Refinement of an automated segmentation pipeline

William Stevens

30th September 2024

Refinement of an automated segmentation pipeline

This report describes progress on an automated segmentation pipeline described previously [1]. The project uses basic image processing operations and algorithms based on flood fill to find distinct surfaces and surface patches in CT scans of the Herculaneum Papyri. The pipeline consists of distinct programs for initial processing of $512 \times 512 \times 512$ voxel cubic volumes, identification of non-intersecting surfaces and surfaces patches, joining together neighbouring patches without overlap, and rendering to see the results. Several utility programs and some third party tools are used for debugging the pipeline. Progress since last month: the behaviour of the DAFF algorithm has been studied and the algorithm has been improved, the program for piecing together surface patches has been rewritten, experiments with different ways of identifying surfaces have been carried out. The pipeline has been run on an AWS EC2 server so that it can be run at scale when it reaches the point when it is likely to produce large surface areas that can be fed into 3D ink detection pipelines.

Introduction

Reading the Herculaneum Papyri presents many challenges because the scrolls are badly damaged. One of the challenges is to efficiently and correctly identify the surfaces of scrolls and establish which layer of the scroll the surfaces belong to. Manually delineating surfaces with the help of automated tools has made some progress. This is time consuming, so fully automated pipelines that can unwrap scrolls are also being developed. This report describes monthly progress on a pipeline for automatically delineating surfaces.

Improvements to an algorithm for finding intersection free surface patches

In last month's report I described two ways of adapting flood-fill to avoid filling intersecting surfaces in the same colour. Over the last month I have been improving one of these: damage-avoiding flood-fill (DAFF).

The DAFF algorithm is described below:

The algorithm is given two orthogonal vectors $\mathbf{u}$ and $\mathbf{v}$ describing a 2D plane. This plane should be roughly parallel to surfaces in the $512 \times 512 \times 512$ volume being processed. A normal $\mathbf{w}$ to these vectors is calculated. The orientation of $\mathbf{u}$ and $\mathbf{v}$, and the direction of the resulting normal do not matter, the only important thing is that $\mathbf{u}$ and $\mathbf{v}$ are orthogonal.

1. Import intersecting surfaces to be segmented into a working array representing the $512 \times 512 \times 512$ volume.
2. Pick a random surface voxel in the working array and begin flood-filling from there.
3. As flood-fill progresses, project each filled voxel (x, y, z) to a point (x', y') on the

plane represented by $\mathbf{u}, \mathbf{v}$, and work out its perpendicular distance z' from the plane by calculating $x' = (x, y, z).\mathbf{u}$, $y' = (x, y, z).\mathbf{v}$ and $z' = (x, y, z).\mathbf{w}$ respectively.

4. If a voxel has already been projected to (x', y') and the absolute difference of the perpendicular distance of that projection and z' was greater than 5, then intersection has occured, goto step 5. Else continue flood filling. If flood filling terminates with no intersection, then export the filled voxels and delete the voxels from the working array.

5. Reset flood filling to the unfilled state, and then start flood filling again from the point of intersection, until intersection occurs again, and repeat this until the distance between successive intersections stops decreasing.

6. Flood fill again from the last intersection encountered, counting the distance from this voxel until we reach half the distance between the last two intersections acheived in the last iteration of step 5. The first voxel encountered at this distance is labelled as a 'plug' and is considered unfillable (it is deleted from the working array).

7. So long as there are still surface voxels in the working array, goto step 2.

The result is a set of surfaces patches exported from step 4, where each patch has no intersections, along with a list of plugs exported from step 6.

Figure 1 shows surface voxels in a $100 \times 100 \times 100$ voxel volume. The result of running the DAFF algorithm on this is shown in figure 2.

During the past month the behaviour of this algorithm has been studied, and the algorithm has been improved and made more efficient (figure 3 shows one of the test shapes that

