# Scroll Sheet Adjustment and Assembly

Santiago Pelufo

June 2024

Last year I attempted automatic segmentation with a naive method of separating papyrus from background and meshing the holes between the papyrus with marching cubes. Given how much the different turns of the scroll come in contact with each other, this method doesn't yield good results. Nevertheless, I was able to observe that where it did trace part of a sheet's surface, it did so more closely than the official segments. The manual segmentation of part of Scroll 1 in 2023 was a big effort and area coverage was more important than extreme precision. I also saw that many of the fragmentary pieces of surface my method had produced had crackle on them. I concluded that crackle, our main known signal for ink, is localized to the front surface of the sheets.

With the current accuracy of segmentation and the depth of the surface volumes presented to ink detection models, it is possible that they will pick up ink signal from adjacent sheets. To achieve more and better ink detection we might need segmentation that is not only automatic but also more accurate.

So this year I set out to see if I could make a program to improve the Scroll 1 2023 Grand Prize (GP) segments, to validate this hypothesis and generate better training data for autosegmentation models. I have developed a pipeline to do this, described in this document and available at github.com/spelufo/stabia which I submit as a Monthly Progress Prize together with the adjusted segment meshes.

Part of the pipeline is a procedure for sheet assembly from chunks, which is of interest for automatic segmentation. The sheet chunk assembly method presented here works to reassemble chunks from the GP banner segments into a single coherent scroll section covering the entire banner with relatively little manual post-processing. More work is required to see how the method does on sheet chunks from other sources.

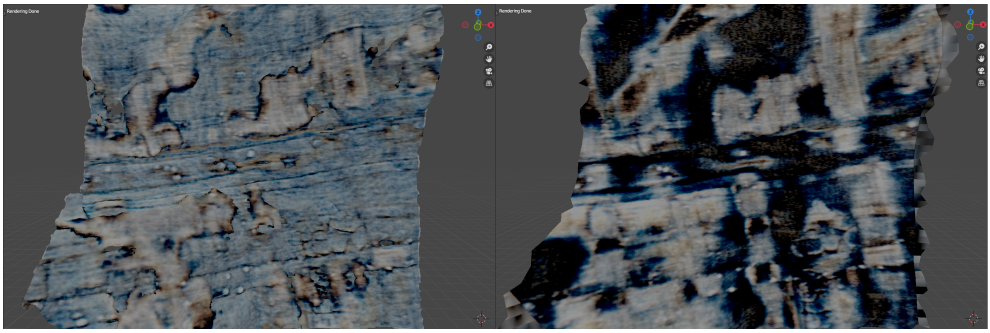

Figure 1: Segmentation surface comparison between a mesh resulting from the 2023 "holes" method (left) and the corresponding piece from an official segment (right). Ink crackle can be seen on the lower right part of both.

# 1 Sheet Adjustment

The adjustment procedure works on a cell by cell basis, processing each of the 754 grid cells of size $500^3$ from the volume grids dataset that cover the GP banner meshes separately. The segment meshes are split on the planes that divide the grid cells to form a set of sheet chunks for each cell.

The SNIC Superpixel algorithm is used to generate an over-segmentation of the volume. The superpixel size is chosen such that a superpixel will usually span the depth of a single sheet of papyrus. To adjust the segment chunks to the surface of the sheets we will assign to each superpixel the segment chunk that overlaps it the most. Hence volumetric instance labels can be generated as a side effect.

The segments can overlap so there can be multiple chunks for the same sheet turn. Duplicates are found and one of the pair is removed. The criteria used is the amount of overlap between the chunks, with a threshold deciding if two chunks are from the same sheet or not. This works well in most cases, but some instances of duplicates make it through the pipeline, and it sometimes causes legitimate chunks that happen to overlap a lot to be removed, leaving holes. Sheet assembly must be robust to this.

