An Optimized Slicer For 3D Printing

Xinqi Bao, Dov Kruger

1. Abstract

In order to print a 3D model, it must be analyzed, cut into slices and each slice must generate gcode instructions that a 3D printer can execute. XbSlicer is a high-performance slicing program implemented in C++, attempting to improve performance and reliability of printing. Existing open source slicers such as Cura, Slic3r, and MatterControl are too slow. These three are related because they are all coded in C# and share the same code base. They all crash on large, complex models. XbSlicer uses optimal algorithms to split the model into layers and print by reducing total motion of the print head.

The implementation of slicing is so efficient that profiling revealed I/O standard library functions took 20 percent of runtime. An efficient method using buffers and low layer code has been implemented. As a result of these optimizations, XbSlicer is 20-25 times faster than these open source slicers and successfully handles large, complex models.

The majority of models today are stored as STL in either ASCII or binary. STL is inherently inefficient because points are repeated multiple times. Support has been added for different file types. A new file type, BIF will be implemented in the future to reduce redundant points. The result should reduce reading by factor of 2 and eliminate computation to determine equivalent points.

2. Introduction

The technology of 3D printing is increasingly popular. The biggest advantage compared with the traditional manufacture is rapid prototyping and reducing material waste. For some complex models, 3D printing technology also can print objects that cannot be produced any other way.

There are three major open source packages for slicing: Cura, Slic3r, and MatterControl. All three are slow, take minutes to slice big files, and often crash on large models.

Figure 1 shows an inefficient path example from model "Bunny" using MatterControl. It goes around the object, with a great deal of wasted motion. This paper describes the optimizations that make XbSlicer faster, as well as generating a better, but still not perfect path. The analysis of each layer attempts to define an optimal path for printing. Figure 2 shows the efficient path using XbSlicer. It saves time for not only slicing, but also printing.

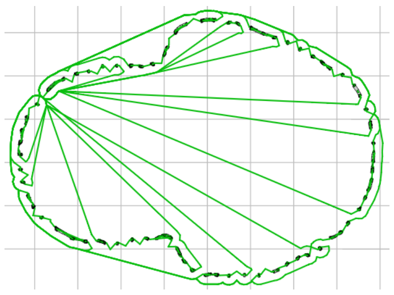


Figure 1: Inefficient path in MatterControl

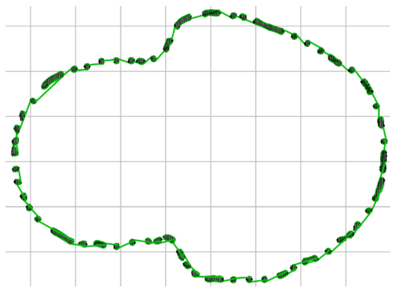


Figure 2: Efficient path in XbSlicer

The main process is getting triangles from the model, calculating each layer's lines which intersected with triangles, making and filling loops, and generating to gcode. The whole program structure shown in Figure 3.

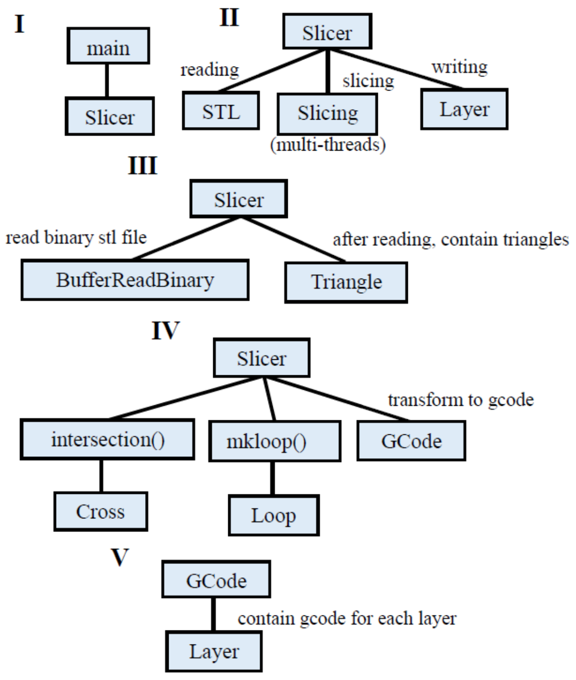


Figure 3: Object structure of XbSlicer

Comparing with MatterControl, XbSlicer can do the same job much faster, and won't crash.

