (let [last-x (atom 0)
          last-y (atom 0)
          last-e (atom 0)]
      (for [cmd cmds]
        (-> cmd
            (#(do (if (not (nil? (:x %))) (reset! last-x (:x
              ↪ %))) %))
            (#(do (if (not (nil? (:y %))) (reset! last-y (:y
              ↪ %))) %))
```

```
            (#(do (if (not (nil? (:e %))) (reset! last-e (:e
              ↪ %))) %))
            (#(if (nil? (:x %)) (assoc % :x @last-x) %))
            (#(if (nil? (:y %)) (assoc % :y @last-y) %))
            (#(if (nil? (:e %)) (assoc % :e @last-e) %)))
        ))))

(defn add-next
  "make single linked list according to extrusions"
  [layers-cmds]
  (for [cmds layers-cmds]
    (for [i (range (count cmds))]
      (cond
        (= 0 i) (assoc (nth cmds i) :next (inc i))
        (= (dec (count cmds)) i) (assoc (nth cmds i) :next
          ↪ nil)
        (> (:e (nth cmds i)) (:e (nth cmds (dec i)))) (assoc
          ↪ (nth cmds i) :next (inc i))
        :else (assoc (nth cmds i) :next nil)))))

(defn collapseZ
  "collapse :z to all points"
  [layers-cmds]
  (loop [resultcmds nil
         counter 0
         last-z 0]
    (if (= counter (count layers-cmds))
```

```
      (reverse (filter #(if (nil? (first %)) false true)
        ↪ resultcmds))
      (recur
        (cons
          (for [cmd (nth layers-cmds counter)]
            (if (nil? (:z cmd))
              (assoc cmd :z last-z)
              nil))
          resultcmds)
        (inc counter)
        (if (= (:z (first (nth layers-cmds counter))) nil)
          last-z
          (:z (first (nth layers-cmds counter)))))))))

(defn cmd-map
  "translate each command to a map like {:x 1, :y 2, ...}"
  [layers-cmds]
  (filter (complement empty?)
    (for [part layers-cmds]
      (filter (complement empty?)
        (for [cmd part]
          (apply merge (filter #(if (not= % nil) true false)
            (for [token (s/split cmd #"\s")]
              (cond
                (re-find #"^X-*\d+.*$" token) {:x (re-find #"
                  ↪ -*\d+\.*\d*" token)}
```

```
                (re-find #"^Y-*\d+.*$" token) {:y (re-find #"
                  ↪ -*\d+\.*\d*" token)}
                (re-find #"^Z-*\d+.*$" token) {:z (re-find #"
                  ↪ -*\d+\.*\d*" token)}
                (re-find #"^F-*\d+.*$" token) {:f (re-find #"
                  ↪ -*\d+\.*\d*" token)}
                (re-find #"^E-*\d+.*$" token) {:e (re-find #"
                  ↪ -*\d+\.*\d*" token)}
                :else nil)))))))))

(defn layered
  "partition the G1 commands array by Z axis movements"
  [str-ary]
  (partition-by (fn [s]
                  (if (re-find (re-pattern "^G1.*Z.*") s)
                    false
                    true))
                str-ary))

(defn filterG1
  "filter out commands that is not started with G1"
  [str-ary]
  (filter (fn [s]
            (if (re-find (re-pattern "^G1.*") s)
              true
              false))
          str-ary))
```

```
(defn readFile [layers file]
  (let [raw-str (-> file .-target .-result)]
    ;(reset! layers (-> raw-str s/split-lines filterG1
      ↪ layered cmd-map collapseZ collapseX collapseY
      ↪ collapseE reverse-layerscmd add-next))
    (reset! layers (-> raw-str s/split-lines filterG1 layered
      ↪  cmd-map collapseZ collapseXYE add-next))
    ;(reset! layers (-> raw-str s/split-lines filterG1
      ↪ layered cmd-map collapseZ collapseX collapseY
      ↪ collapseE reverse-layerscmd remove-long-jumps))
    ;(.log js/console (print-str (s/join "\n" @layers)))
    ;(d (nth @layers 1))
    ;(.log js/console (print-str (nth @layers 2)))
    ))