The sheet chunks are also sorted by scroll turn number (a.k.a. winding number), the one closest to the core first, which is not trivial. An undirected weighted graph is formed where there is a node for each sheet chunk and an edge represents evidence for a sheet chunk being adjacent to another, with its weight being a measure of how likely it is that the chunks are consecutive. We must turn this graph into a line (or a set of line segments), since each chunk segment is from a different scroll turn and must only

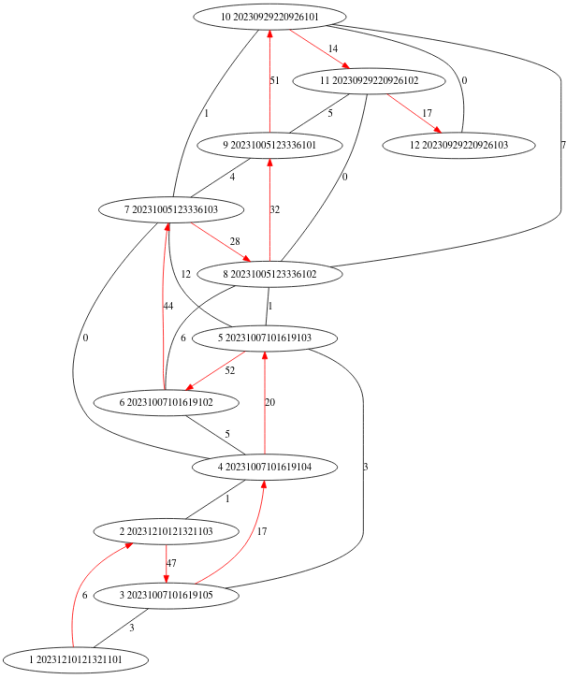


Figure 2: A sheet chunk graph. The red edges show the result sequence.

be adjacent to adjacent turns of the scroll.

In practice, this graph has arbitrary shape (e.g. Figure 2). But the weights between chunks from consecutive turns are markedly bigger than the ones from segmentation or processing artifacts, and we can find the sequence of chunks for each connected component of the graph by finding the longest (highest weight) path between the nodes.

After this it only remains to decide if the linear order found should be reversed or not. This is done with a heuristic using the position of the scroll core. Note that cells that overlap the scroll core can have chunks facing each other. The heuristic doesn't work in that case, but these cells are a minority.

Finally, the output of sheet adjustment is a point cloud. For each superpixel of each sheet chunk we use the mesh normals to cast a ray from the superpixel center to the superpixel boundary, and generate a point there if it didn't hit the back of another superpixel of the chunk. This way the point cloud for each sheet chunk traces the front

surface of the papyrus.

## 2 Sheet Chunk Assembly

To assemble the sheet chunks into a single surface we must "glue them" together at the cell boundaries where we have cut them before. The proximity and amount of contact between sheets makes doing this robustly challenging. Duplicates and missing sheets must be dealt with.

I define a graph with two relations on the chunks (two kinds of edges). One kind, called the *same* edges, are those between chunks thought to be of the same sheet turn. They are undirected. The other kind, called the *less* edges, are directed and they imply the source chunk comes before in the sheet turn sequence, its turn number is less (and usually the difference is 1). These come from the order deduced during sheet chunk adjustment within a cell. A few *same* edges are turned into *less* edges when they cross the turn change boundary, which is aligned to grid cell boundaries.

To build the *same* edges we run a chunk matching procedure on each square boundary between cells. It measures how much pairs of chunks line up and uses dynamic programming to find the best set of pairs.

The goal of assembly when presented in these terms is to assign a scroll turn to each chunk. Because the edges are estimates of the true relationship between the chunks, some of them are incorrect. The graph is not acyclic and there isn't a unique layering of it. The objective is to find a subset of the graph that is, and a layering consistent with the real turn numbers. [1]

Assuming most of the edges are correct, we'd like to break as few as possible conflicting edges to make the graph acyclic. This is known as the Minimum Feedback Arc Set problem and it is NP-hard. However, we are content with approximate solutions if they explain the data.

The graph Thaumato Anakalyptor uses corresponds to the *same* relationship used here. In this way our graph is a generalization of Thaumato's. The rationale for the *less* edges is that they should help make the conflicts (cycles) shorter, more localized. It is also clear to me that when interpreting papyrus cross sections visually we infer where sheets continue from an unclear area by looking at the neighboring sheets past

[1] Neither is the subgraph we seek, strictly speaking. In this whole discussion we gloss over the fact that "same" undirected edges are encoded as a bidirectional pair of a directed graph, and by cycles we really only mean those cycles that have at least one "less" edge, which are the ones that generate inconsistencies.