3. Previous work

Jing Hu introduced a framework about layer contour reconstruction algorithms by STL-based slicing process for 3D printing [1]. The experimental results by the traditional uniform slicing show the contour outline of each layer and comparison among the different slicing thickness of the cutting z-plane. Fabian Schurig also used this same idea but in different algorithms in staircases [2].

S.H.Choi et al. presented a memory efficient tolerant-slicing algorithm [3][4]. Instead of storing the whole model into the computer memory, it reads only the facets of the current layer, hence greatly reduces the amount of computer memory required and involves less computationally intensive searching operations.

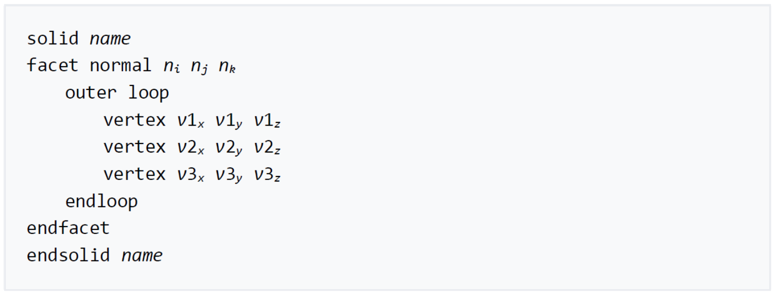
Rodrigo Minetto and his team described an algorithm for slicing an unstructured triangular mesh model by series of parallel planes. They also described an asymptotically optimal linear time algorithm for constructing a set of polygons from the unsorted lists of line segments produced by the slicing step [5].

Current open source slicers: Cura (created in 2016), MatterControl (created in 2015), and Slic3r (created in 2011) provided the idea of the whole process about slicing. They have been covered to handle the most models in normal task.

4. Reading Model Files

3D models consist of triangles. By reading models, it's actually reading triangles and storing them into memory. The standard file for 3D model printing is called STL. There are two versions, ASCII and binary.

ASCII format takes the following form:



```
solid name
facet normal n_i n_j n_k
    outer loop
        vertex v1_x v1_y v1_z
        vertex v2_x v2_y v2_z
        vertex v3_x v3_y v3_z
    endloop
endfacet
endsolid name
```

Figure 4: Format of ASCII STL file

The first and the last line appear only once. The first line contains the name of the model. There will be many facets. Each facet consists of a normal and three vertexes. Three vertexes represent as a triangle, and the normal vector decide which side of triangle is outside for this model.

The binary file is similar but without meta data:

```
UINT8[80] – Header
UINT32 – Number of triangles

REAL32[3] – Normal vector
REAL32[3] – Vertex 1
REAL32[3] – Vertex 2
REAL32[3] – Vertex 3
UINT16 – Attribute byte count
```

Figure 5: Format of Binary STL file

The first 80 characters are just comment. Since ASCII STL starts with the word ”solid”, binary STL should never start with ”solid” because that identifies ASCII file format.

The comment is a 4 bytes little-endian integer specifying the number of triangles in this file. With this information, a simple loop can be used to read all triangles in the file.

Each triangle is described by 50 bytes. Similar to the ASCII format, there are four 3D vectors for each triangle, three for the vertexes and one for the normal vector. In other words, 12 little-endian floats (48 bytes), and 2 bytes undefined. The last two bytes in each triangle are reserved for attribute. It can be the color of this triangle or something else.

4.1. Benefits for reading binary STL file

Binary files are much smaller than ASCII files. The size by itself makes binary STL faster.

In ASCII files, numbers are represented as strings. However, in addition, the program must to read in strings, and convert to double. Since model files can be quite large, this can take a significant amount of time. Furthermore, each triangle must be checked first. Only after ensuring it is a triangle, can it be read in and stored.