(defn setOnLoad
  "called by web component of file onload. f (file) being
    ↪ read into layers atom"
  [f layers]
  (let [reader (js/FileReader.)]
    (set! (.-onload reader) (partial readFile layers))
    (.readAsText reader f)))
```

B.3 Source code of "embodier.webcomponents"

```
(ns embodier.webcomponents
```

```
  (:require
    [embodier.fileapi :as file]
    [embodier.canvasdraw :as draw]
    [reagent.core :refer [atom]]))

(def default {:gcode-file false
              :layer-view false})

(def width 640)
(def height 480)
(def routes (atom (assoc default :upload-file true)))
(def layers (atom nil))
(def current-layer-num (atom 0))
(def req-id (atom nil))
(defn notify [text]
  (set! (.-innerHTML (.getElementById js/document "
    ↪ notification")) text))

(defn logo []
  [:div {:style {:font-size "18px"}} "Gcode Viewer"])

(defn github []
  [:a {:href "https://github.com/gzmask/embodier"} "Github"])

(defn header []
  [:div.row
```

```
    [:div.col-md-1 {:style {:background-color "#ccc"}} [:a {:
      ↪ href "#/"} "Home"]]
    [:div.col-md-1 {:style {:background-color "#ccc"}} [:a {:
      ↪ href "#/gcode"} "Gcode File"]]
    [:div.col-md-1 {:style {:background-color "#ccc"}} [:a {:
      ↪ href "#/layers"} "View Layers"]]
    [:div.col-md-1 {:style {:background-color "#ccc"}} [github
      ↪ ]]
    [:div.col-md-2 [logo]]])

(defn upload-button []
  [:input#upload-button {:type "file"
                         :name "files[]"
                         :style {:color "#555"}
                         :on-change #(do
                                       (file/setOnLoad (aget
                                         ↪ (.. % -target -
                                         ↪ files) 0) layers
                                         ↪ )
                                       (reset! routes (assoc
                                         ↪ default :
                                         ↪ layer-view true)
                                         ↪ ))}])

(defn gcode-dropper []
  [:div
    [:div.row
```

```
      [:div.col-md-4.col-md-offset-1
       [:div#file-dropper.bcircle.circle_file [:span.glyphicon
        ↪ .glyphicon-hdd] "Select Gcode File"]]]
     [:div.row
      [:div.col-md-3.col-md-offset-3 [upload-button]]]])

(defn layer-view-before []
  [:div#layer-view-before.bcircle.circle_layer
   [:canvas#mycanvas]])

(defn control-range! [name min max]
     [:div.col-md-10.col-md-offset-2 [:div.input-group
                        [:span.input-group-addon "min:" min]
                        [:input {:type "range"
                                 :name name
                                 :value @current-layer-num
                                 :on-change #(do (notify "
                                  ↪ Rendering...")
                                 (reset! current-layer-num (-> %
                                  ↪  .-target .-value)))
                                 :on-mouse-out #(draw/show-layer
                                  ↪  layers "
                                  ↪ layer-view-before"
                                  ↪ current-layer-num req-id)
                                 :min min :max max
                                 :style {:padding-top "4px"}
                                 }]
```

```
                    [:span.input-group-addon "at:"
                      ↪ @current-layer-num]
                    [:span.input-group-addon "max:" max]]])

(defn layer-viewer []
  [:div#layer-view.row
   [:div.col-md-8
    [:div.row
     [control-range! "layer" 1 (dec (count @layers))]]
    [:div.row
     [:div.col-md-11.col-md-offset-1 [layer-view-before]]]
    [:div.row
     [:span.col-md-offset-6 "hold 'D' to drag"]]]])