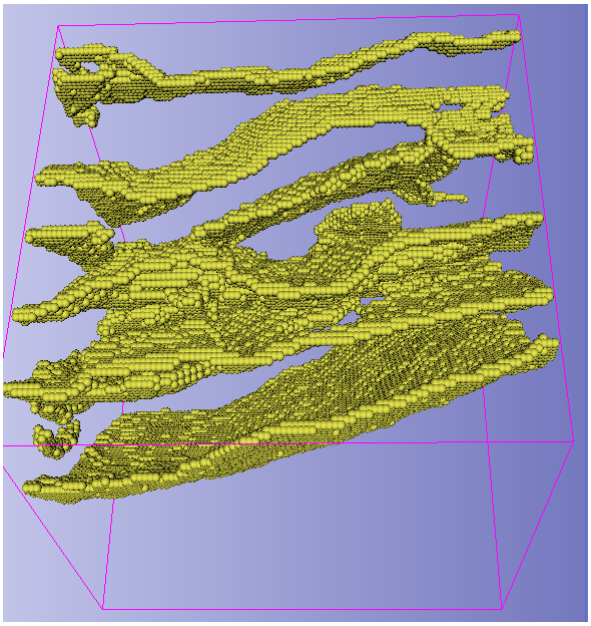

Figure 1: Surface voxels in a $100 \times 100 \times 100$ voxel volume.

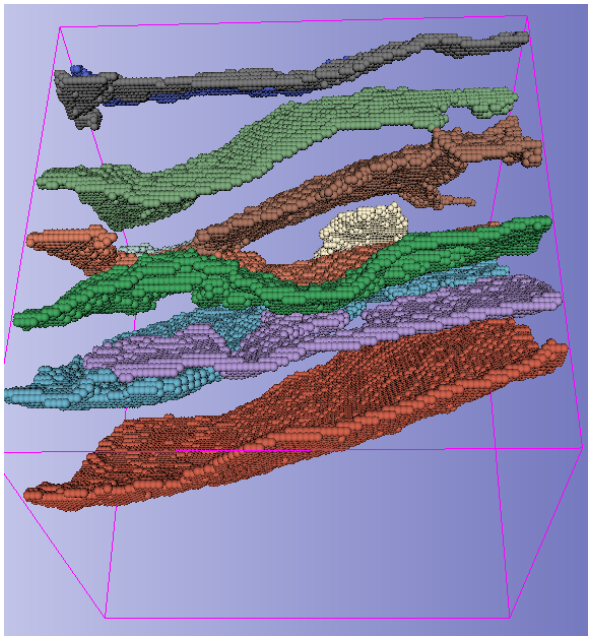

Figure 2: Result of applying the DAFF algorithm to the volume shown in figure 1. The top and bottom surfaces were already distinct from the others, but the middle surfaces were all connected, and DAFF is able to separate them. Sometimes it splits a single surface into more than one patch.

was used during debugging and refinement). The previous version of this algorithm only projected onto the x, z plane, so would only work with $512 \times 512 \times 512$ volumes laying approximatly in the $+y$ direction from the scroll umbilicus. It will now project onto any specified plane, so will work anywhere away from the centre of the scroll. Currently the projection plane has to be specified, but in future this will be computed automatically. Getting it to work near the centre of the scroll could be done in a number of ways - one way would be to reduce the working volume size near the centre.

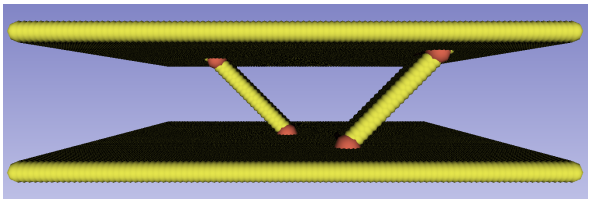

Figure 3: One of the test shapes used for helping to debug and refine the DAFF algorithm. The red voxels shows where DAFF puts plugs to separate the four resulting segments

Step 5 of the algorithm has been improved as a result of exploring the behaviour of the algorithm. In the previous version, step 5 sought an intersection, then sought the next intersection, then looked for the midpoint between the two. Step 5 now keeps looking for another intersection until the distance between intersections stops decreasing. Looking at the log output from the DAFF algorithm shows that usually this happens after 3 or 4 iterations of step 5, but not infrequently it happens after more than 4 iterations, and the largest number of iterations seen so far is 9. The effect of this improvement in the algorithm seems to be that it increases the chance that plugs are located in what look like sensible places - i.e. near places where two surfaces meet - and reduces spurious plugs that have no effect. This makes the surface patches that it produces larger.

The other main improvement was in step 4. In the previous version intersection-free volumes were not removed from the working array in step 4, and termination was based on failure to find intersection after 1000 randomly chosen points - this could leave some small surface areas unexplored. Now that intersection-free volumes are deleted from the working array when they are found, the algorithm speeds up as it progresses and is guaranteed to terminate once all surface voxels have been processed. When this change was implemented it doubled the performance of the algorithm.

