Documentation file for the custom slicer, built as part of the GCode project for the Interact lab.

Author: Panagiotis Florous

Email: [panagiotisflorous@gmail.com](mailto:panagiotisflorous@gmail.com)

# Summary

This is a short text describing the logic behind the custom slicer I wrote. The goal is to explain the reasoning behind the tech I used for many of the optimizations, in order for anyone else to expand on or modify what I did.

The program is not a traditional slicer by any means, and cannot be used for 3D printing. It is a completely custom slicer who’s main goal is to make volumetric rendering in an acoustic levitation environment as easy for the user as possible. The slicer only works for non-compound, closed meshes, with the exception that the opening in the mesh is parallel to the slicing plane.

The program consists of 2 classes with major functionality, 2 helper classes for representing points and faces, and a struct to represent the slicing plane.

# Classes

## Plane

This class describes a plane with a vector normal and a distance variable called “direction” because I was confused when I first wrote that. It also include helper methods related to the Plane, which is used for slicing. In general for a normal vector vx, vy, vz and “direction” d, a point x,y,z is on that plane when vx\*x + vy\*y + vz\*z = d.

### Constructor

Makes the vector and calculates the d from 3 points.

### intersectionPoint

Takes to point and finds where this line described by the two points intersects with the plane. Note that it will give a result as long as the line and the plane are not parallel so for slicing applications the check needs to be done in the slicer that the two points are in opposite sides of the Plane.

### pointAbovePlane

Help method to achieve what I said above. It returns -1 for below 1 for above and 0 for a point on the plane. Note that this will almost never return 0.

### getFaceWeight

Another helper method, it gets a Face as described below and returns the “weight” of its 3 points.

As weight I defined the number that would come up if we were to add the pointAbovePlane values of the 3 points. In first sight this might seem stupid but in reality it is incredibly useful since it can help us eliminate faces really quickly from computation. Below I list what each weight represents.

* 3 : All 3 points above the plane
* -3: All 3 points below the plane
* 2: 2 points above the plane one on it
* -2: 2 points below the plane one on it
* 1: This means either 1 point above and 2 on the plane or 2 above and 1 below
* -1: This means either 1 point below and 2 on the plane or 2 below and 1 above
* 0: This means either that all points are on the plane or 1 above, 1 on, and one below

Just to explain further. As you can see the smaller the number the more the possibilities but it can help you eliminate from computation really fast, triangles that have a 3,-3,2,-2 as weight and continue with more useful things with our CPU time.

## Point

The Point class is a helper class. It only stores a glm::vec3 for the 3D coordinates of the point as well as a float that represents the distance of the point to the lowest bounding plane of the model. This distance is stored for optimization purposes discussed later.

Apart from that it only contains setters and getters.

## Face

The Face class is also a helper class. It stores a glm::vec3 for the 3 point’s indices describing this face, and a vector of pointers to these point’s distances, described above. If the glm::vec3 description does not make sense I suggest checking the Wavefront .OBJ documentation to see how the faces function there as it is the same here.

Apart from that it only contains getters.

## OBJReader

The OBJReader class is the first with major functionality. This class’s purpose is to read and store the OBJ file in a format that we are able to work with in our program.

### Constructors

The class has two constructors. One is a simple read and store while the other is a delegate constructor calling our read/store one first and then some functions. The second constructor is there to save time in slicing if we know our slicing plane on read-time.

### faceCleanup

Not much to say here. Simply strips the normal and textures from the faces. If unsure what that means again check Wavefront .OBJ docs.

### minMaxCalc

This function takes a Plane object and finds according to the function A\*x + B\*y + C\*z = d what is the lowest and highest “d” for our model. This is done to find the “bounding planes” of our model with that specific plane’s normals. It also saves all “d” in their respective points, and lastly sorts our faces according to their point with the lowest “d”.

To do so we use as A B C the slicing plane’s normal and as x y z our point’s coordinates. We iterate through every point so that’s a linear operation and potentially time consuming.

Despite that, it is a necessary procedure, and by saving the distances in the points we save much more time during slicing than what we lose during reading.

### Getters/Setters

Lastly we only have getters for this class. Any modifications are done internally ONLY, through method calls. This is crucial as to preserve our data to avoid corrupt models.

## OBJSlicer

The OBJSlicer class is the second with major functionality. This class’s purpose is to slice the model, find optimal travel speeds, handle transitions between layers, and generate the GCode. This class has a lot built-in, with many optimizations throughout so I will explain the logic as well as have some diagrams to guide you.