In binary files, the number of triangles for the model is known because it is stored in the header. Therefore, an array of triangles can be pre-allocated for greater speed, and no formatting is done.

4.2. Improved Format for Model File

A new format has been designed for this program: BIF (Binary Indexed Format).

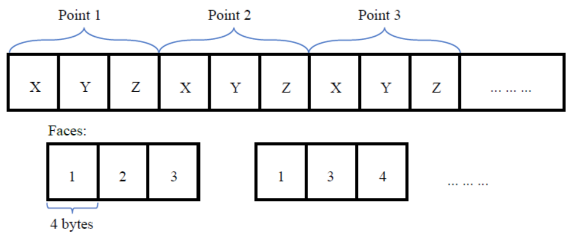


Figure 6: BIF Format

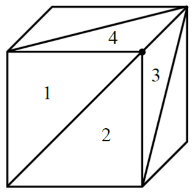


Figure 7: Point shared by multiple triangles

The set of all points in the model is first defined. Triangles consist of 3 indices identifying which points are in each triangle, shown in Figure 6. Since BIF stores each point only once, and STL often has 3 to 6, a factor of 2 reduction in size is expected. Point shared by multiple triangles shown in Figure 7.

5. Slicing

5.1. Intersections

Once the triangles are loaded in RAM, the layers must be computed. Since slicing a particular layer, the Z value is known. The intersections crossed by triangles and layer can be calculated.

Triangles can intersect each layer in different ways, shown in Figure 8.

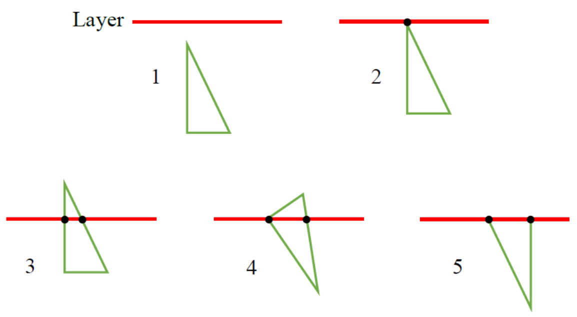


Figure 8: different positions for layer and triangles

Case 1:

The minimum or maximum do not match the height, no intersection.

Case 2:

Crossing by one vertex, not count.

Case 3:

Two edges crossing the layer, calculate the intersection and push to object.

Case 4:

Crossing by one vertex and one edge.

Case 5:

One of the edges is on the layer. Have to check if the same intersection exists in the object already, if not, add to it.

After calculating all the intersections for the layer, link them to form closed loops. Layer may contain several loops, each loop represented as an object separately.

5.2. Assign triangles

Typically, for a particular layer, not all the triangles are necessary to be checked, only those have potential to cross the layer should. By assigning triangles, push every potential triangle into groups with different range of Z value. Triangles may cross more than one group. Shown in Figure 9, set the bigger one triangle into group 0, 1, and 2, and the smaller one only into group 1.

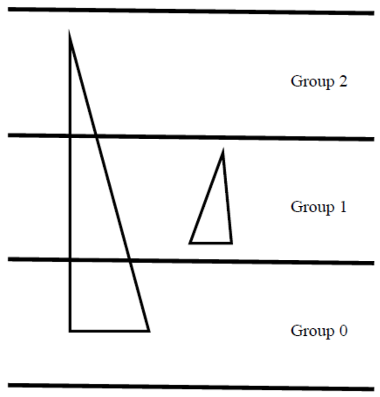


Figure 9: Assign triangles

6. GCode

Gcode is the command used by 3D printers and CNC. The major commands used in 3D printers are G0 and G1, with parameters X, Y, Z, F, and E. G0 tells the printer to move to a position without extruding material. G1 means moving and extruding. X, Y, and Z are the coordinate values. F represents the speed for printing head to move. E is the value of extruder position.

```
G0 F6600 Z0.2  //move head to Z at 0.2
G1 F783 X10 Y0 E0.0299589 //move to X=10, Y=0 with extruding
G0 F6600 X10 Y10 Z0.5  //move to X=10, Y=10, and Z=0.5
```