(defn app []
  [:div
   [header]
   [:div.row
    [:div#notification.col-md-12 {:style {:color "#888" :
      ↪ font-size "20px"}} " "]]
   (for [x (range 4)] ^{:key x} [:br])
   (if (:layer-view @routes) [layer-viewer])
   (if (:gcode-file @routes) [gcode-dropper])
   ])
```

B.4 Source code of “embodier.canvasdraw”

```
(ns embodier.canvasdraw)

(def THREE js/THREE)

(defn notify [text]
  (set! (.-innerHTML (.getElementById js/document "
    ↪ notification")) text))

(defn three-partics
  "given a collection of points ({:x ?, :y ?}, ...), returns
    ↪ a threejs ParticleSystem"
  [points color]
  (let [geo (THREE.Geometry.)
        mat (THREE.ParticleBasicMaterial. (clj->js {:size
          ↪ 0.2, :color color}))
        p-list (for [p points] (THREE.Vector3. (:x p) (:y p)
          ↪ (:z p)))
        setup (set! (-> geo .-vertices) (apply array p-list))
        partics (THREE.ParticleSystem. geo mat)]
    partics))

(defn three-line
  "given two of points ({:x ? :y ? :z ?} ...), returns a
    ↪ threejs line"
  [points color]
  (let [geo (THREE.Geometry.)
```

```
        mat (THREE.LineBasicMaterial. (clj->js {:color color
          ↪ }))
        line (THREE.Line. geo mat)
        p-list (for [p points] (THREE.Vector3. (:x p) (:y p)
          ↪ (:z p)))]
    (set!
      (-> geo .-vertices)
      (apply array p-list))
    line))

(defn draw-lines
  "given cmds with :next, add lines to scene in pairs
    ↪ accordingly"
  [cmds scene color]
  (doseq [cmd cmds]
    (if (not (nil? (:next cmd)))
      (let [p [cmd (nth cmds (:next cmd))]]
        (.add scene (three-line p color))))))

(defn trackball-control [cam render dom]
  (let [control (THREE.TrackballControls. cam dom)]
    (set! (.-rotateSpeed control) 1.0)
    (set! (.-zoomSpeed control) 1.2)
    (set! (.-panSpeed control) 0.8)
    (set! (.-noZoom control) false)
    (set! (.-noPan control) false)
    (set! (.-staticMoving control) true)
```

```
    (set! (.-dynamicDampingFactor control) 0.3)
    (set! (.-keys control) (array 65 83 68))
    (.addEventListener control "change" render)
    control))

(defn update-scene
  "add layers from file-api into the scene"
  [scene layers current-layer]
  (let [children (.-children scene)]
    (loop [i (dec (count children))]
      (if (< i 0)
        (do
          (draw-lines (nth @layers @current-layer) scene 0
            ↪ x00ff00)
          (loop [i (dec @current-layer)]
            (if (< i 0)
              (notify "Render is done.")
              (recur (do (.add scene (three-partics (nth
                ↪ @layers i) 0x000088))
                         (dec i))))))
        (recur
          (do
            (.remove scene (aget children i))
            (dec i)))))))

(defn NaN? [node]
  "this is the js nil."
```

```
  (and (= (.call js/toString node) (str "[object Number]"))
       (js/eval (str node " != +" node ))))


(defn get-center
  "given an array of {:x :y :z} points, find the center point
    ↪ "
  [cmds]
  (reduce (fn [p p-]
            {:x (/ (+ (js/parseInt (:x p))
                      (js/parseInt (:x p-))) 2)
             :y (/ (+ (js/parseInt (:y p))
                      (js/parseInt (:y p-))) 2)
             :z (/ (+ (js/parseInt (:z p))
                      (js/parseInt (:z p-))) 2)}) cmds))


(def width 640)
(def height 480)
(def scene (THREE.Scene.))
(def camera (THREE.PerspectiveCamera. 75 (/ width height) 0.1
  ↪  1000))
(set! (.-y (.-position camera))  -25)
(set! (.-z (.-position camera))  25)
(def renderer (THREE.WebGLRenderer.))


