Finding papyrus scroll surfaces and damaged areas using flood fill

William Stevens

30th August 2024

Finding papyrus scroll surfaces and damaged areas using flood fill

This project explores the use of basic image processing operations and flood fill for finding distinct surfaces and surface patches in CT scans of the Herculaneum Papyri. Scans are split into $512 \times 512 \times 512$ voxel cubes, these are processed to identify surface voxels. Because the scrolls are damaged, surface areas in one scroll layer are often connected to surface areas in another, so basic flood filling of surface voxlels often ends up filling several layers in the same colour. Two approaches are explored to avoid this problem. The first is to modify the flood fill algorithm so that it prefers flat areas. This preference seems to help the algorithm avoid filling multiple layers in the same colour. The second approach is an algorithm to automatically detect where the connections between one layer and another are and plug them. Once surface patches have been identified they are stitched together using an algorithm that mathches boundary coordinates.

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. Within the domain of possible solutions to the problem of unwrapping the scrolls there is still much unexplored territory. This report describes methods for automatically identifying distinct surfaces patches and damaged areas in which two surfaces come into contact with each other.

I became interested in the Vesuvius Challenge after seeing a TV documentary about it in May 2024. After extracting a few surface areas manually, I initially started looking into processing $512 \times 512 \times 512$ voxel cubic volumes using flood fill to identify and colour distinct voids between papyrus layers, and then finding the surfaces at the edge of the filled voids. There were a oouple of problems with this approach : it doesn't handle compressed areas of papyrus, and because the scrolls are damaged, flood filling a void between two layers often leaks into neighbouring voids. Some basic image processing can be done to plug small holes, and larger holes can be plugged manually, but this is time consuming.

Identifying distinct surfaces and surface patches

After reading about Thaumato Anakalyptor [1] I changed the approach and focussed on finding and filling distinct surfaces rather than the voids between layers. A 3D Sobel filter was used to help identify surfaces facing towards the scroll umbilicus. Surface voxels are defined based on a combination of a threshhold transition from scroll to void and the output of the Sobel filter. Surfaces defined in this way are only one or two voxels wide, so flood filling them can be done fairly quickly. Unfortunately because the scrolls are damaged, surfaces that were distinct in the intact scroll intersect one another, flood filling tends to flood a large number of surfaces at once. Figure 1 illustates this.

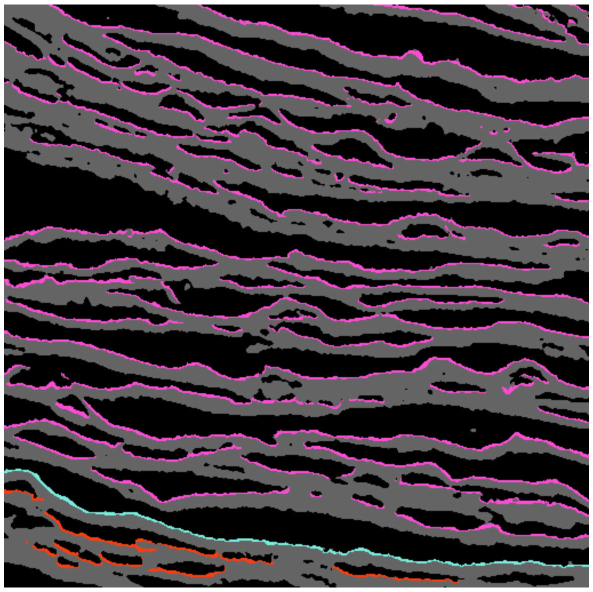

Figure 1: Because the scroll is damaged, the surfaces coloured in magenta intersect sometimes. Flood filling fills a large fraction of the surfaces in this volume, rather than filling distinct surface layers.

I found two ways to address this problem. The first way (flatness-affinity flood-fill : FAFF) is to modify the flood fill algorithm so that it prefers flat, simple areas. This was done by only allowing a voxel to be filled if the centre of mass of all surface voxels in the neighbourhood of the voxel doesn't deviate too much from the voxel's coordinates. The hope is that areas of surface with connection to other surfaces fail this test. Figure 2 illustrates this. It is partially successful, but doesn't completely solve the problem of flood fill leaking between surfaces.