In G1 command, there must be an E value greater than its previous value instructing the printer to extrude filament.

$$\Delta E = distance(old(X,Y), new(X,Y)) * factor$$

6.1. The idea to generate gcode

For each layer, the minimum and maximum X and Y are known. First the contours of each loop are printed. In order to fill the inside of each loop, a line parallel to the Y axis is scanning through the minimum X to maximum, shown in Figure 10.

For each position, several points can be computed out by crossing with loops. Using G1 command to connect first two points, then G0, G1 and go on, shown as Figure 11.

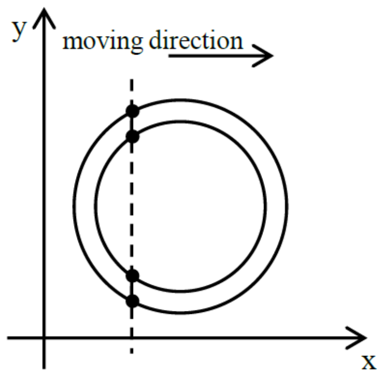


Figure 10: crossing with loops

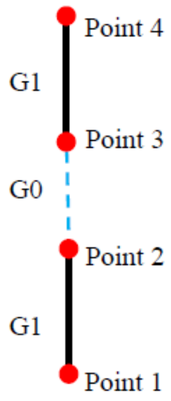


Figure 11: generate G0 and G1 commands

6.2. Loop by loop

However, this traditional solution might not be efficient for actual printing. Shown as Figure 12, lots of G0 commands have been used in this layer, which are doing nothing but moving around. An optimized algorithm has been implemented to reduce these, by printing loop by loop instead all the loops at the same time.

By defining loop and subloops structure to split separate loops. Subloops are filled with container loop following the traditional method. Once finished a container loop, it will go to the next loop which is the closest one. The result shown in Figure 13.

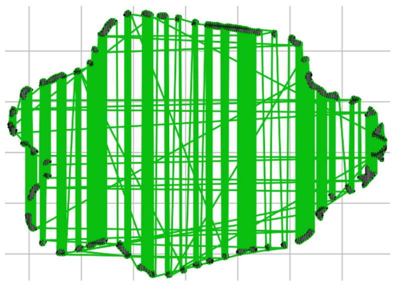


Figure 12: traditional way to print a layer

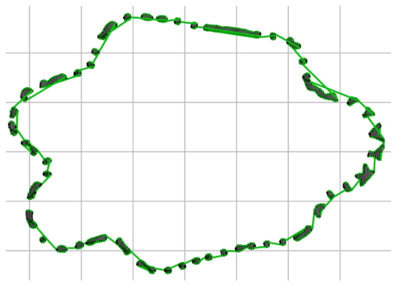


Figure 13: loop by loop algorithm to print a layer

6.3. Assign intersections

Similar method with assigning triangles, intersections also can be assigned to several groups by the range of X. Then only the partial of intersections is computed.

7. Implementations

7.1. High-Performance binary I/O

For the highest performance to read binary files, a custom buffer class is used to load entire blocks directly from the hard drive. Since each triangle is known to be exactly 50 bytes, this saves much time on reading.

Shown as Figure 14, there is buffer and prebuffer in this block. Prebuffer is a memory right before the buffer. Triangles are read from the buffer until only a fractional part remains. Then move the remaining part into prebuffer, and load next block to buffer.

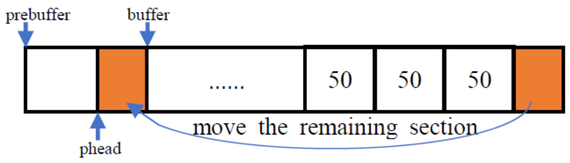


Figure 14: Buffer for reading

While reading the triangles, the Slicer object calculates the minimum and maximum for X, Y, and Z of the model. The reason for doing this is to center the whole model on the printer.