(defn show-layer
  [layers dom-id current-layer req-id]
  (let [dom (.getElementById js/document dom-id)
```

```
      center-point (get-center (nth @layers @current-layer)
        ↪ )
      render #(.render renderer scene camera)
      control (trackball-control camera render dom)
      animate (fn an[]
                (reset! req-id (js/requestAnimationFrame an
                  ↪ ))
                (.update control)
                (render))]
  (.setSize renderer width height)
  (set! (.-innerHTML dom) "")
  (.appendChild dom (.-domElement renderer))
  (update-scene scene layers current-layer)
  (set! (.-target control) (THREE.Vector3. (:x center-point
    ↪ ) (:y center-point) (:z center-point)))
  (js/cancelAnimationFrame @req-id)
  (animate)
  ))
```

Appendix C

Links

1. Source Code of Embodier slicer:

   https://github.com/gzmask/embodier.stl.slicer

2. Source Code of Embodier Gcode viewer:

   https://github.com/gzmask/embodier-gcode-webgl

3. G-Code viewer:

   http://gzmask.github.io/embodier-gcode-webgl

References

[1] Hideo Kodama, "A Scheme for Three-Dimensional Display by Automatic Fabrication of Three-Dimensional Model", IEICE TRANSACTIONS on Electronics (Japanese Edition), Vol.J64-C, No.4, pp. 237-241, April 1981.

[2] Hideo Kodama, "Automatic Method for Fabricating A Three-dimensional Plastic Model with Photo-hardening Polymer", Review of Scientific Instruments, Vol.52, No. 11, pp. 1770-1773, November 1981.

[3] PCMag.com, "3D Printing: What You Need to Know", web, http://www.pcmag.com/slideshow_viewer/0,3253,l=293816&a=289174&po=1,00.asp, Retrieved October 2013.

[4] Chee Kai Chua, Kah Fai Leong, Chu Sing Lim, "Rapid Prototyping: Principles and Applications", World Scientific, Vol. 1, pp. 124, 2003

[5] Hassler Whitney, "Differentiable manifolds", The Annals of Mathematics, Vol.37, pp. 645-680, July 1936.

[6] Ching-Kuang Shene, "Mesh Basics - Computing with Geometry", web, http://www.cs.mtu.edu/s̃hene/COURSES/cs3621/SLIDES/Mesh.pdf, pp 3, Spring 2012.

[7] Weiming Wang, Tuanfeng Y. Wang, Zhouwang Yang, Ligang Liu, Xin Tong, Weihua Tong, Jiansong Deng, Falai Chen, Xiuping Liu, "Cost-effective Printing

of 3D Objects with Skin-Frame Structures", ACM Transactions on Graphics (SIGGRAPH Aisa), Vol.32, No.5, Article 177: pp. 1-10, 2013

[8] Rylan Grayston, "The Peachy Printer", Kickstarter project campaign, web, https://www.kickstarter.com/projects/117421627/the-peachy-printer-the-first-100-3d-printer-and-sc/description, September 2013.

[9] Rylan Grayston, "Known Issues & Delays", The Peachy Printer Weekly Update #38, Web,
https://youtu.be/edYiZAQKbDw, June 2015.

[10] Chinese Cities Instance, "University of Waterloo Traveling Saleman Problems", Web,
http://www.math.uwaterloo.ca/tsp/world/countries.html, retrieved June 2015.

[11] China Computation Log, Web,
http://www.math.uwaterloo.ca/tsp/world/chlog.html, September 15 2001.

[12] Mircea Marin, "Lecture 10: Eulerian trails and circuits. Hamiltonian paths and cycles.", Web,
http://web.info.uvt.ro/ mmarin/lectures/GTC/L-11e.pdf, December 2014.

[13] Kenneth Rosen, "Graph Terminology and Special Types of Graphs", Discrete Mathematics and Its Applications (7th Ed), Ch. 10-2, pp. 653, 2012.