Other work this month

Stitching together in-volume surface patches

The algorithm for stitching together surface patches has been improved, largely through a process of trial-and-error. It now works as follows:

1. Initialise an empty working image.
2. Begin iterating i through each unused patch (ordered from large to small).
3. If the working image is empty or if i abuts the working image without intersection, and the length of the abutting surface is more than $\sqrt{|i|}/2$ then add i to the working image and mark i as used.
4. Move onto the next i and goto step 3. (Or fall through to step 5 if no more i).
5. Export the working image.
6. If there are still unused patches then goto step 1.

Surface voxel identification

Image processing to identify surface voxels is currently done as follows:

1. Split the scroll images into $512 \times 512 \times 512$ cubic volumes. Only the most signifcant 8-bits of the greyscale value from the original tiff images are preserved.

2. A threshhold operation is carried out to identify the papyrus areas. Voxels with a greyscale value ≤ 135 are empty and > 135 are scroll. Some papyrus may be missed (but will be regained in step 3), but most noise in voids is rejected.

3. Three dilation operations with a threshhold of ≥ 110 are performed to regain papyrus areas missed in step 2.

4. Separately from this, a 3D Sobel edge detection filter is applied to the 8-bit greyscale volume. Voxels where the magnitude of the Sobel vector is > 150 and where the Sobel vector points no more than 90 degrees away from the direction towards the scroll umbilicus are retained.

5. Surface voxels are those identified by both step 3 and step 4 that also have at least one empty neighbour in a 6-neighbourhood.

This is only a small change since last month. (steps 3 uses fewer dilation operations, in step 4 the Sobel edge detection threshhold is lower, and in step 5 a 6-neighbourhood is used rather than a 26-neighbourhood). Figure 4 shows the surfaces that are identified using the steps described above.

Several other possible surface point identification methods were tried, including using a Laplace filter in conjunction with a Sobel filter to detect when the second derivate of the signal was zero, and also using 3D adaptive histogram equalization prior to the pipeline above. Both of these attempts produced more surface points, but also more spurious points and rougher surfaces due to noise - this causes the DAFF algorithm to split the surfaces into much smaller patches, which take longer to process and don't necessarily join together well

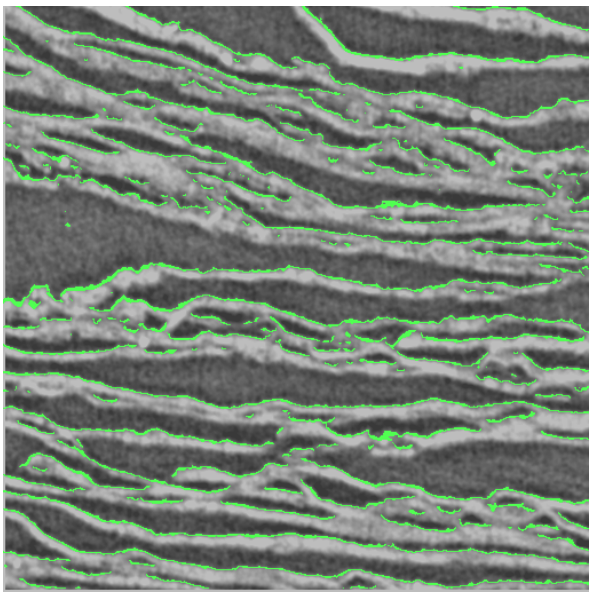

Figure 4: Extracting the surface signal from a single $512 \times 512 \times 512$ volume

(possibly because very small areas are not used, so it is like having a jigsaw with some criticl missing pieces). Both methods show promise and will be investigated further to see whether the surfaces can be smoothed, but I reverted to the steps listed above prior to writing this report due to lack of time.

Example output

Figure 5 shows surfaces obtained from a single $512 \times 512 \times 512$ volume of scroll 1 located at $x = 2888, y = 4200, z = 5800$. In this figure there are about 12 surfaces spanning more than about half of the render area, albeit with some holes. Not shown in this figure are 235 smaller surface patches, effectively leftover jigsaw puzzle pieces. Working out what can be done to place these pieces is a remaining challenge. Some of these pieces belong together on the same surface, but non-local information is needed to be able to infer this.

Figure 6 is a close-up of one of these surfaces.