### Constructors

The slicer can be called with or without an OBJReader object however it makes sure it always ends up with one. The constructors just initialize all the things that are necessary before slicing. Note that I have not implemented a constructor that utilizes the secondary OBJReader constructor’s optimization. This was because when testing I could afford wasting a little time, since the optimization is not too big and possibly obsolete due to other parts of the project. I will do it if we go back to the project, if it’s not there you can do it it’s a simple bool value and an if statement.

### makePath

This method is the core to our slicer, the “main method” if you wish.

We start by fetching the Face vector from our Reader, casting it into a list, and reversing its order. This is done to save time later.

We fetch our low and high bounding planes and divide their difference by the number of needed slices + 2 to get the “layer thickness” necessary. We do +2 to avoid situations where we have no paths but stray points in the first and last slices.

We start looping for every slice we need. Here we find our first optimization, reducing the search space. Searching linearly every Face for every slice is an n^2 operation (m\*n to be exact) and this is fine for low poly models, but gives us insane times in high poly ones. To search as little Faces as possible we utilize the fact that we sorted our Faces.

The logic goes like this:

* Any face which has its lowest point on or above the slicing plane is not sliced by that plane.
* Any face which has its highest point below the slicing plane is not sliced by that plane **and every plane following.**
* We have the list of Faces is **descending** order (we flipped the list before).
* Using lower\_bound (and a custom comparator which we’ll see later) we find the first triangle which has its lowest point on or above the slicing plane with binary search, saving us time.
* We then check for intersections for all triangles further in the least.
* If a triangle does not intersect we erase this triangle from the list.

This way we have log(n) time for the search, and with each layer we drastically reduce the number of faces that we account for. To stress how important this optimization is compared to linear search, in an incredibly high poly model we went from reading and slicing in **11 minutes** down to **1 minute**, with the reading taking up most of the time.

Now, to check the above cases we utilize the getFaceWeight function(look Plane for more information). For each intersecting Face we call the getStep method described below.

For each layer, I call the generateGCode and generate Curvature functions and I clear the path I create as well as the velocity vector.

Finally I iterate the lowBound by one step and update the last point in the path for the next layer to know where to begin.

### getStep

The getStep method creates one line in our path given a Face that intersects with the slicing plane and the index of the last point in the previous layer. We need that index for another search space optimization.

Every case of triangle intersection except for one (discussed in Plane class) has two points of one side of the plane and one on the other. We want to find the “lone” Point to help us preserve the direction of our path.

We start by doing that with a while loop until the “order” vector has the “lone” Point’s index in position 0.

We check whether or not the “lone” vector is above or below the plane and we call the intersectionPoint method of Plane with respect to direction (above to below or vice versa).

Since each layer consists of connected lines generated through the intersections between a face and a plane, if we were to simply find all the intersections we would end up with duplicates. In order to account for that we need to check that the intersection point that we found is not already present due to another getStep call. Since we only care for this layer, the index we have as input serves the purpose of limiting the search to the current slice.

### generateGCode

This method simply makes a GCode-like file so we can use it to “play back” the slice that we created.

It iterates the path for the layer that we created, until it reaches the starting point again, and if there is more left to the path it starts iterating over the leftovers to find more closed paths in the layer. The reason this is needed is for situations where we have shapes like feet, or torso and arms in the same layer. These cases create multiple closed paths in each layer.

The way we traverse through the paths is by taking advantage of the pairs we made before. Each pair in the path map shows the start and the finish of the step in the path. This way by iterating through the map we created the sorted path to traverse the layer.

### generateCurvature

This method functions the exact same way as generateGCode with the difference that it takes 3 points in the path that have distance between them and passes these 3 points in the radius function described below.

We take tha value and save it as the curvature of the first point, thus having a map of the curvature shift overtime. We need this to calculate the optimal speed that the bead has to travel between each of the points.

### generateOBJ

Possibly broken method that was written to help visualize the sliced model in OpenGL UI. The method itself works but might not for specific cases. Generally not well tested and quickly put together. It follows similar logic to generateGCode and generateCurvature for traversing the path.

### Getters

Three simple getters for the pathLenght, Gcode, and the OBJ generated path.

# Conclusion

That was my approach for the Slicer, it is relatively fast at slicing but it could definitely use some work when reading an OBJ in a format the program can work with.

This “doc”-like file is written in 2019. If you have any further questions at the time reading don’t hesitate to contact me to resolve any questions.

Best of luck!