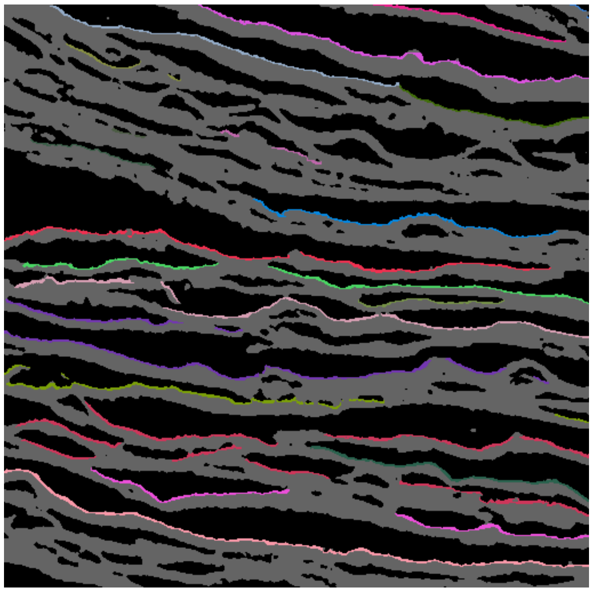

Figure 2: Result of filling surfaces using FAFF. Some damaged areas are avoided, but there are still places (e.g. the violet filled surfaces on the middle-left of the image) where surface intersection results in several layers being filled.

The second method (damage-avoiding flood-fill : DAFF) is to use flood fill to detect and plug damaged areas. This works by picking random surface voxels, projecting flood-fill onto a 2D surface and using this to detect when intersection occurs (i.e. projecting to a point that already has a projection). Flood-fill begins afresh from the point that resulted in intersection, until intersection occurs again - this results in two points close to an intersection, and flood fill is used for a third time to detect the mid-point between these two points. The mid-point is then marked as a plug - a voxel that should be considered unfillable so as to prevent intersection. Figure 3 illustrates the operation of the DAFF algorithm upto this point. Once all such plugs have been found so that there is no longer any intersection, flood fill is used for a fourth time to fill all distinct surfaces areas. Figure 4 illustrates the result of doing this.

To date, I have used either FAFF or DAFF. In future it may be better to use both in conjunction by applying FAFF first, then using DAFF on any surfaces that still show intersection. FAFF is faster but doesn't completely prevent intersection, whereas DAFF guarantees no in-

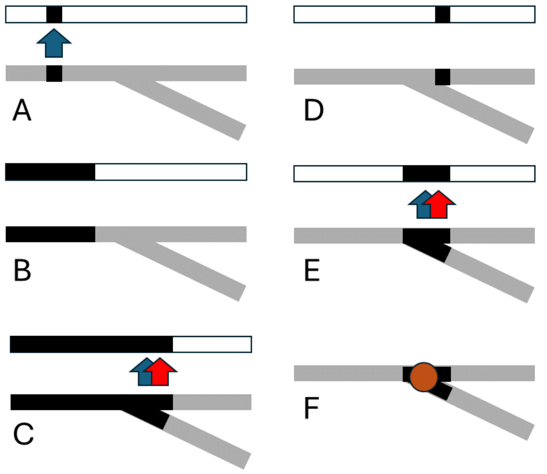

Figure 3: Explanation of the DAFF algorithm, A. Flood fill begins from a randomly chosen surface point. Flood fill is projected onto a 2D surface perpendicular to the direction to the umbilicus. B. Flood fill continues. C. Intersection is detected when two surface voxels are projected to the same 2D point. D. Flood fill begins again from the intersection point. E. Filling continues until intersection occurs again. F. Flood fill is used for a third time to find the mid point between the points found in D and E. This mid-point is labelled as a plug - a voxel through which flood fill cannot propagate.

tersection, but at greater computational cost. Since DAFFs performance more than linearly decreases with increases in surface volume, it's faster to use DAFF on the relatively small surfaces output by FAFF rather than on the entire connected surface of the whole $512 \times 512 \times 512$ volume.

I chose to use $512 \times 512 \times 512$ cubic volumes rather than the $500 \times 500 \times 500$ volumes used by other work because early on in my work I implemented a 3D version of Gosper's Hashlife algorithm [2] to perform flood-filling, and this works best when the volume dimensions are a power of 2. The Hashlife algorithm didn't seem to have any advantages over a conventional flood-fill, probably because there is not much regularity in the scroll strcuture. I have kept

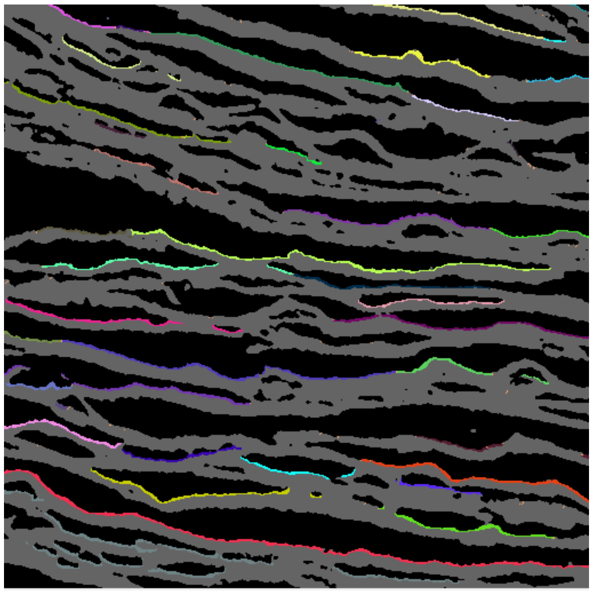

Figure 4: Result of filling surfaces using DAFF. There is no surface intersection, but some surface patches are small.