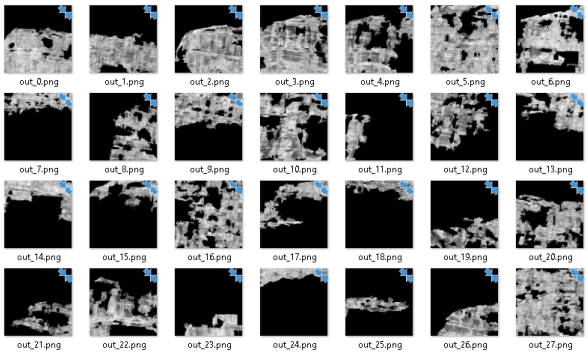

Figure 5: Some surfaces output by the pipeline for a single $512 \times 512 \times 512$ volume

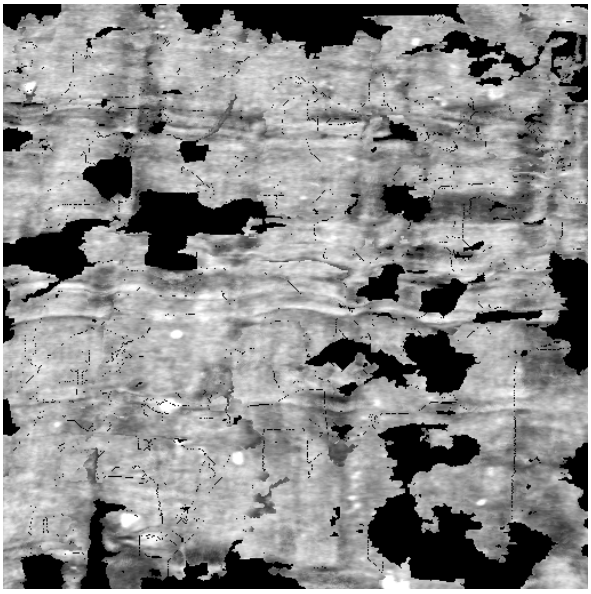

Figure 6: A single surface from figure 5

Performance

On modest hardware (A single core of a 1.1 GHz Pentium N4200 with 4GB RAM) a single $512 \times 512 \times 512$ volume can processed in about 1 hour starting from a stack of 512 16-bit TIFF images, each 512×512 pixels, and ending with a rendering of the sitched-together surface patches found in the volume. Most of this time is taken up by the DAFF algorithm. This has not changed since last month because performance improvements in the DAFF algorithm have been balanced by the change needed in step 5 of the algorithm that require more flood-fills.

Running the DAFF algorithm in an AWS EC2 t2.micro instance speeds it up by a factor of 3 so that it runs in 20 minutes. My short term aim is to be able to process each $512 \times 512 \times 512$ volume in less than 5 minutes - a four-fold increase in performance is needed to do this.

Future improvemens

Further optimizations of DAFF

There are two potential improvements that have been conceived of but not yet implemented. Firtly, when any-plane projection was introduced to the DAFF algorithm, it reduced the performance. Some of this reduction could be mitigated by automatically turning the user-specified plane vectors into constants in the code, and re-compiling the code prior to each run. The projection function would probably then be compiled to more efficient code.

Secondly, the flood fill algrithm is queue-based (by necessity for the DAFF algorithm to work correctly), and pushes all neighbours of each filled voxel onto the queue. They are tested for fillability when they are taken out of the queue later. Performance would probably be better if fillability was assessed prior to pushing them onto the queue. Better still, the fillabiilty of all neighbour voxels of each surface voxel could be pre-computed and stored, so that this would never need to be done during flood-fill (where it is currently done many times). The number of memory accesses during the inner-loop of flood fill would be significantly smaller if this were done.

Using GPUs

So far GPUs have not been used for this project. The basic image processing operations that preceed the DAFF algorithm in the

pipeline could easily be run on GPUs using existing image processing libraries. Some of the available AWS EC2 virtual machines are GPU-equipped.

Pre-computation of flood-fill neighbour fillability, described in the previous subsection, could also be implemented easily using GPUs - effectively off-loading more computationally-intensive work from the inner loop of flood-fill.

Flood-fill itself is not so easy to parellelise, but it would probably be possible to run some of the multiple flood-fills needed by DAFF simultaneously.

Systematic exploration of surface voxel detection methods

So far, the exploration of image processing methods for surface voxel detection in this pipeline has been done on a trial-and-error basis. It could be done more systematically if some metrics were developed for comparing methods of identifying surface voxels. This would need to be done both in terms of the surface voxels input into the pipeline, and also on the output from the pipeline, since an apparently good surface-voxel-identifying algorithm might not fare as well as a less good algorithm if the pipeline is incapable of correctly handling the extra information output by the better algorithm.

Source code

Source code used in this project can be found at `https://github.com/WillStevens/scrollreading`.

References

[1] `https://github.com/WillStevens/scrollreading/blob/main/report.pdf`