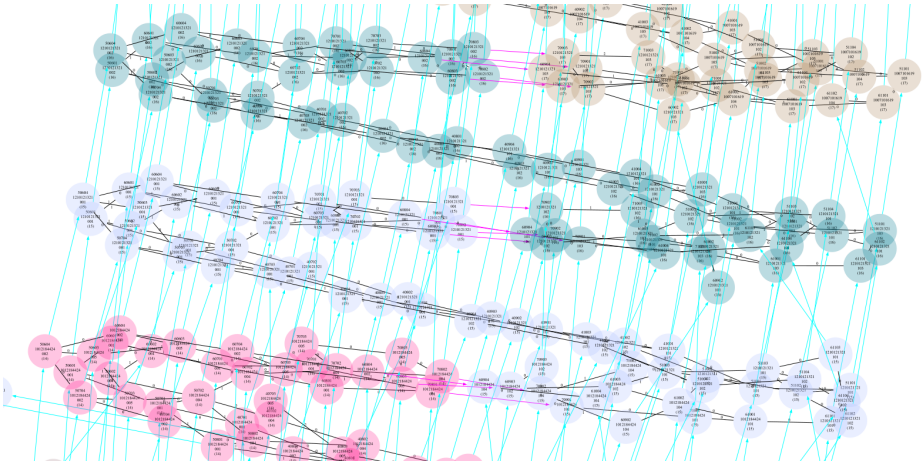

Figure 3: A section of an assembly graph, with different turns in different colors. *Same* edges are black. *Less* edges in cyan or magenta (when crossing the turn boundary).

it.

My first approach to cycle resolution was to try the simplest thing that came to mind: break all the edges on every cycle and hope that there's enough redundancy in the graph so that what remains is enough to number the sheets. It does work to an extent, but not as well as the current solution, requiring a lot more manual post-processing work because it breaks many good edges and can disconnect parts of the graph.

Regardless of how we break cycles, finding all of them isn't easy. Johnson's algorithm has the best algorithmic complexity, but its memory usage quickly becomes prohibitive. With the insight that most cycles must be caused by a few mistaken edges and that the cycles they will produce should be local we use a DFS based procedure to search for cycles up to length 12. This algorithm's runtime grows exponentially as the maximum cycle length is increased. This means the effectiveness of the procedure is very dependent on the structure of the graph. If it has long cycles that are not resolved by breaking shorter ones it is unlikely to succeed.

When using lower cycle length limits one can see that some turns of the scrolls fail to be separated from each other. The size of the graph is also a factor, so for more challenging graphs one can envision applying the same logic at different scales, perhaps recursively.

Still, for the graph of chunk segments from the GP banner it works well and finds a solution separating all the sheet turns in a reasonable amount of time. The whole

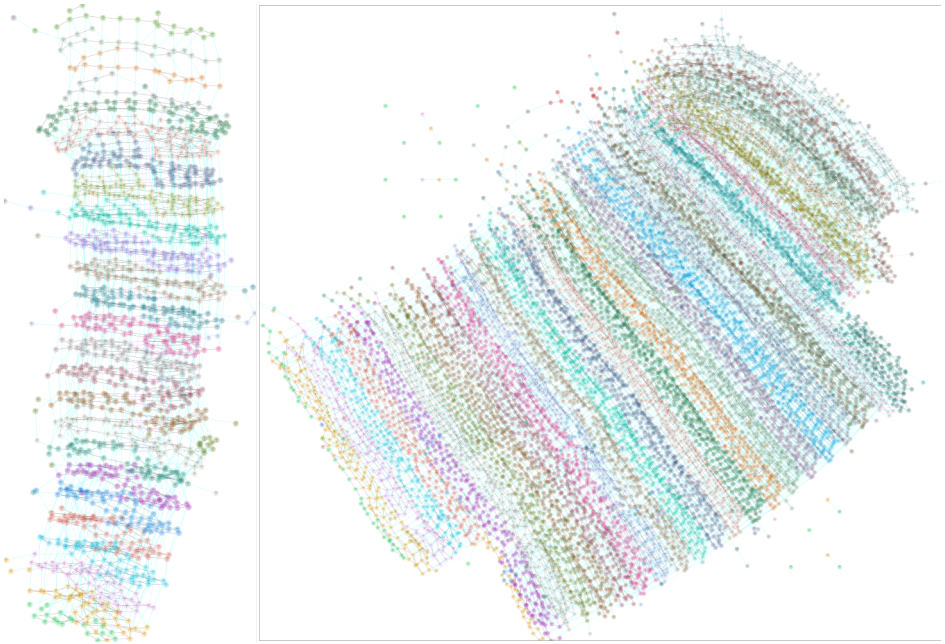

Figure 4: Two assembly graphs. Left: The graph for the "jz" range of cell layers 5 through 9. Right: The solved assembly graph with all the GP chunks.

assembler procedure takes about 12 minutes.