7.2. Writing buffer

The writing buffer is similar to the reading buffer. It has a buffer and post buffer, shown in Figure 15.

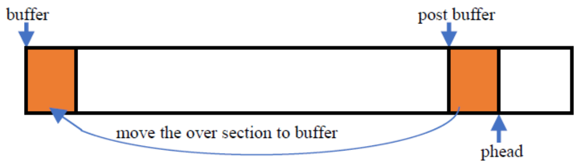


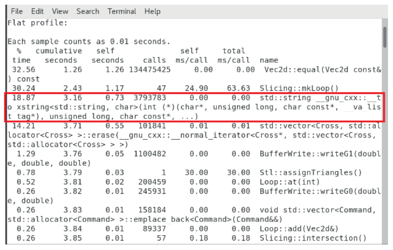
Figure 15: Buffer for writing

Writing characters into buffer, until the writing pointer is over the buffer range and into the post buffer, flush the buffer to hard drive, then move the over part in post buffer to buffer.

In order to test performance of this writing buffer, 10 million random double numbers have been written, and the running speed is 84 MB/s. On the same device, the direct writing to drive speed is 405 MB/s. Even this buffer has to turn double value to string firstly, that means more work have to be done than direct writing, the performance is still great.

7.3. Custom double to string

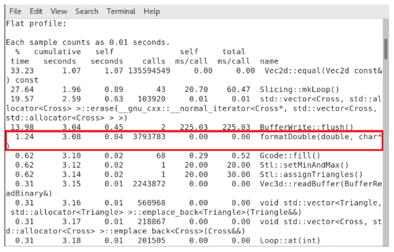
A bunch of double values are written to the file, they have to be converted into string type before output. The to_string() function in c++ library has been used in first place, but it is not efficient enough. Each time calling to_string() function, it will allocate a memory for string, then copy it into buffer. It is unnecessary for this extra memory, no doubt it would take more time. A faster way has been implemented to put the result into buffer directly. Figure 16 shows the profile by using to_string() function.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 32.56      1.26     1.26 134475425     0.00     0.00  Vec2d::equal(Vec2d const&
) const
 30.24      2.43     1.17       47    24.90    63.63  Slicing::mkLoop()
 18.87      3.16     0.73  3793783     0.00     0.00  std::string __gnu_cxx::__t
o xstring<std::string, char>(int (*)(char*, unsigned long, char const*, __va_lis
t_tag*), unsigned long, char const*, ...)
 14.21      3.71     0.55   101841     0.01     0.01  std::vector<Cross, std::al
locator<Cross> >::erase(__gnu_cxx::__normal_iterator<Cross*, std::vector<Cross,
std::allocator<Cross> > >)
  1.29      3.76     0.05  1100482     0.00     0.00  BufferWrite::writeG1(doubl
e, double, double)
  0.78      3.79     0.03        1    30.00    30.00  Stl::assignTriangles()
  0.52      3.81     0.02   200459     0.00     0.00  Loop::at(int)
  0.26      3.82     0.01   245931     0.00     0.00  BufferWrite::writeG0(doubl
e, double)
  0.26      3.83     0.01   158184     0.00     0.00  void std::vector<Command,
std::allocator<Command> >::emplace_back<Command>(Command&&)
  0.26      3.84     0.01    89337     0.00     0.00  Loop::add(Vec2d&)
  0.26      3.85     0.01       57     0.18     0.18  Slicing::intersection()
```

Figure 16: Profile by using library function



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.23      1.07     1.07 135594549     0.00     0.00  Vec2d::equal(Vec2d const&
) const
 27.64      1.96     0.89       43    20.70    60.47  Slicing::mkLoop()
 19.57      2.59     0.63   103920     0.01     0.01  std::vector<Cross, std::al
locator<Cross> >::erase(__gnu_cxx::__normal_iterator<Cross*, std::vector<Cross,
std::allocator<Cross> > >)
 13.98      3.04     0.45        2   225.03   225.03  BufferWrite::flush()
  1.24      3.08     0.04  3793783     0.00     0.00  formatDouble(double, char*
)
  0.62      3.10     0.02       68     0.29     0.52  Gcode::fill()
  0.62      3.12     0.02        1    20.00    20.00  Stl::setMinAndMax()
  0.62      3.14     0.02        1    20.00    30.00  Stl::assignTriangles()
  0.31      3.15     0.01  2243872     0.00     0.00  Vec3d::readBuffer(BufferRe
adBinary&)
  0.31      3.16     0.01   560968     0.00     0.00  void std::vector<Triangle,
 std::allocator<Triangle> >::emplace_back<Triangle>(Triangle&&)
  0.31      3.17     0.01   218867     0.00     0.00  void std::vector<Cross, st
d::allocator<Cross> >::emplace_back<Cross>(Cross&&)
  0.31      3.18     0.01   201505     0.00     0.00  Loop::at(int)
```