the same volume size because it is possible that Hashlife may still be useful - DAFF works by repeatedly flood-filling and then back-tracking in parts of a volume. Since Hashlife can exploit regularities in time as well as space, it may still have some value in this project. I have previously worked on a 3D implementation of Hashlife [3] that will run up until a condition is met and then halt - the intersection condition in DAFF could be handled in a similar way.

Details

When FAFF is used, the processing pipeline has the following steps:

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. Five 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 > 220 and where the Sobel vector points no more than 90 degrees away from the direction towards the scroll umbilicus are retained.

5. A voxel is called *basic-fillable* if it is identified by both step 3 and step 4, and also has at least one empty neighbour in a $3 \times 3 \times 3$ neighbourhood.

6. A voxel V is called *fillable* if it is both *basic-fillable* and if the centre of mass of all *basic-fillable* voxels in a $7 \times 7 \times 7$ neighbourhood is no more than 1.7 voxels away from V. (This is done as an integer calculation - see the source code for the exact definition).

In step 6, the flatness threshhold can be altered to change the behaviour of the algorithm. If the algorithm is too permissive then it tends to fill all surfaces in the same colour, if it is too restricive then it leads to small disconnected patches.

7. Surfaces are dilated by 5 voxels to fill in any holes left by the restrictions placed on the flood fill algorithm. It is possible that dilation will cause surfaces to begin to spread to incorrect layers, but because dilation is only 5 voxels and because all surfaces are dilated simultaneously and will meet in the middle of any problem regions, it is hoped that any errors caused by dilation are small.

8. All distinct surfaces that exceed a volume threshhold of 10000 voxels are exported into CSV files.

9. Surfaces are rendered so that the results can be inspected. Currently this is done using a simple projection onto a plain perpendicular to the umbilicus direction, with no attempt to flatten the surface.

10. Neighbouring surfaces patches within a $512 \times 512 \times 512$ volume are joined together, provided that their common boundary is > 100 voxels.

11. Surfaces from neighbouring $512 \times 512 \times 512$ volumes that abut each either are joined together using a similar algorithm to step 10.

Steps 10 and 11 are still being worked on. Preliminary results from both are shown in the next section.

The pipeline can be modified to use DAFF by replacing step 6 with a flood-fill that fills all *basic-fillable* voxels, runs DAFF to generate plugs, then re-runs flood-fill with the plugs in place.

So far, this work has been carried out in a region of scroll 1 located in the +y direction from the scroll axis, so that the direction to the scroll axis does not need to be worked out, and can be assumed to be in the -y direction for all volumes. For this pipeline to work in other regions of the scroll, it will be necessary to work out an approximate local surface orientation with respect to the scroll axis. This can be done in a similar way to what is done in Thaumato Anakalyptor.

The various threshholds described above have not been chosen systematically and are not necessarily optimal.

Example output

Figure 5 is a 3D view showing where DAFF puts plugs among surface voxels in a region where several surfaces intersect. In some cases it is clear why DAFF has chosen the plug location, but less clear in other instances. A closer examination of DAFFs behaviour is needed to better understand this. Test cases would also help to verify that it is behaving as expected.

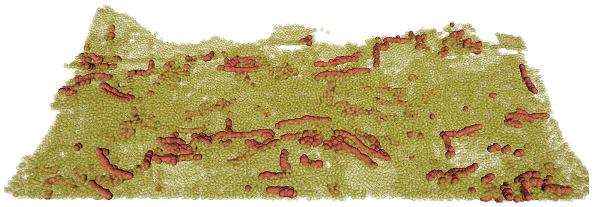

Figure 5: An example showing where DAFF thinks plugs (in red) should be placed to prevent flood-fill from propagating between several sheets

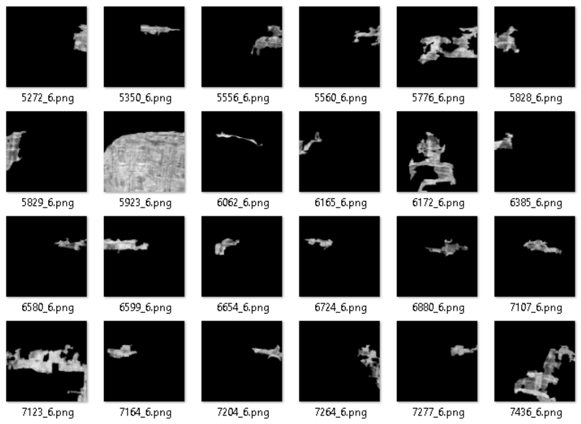

Figure 6: Some surfaces patches extracted from a single 512 × 512 × 512 volume using DAFF