I saw a marked improvement in the results when I chose a different cycle breaking strategy. Given the expected topology of the graph a single bad edge will generate a series of progressively larger cycles including the edge. Hence if we keep a counter for each edge accumulating the number of cycles that pass through it, those edges that have the highest counts must be the root of the problem, having induced all the cycles that "voted" for it.

Taking this into account, we find all cycles up to a given length and put all of their edges into a priority queue where the priority is the number of cycles an edge belongs too. We will break the most troublesome edges first. Every time we do, we decrease the priority of every edge in a cycle containing the edge we just broke. If an edge's priority becomes zero all the conflicts it was a part of have been resolved. I don't think removing edges in this manner guarantees making the graph the acyclic in one pass (for the GP chunk graph one pass is enough), so the procedure is repeated until no cycles are found or a maximum number of repetitions is reached.

The same approach of breaking cycles and DAG layering is used in the Sugiyama

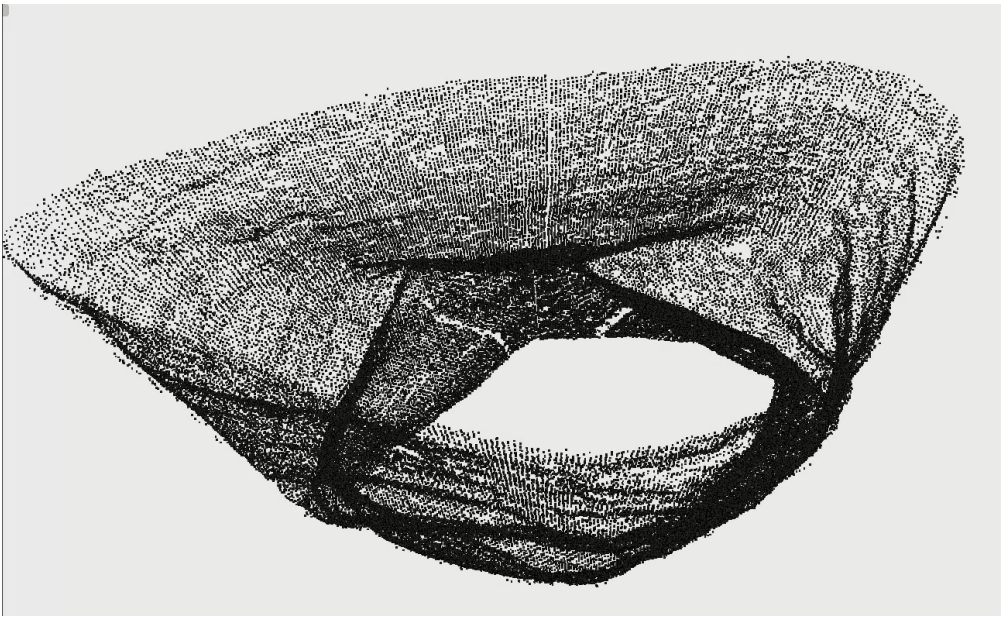

Figure 5: The assembled point cloud for (part of) a sheet turn.

framework for graph drawing, employed by Graphviz *dot*, for instance. Contrary to us, Graphviz doesn't care too much about which or how many edges it has to break to get an acyclic graph. Layering is achieved by solving an integer linear program to find the y positions for the nodes that minimize the total edge length and satisfies the partial order constraints of the DAG. Once we have resolved all conflicts we can use the same approach. An ILP solver from JuMP or another optmization library can be used to the same effect. The current implementation shells out to Graphviz.

We don't layer the assembly graph, but its condensation. In it each strongly connected component of the assembly graph becomes a node. All the *same* edges are collapsed in this transformation, leaving a DAG with only *less* edges, where each node corresponds to a group of chunks known to have the same turn number.