Figure 17: Profile by using custom function

The optimized function reads the exponent from binary double format. Getting 3 digits each time, searching in an array and getting characters for each digit. The exactly 6 digits in decimal number will be written into gcode file.

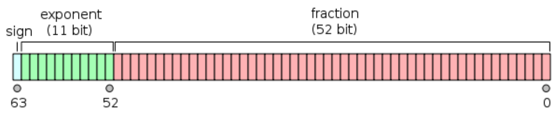


Figure 18: Binary double format

$$realvalue = (-1)^{sign}(1.b_{51}b_{50}\ldots b_0)_2 * 2^{e-1023}$$

Once having the exponent e, the number is kwon in the range 2^{e-1023} to 2^{e-1022}. Then split it into integer and decimal, write separately. An array of characters has been hard code, which contains all the digits for 0 to 999. Using three digits as index, get the digits from array and write them directly into buffer.

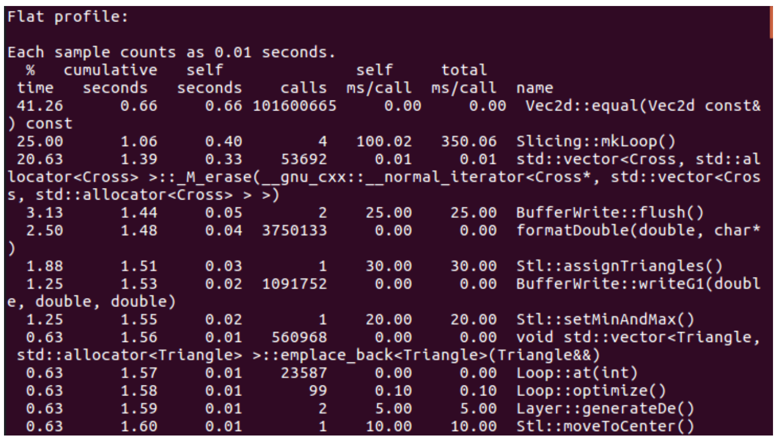
These two different methods have been tested on writing 10 million random double numbers. As the result, the new method runs 20 times faster than to_string() function. The profile by using this custom function shown in Figure 17.

7.4. Multithreads

Multithreads have been implemented for XbSlicer. Multithreads will help CPU usage covered more efficient. As the result, XbSlicer has been speeded up by 1.97, 2.63, and 3.57 times for using 2, 4, and 8 threads.

8. Results

The final profile for XbSlicer shown in Figure 19.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 41.26      0.66     0.66 101600665     0.00     0.00  Vec2d::equal(Vec2d const&
) const
 25.00      1.06     0.40        4   100.02   350.06  Slicing::mkLoop()
 20.63      1.39     0.33    53692     0.01     0.01  std::vector<Cross, std::al
locator<Cross> >::_M_erase(__gnu_cxx::__normal_iterator<Cross*, std::vector<Cros
s, std::allocator<Cross> > >)
  3.13      1.44     0.05        2    25.00    25.00  BufferWrite::flush()
  2.50      1.48     0.04  3750133     0.00     0.00  formatDouble(double, char*
)
  1.88      1.51     0.03        1    30.00    30.00  Stl::assignTriangles()
  1.25      1.53     0.02  1091752     0.00     0.00  BufferWrite::writeG1(doubl
e, double, double)
  1.25      1.55     0.02        1    20.00    20.00  Stl::setMinAndMax()
  0.63      1.56     0.01   560968     0.00     0.00  void std::vector<Triangle,
 std::allocator<Triangle> >::emplace_back<Triangle>(Triangle&&)
  0.63      1.57     0.01    23587     0.00     0.00  Loop::at(int)
  0.63      1.58     0.01       99     0.10     0.10  Loop::optimize()
  0.63      1.59     0.01        2     5.00     5.00  Layer::generateDe()
  0.63      1.60     0.01        1    10.00    10.00  Stl::moveToCenter()
```

Figure 19: Profile for XbSlicer

Comparing XbSlicer with three major open source slicers, three different models have been tested, which are ellipticalvase, liberty, and bunny.

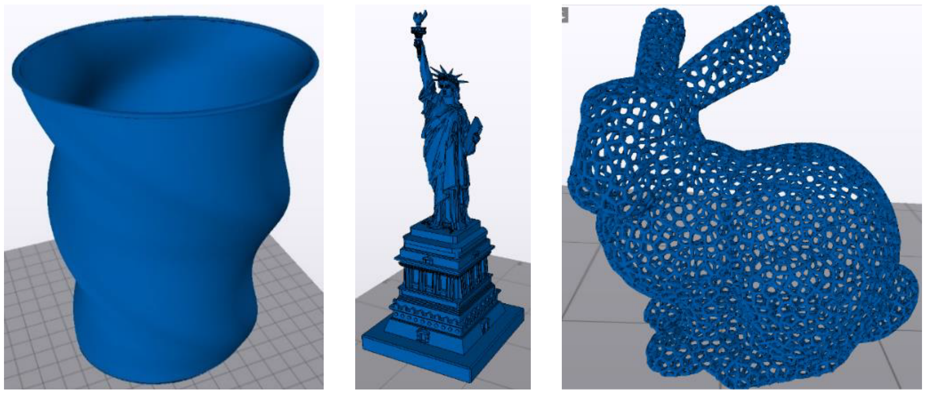


Figure 20: Test models: ellipticalvase, liberty, and Bunny

Table 1: Time consuming on three test models sliced by different slicers

time/s	ellipticalvase	liberty	Bunny
XbSlicer	3.914	0.210	0.736
MatterControl	30.96	8.50	42.17
Cura	10.16	6.65	9.86
Slic3r	14.68	7.59	35.56

Table 1 shows the time consuming on different models and different slicers. In these three test models, the ellipticalvase stored as an ASCII STL, reading the file dominates the time, which makes less speeding up by XbSlicer. The other two models are both in binary.

In average, to slice two binary models, XbSlicer is 37.88 times faster than open source slicers. For the ASCII model, it still runs 4.75 times faster.

9. Improvements

The optimization used in loop by loop method is not the best. It can only optimize some part of the scenarios, shown in Figure 21.

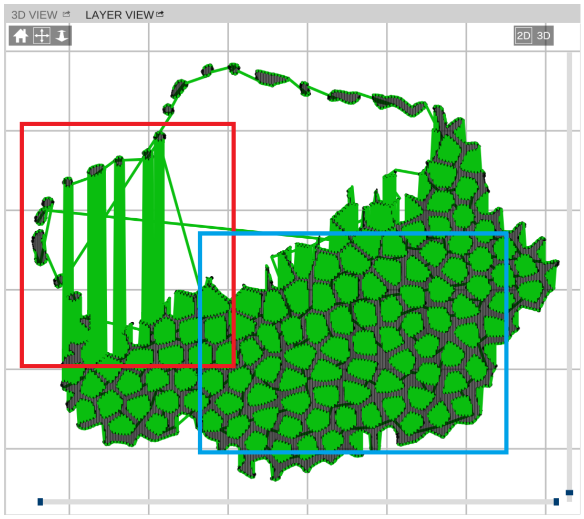


Figure 21: Defects of loop by loop algorithm

In the left area, it still using G0 between loops. Which means these loops are not following the way they supposed be. The reason is even they are separate visually, they are contained as subloops in one big loop. The algorithm determines subloops by their minimum and maximum X and Y instead of shapes. Simply described in Figure 22. The algorithm has to be improved. Nevertheless, the current method optimizes some scenarios, which is better than nothing.

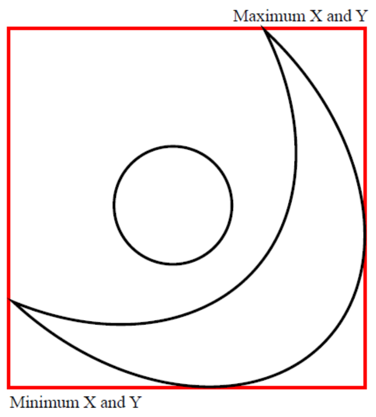