Visualization with another Graphviz program, neato, which draws graphs modeling spring forces between connected nodes, was very helpful during development. I found that with the right initialization, the layout assumes the form of a spiral (Figure 4). I used color to show the different kinds of edges, including the broken ones (in red), and this way I checked that the procedure did what I expected. A more faithful assessment of the results is how the point clouds for each turn look like after assembly (Figure 5), but this visualizations enabled a shorter feedback loop.

The resulting assembly is output as a python script to import all the chunk point clouds into Blender into separate collections, one per turn. Most of the chunks are correctly separated into turns, but some can be misplaced. There can be spurious turns with a few chunks that need to be moved to adjacent layers or deleted. Swaps in the relative turn order of chunks are very rare. Most issues come from the overlapping areas between segments, and a failure in sheet adjustment to remove duplicates. The top of the scroll is specially problematic because where the scroll ends the segmentation team had to trace the sheet on the xy plane regardless, and there's one segment cutting across the others there. Another source of problems can be cells where the guessed order of the chunks is incorrect, usually near the core. These create long feedback paths on the graphs. Once the root cause of this issue was identified, it is easy to find these cells using the graph visualization and add them to a list of reversed cells for the procedure to fix. There's also a list of cells (just three) that are skipped from inclusion in the graph. Doing this before moving on to post-processing cut the time it took in half.

It takes about two hours of focused manual work in Blender to finish the job by visual inspection and regrouping of chunk point clouds amongst the Blender collections. One works sequentially, finishing each turn of the scroll before moving on to the next. A small window of adjacent collections is considered for sourcing missing chunks to complete a turn. When a spurious collection is found, is contains few loose chunks that must be deleted or moved to adjacent collections. Those chunks that are very out of place are put aside for a final pass attempting to find their turn. I won't go into more detail about it here. I have video of it and can teach others how to go about it, but for the GP banner chunks it is already done. When finished, a custom export command is run to dump the assembly to a json file with references to the chunks that comprise each turn of the scroll.

# 3 Meshing

At this point we have point clouds for each turn of the scroll. Next we need to create meshes from them. This is done using Poisson Surface Reconstruction [Kazhdan, Bolitho, and Hoppe, 2006], using Khazdan's implementation. This part of the pipeline is the most recent and there is room for improvement.

A few obstacles need to be overcome to make Poisson reconstruction effective for our purposes. Reconstructing the whole scroll at is not possible, the turns wouldn't be separated from each other. At most we can reconstruct a single turn, which will generate a closed surface with a seam at the turn boundary. This seam distorts the

surface nearby, and we would need to cut it. Instead, we reconstruct each half turn separately and stitch consecutive half turns later. To get good matching between half turns at the boundaries between them it helps to reconstruct a full turn with the seam opposite to the half turn and discard half of it. In this way the seam isn't part of the result but the points near the boundary are well supported, and the surface closes in on itself, minimizing the overhang that is otherwise created. Removing this overhang is necessary but can unfortunately lead to meshes that have holes and are non-manifold. This is a problem for flattening (i.e. UV mapping).

Mesh cleanup and stitching of half turns into bigger sections of the scroll is done in Blender, mostly automated as python scripts in the vesuvius-blender add-on. Stitching uses a merge by distance operation on vertices in the half turn boundary. This too can generate non-manifold vertices. The problem is ameliorated by having written julia bindings to Khazdan's Poisson Reconstruction code to enable us to control the alignment of the grid that reconstruction takes place in. With this in place, most vertices in the reconstruction are aligned to this grid and match closely across the boundary.

Note that stitching isn't strictly necessary to move on to flattening and ink detection, but the half turns near the core don't span a full column of text. We would like to build a single mesh for the whole banner. I think this is possible with these methods, but due to hardware limitations for the moment I've settled for reconstructing smaller sections of the banner, stitching only as many half turns as I can before they become too large to manipulate on my machine.

I have procedures to do mesh cleanup by deleting non-manifold vertices and filling holes that enable me to get a manifold mesh. Remeshing with ACVD after stitching also helps solve non-manifold issues, and is done to reduce the triangle count and produce a mesh that is easier to flatten. Stitching also results in meshes with positive genus in a few cases. This is currently fixed by hand in Blender. Attempting UV mapping will result in a map folding in on itself, and the vertices that cause the mesh to have a non-zero genus are to be found near where the vertices are most compressed together on the UV map. It can take a little while to hunt them down. This needs improvement. At the very least we can stop stitching immediately after a non-zero genus is detected to make the problems easier to find.