Figure 22: Example for algorithm's defect

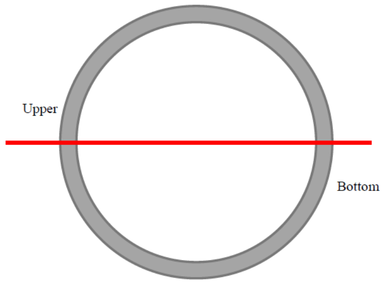


Figure 23: Big loop with only one subloop

On the other hand, the subloops within big loop, are still handled the traditional way, shown as the right area in Figure 21. The inside small loops are empty. The best result is without using G0 commands for these empty areas, only go through where has to be extruded.

For one case, if the loop only contains one subloop, shown in Figure 23. When generating gcode, it can use a stack to store the G1 commands for the bottom part. Then pop them after upper commands. However, when loop contains more subloops, it would turn to be more complex.

Also, the extrude value is computed using distance times a factor in this slicer. The exact amount of extrusion is critical for the quality of the resulting part and is still being studied.

10. Future work

The BIF format replacing STL files has to be tested. Further improvements are possible. A senior design team will work on it based on the current work. Also, to fill loops, more patterns and different angle options should be implemented.

The Hydra project began as an exploration of using Multi-headed printer to speed up the printing process. This work will continue.

Some models require support material to help to hold the object, in place which it is being printed. The senior design team is extending that work.

XbSlicer demonstrates how much current slicers can be improved. We hope that it will become an option for improved opensource 3D printing.

11. Acknowledgements

The author would like to thank Prof. Min Song from Department of Electrical & Computer Engineering at Stevens Institute of Technology for supporting the working space on which this research is been done. Thank Courtney Walsh, Research & Instructional Services Librarian at Stevens Institute of Technology for her citation help. Also, thank Writing & Communications Center (WCC) at Stevens Institute of Technology for their writing advice.

12. References

[1] Hu, J. (2017). Study on stl-based slicing process for 3d printing. Paper presented at the 2017 Annual international solid freeform fabrication symposium on University of Texas at Austin, 885-895.

[2] Schurig, F. (2015). Slicing Algorithms for 3D-Printing. Retrieved from https://blog.hk-fs.de/wp-content/uploads/2015/08/Paper_Fabian_Schurig_3D_Printing.pdf

[3] Choi, S., & Kwok, K. (1999, August). A memory efficient slicing algorithm for large stl files. Paper presented at the 1999 Annual international solid freeform fabrication symposium on University of Texas at Austin, 155-162.

[4] Choi, S., & Kwok, K. (2002). Hierarchical slice contours for layered-manufacturing. Computers in Industry, 48(3), 219-239.

[5] Minetto, R., Volpato, N., Stolfi, J., Gregori, R., & Da Silva, M. (2017). An optimal algorithm for 3d triangle mesh slicing. Computer-Aided Design, 92.

[6] XbSlicer GitHub repository: https://github.com/XinqiBao/XbSlicer