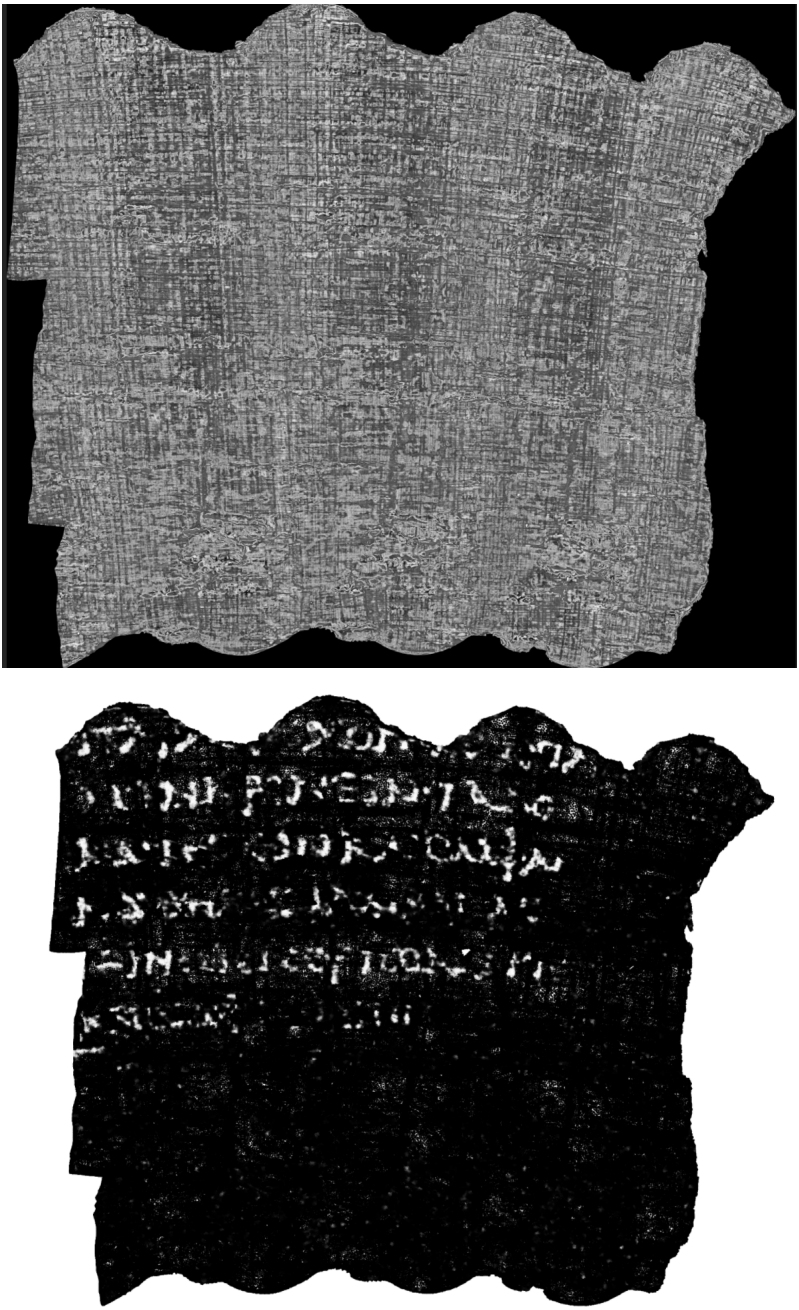

Figure 6: The central layer of the surface volume for the last column of the GP banner. Preliminary ink detection using Giorgio Angelotti's script to render Ryan Chesler's 3d ink detection.

# 4 Conclusion

Visual inspection of the surface volumes rendered from this adjusted meshes show encouraging signs. The fibers are more clearly aligned and crackle is evident in many places. Further development is needed to see how this impacts ink detection.

This document has described the methods and challenges encountered in the process of sheet adjustment. Several of these are shared by automatic segmentation. I hope this account constitutes progress on this front as well as providing this new dataset to train instance segmentation models.