Figure 6 illustrates some surface patches obtained from a single 512 × 512 × 512 volume using DAFF. Figure 7 shows some results of automatically stitching these patches together, and Figure 8 is a close-up of one of these surfaces. When stitching patches together, it is possible that surface intersection may be re-introduced. To help prevent this, only patches with a shared boundary of > 100 voxels were stitched, but visual inspection revealed that 4 out of the 31 surfaces showed signs of intersection, and I have not yet explored the reasons why. The same surface stitching problem faced by others is likely to arise. It is possible that information that DAFF gives about where it thinks plugs should go could help with this problem.

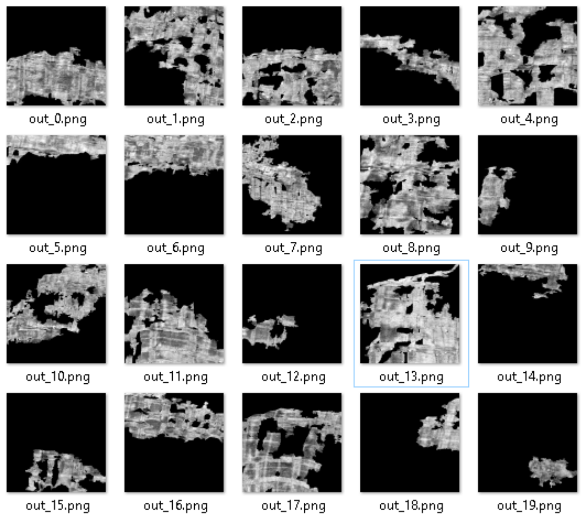

Figure 7: Some surfaces stitched together from the patches in Figure 6

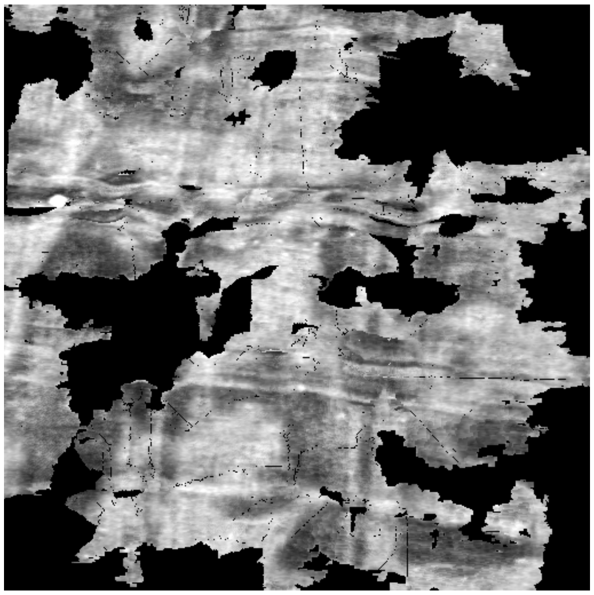

Figure 8: A single surface stitched together from some of the patches in Figure reffig:surfaces

Figure 9 illustrates the largest stitched together surface so far obtained from the 12 cube region that has so far been used for this project. (This is from an earlier run of the pipeline than the other figures in this section and used FAFF rather than DAFF, and didn't do any within-volume stitching prior to stitch-

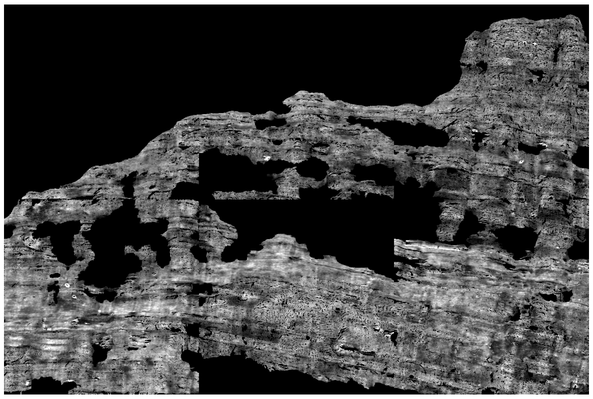

Figure 9: A 1.6cm by 0.8cm area stitched together from surfaces obtained from 8 512 × 512 × 512 cubic volumes

ing surfaces from neighbouring volumes).

Performance

On modest hardware (A single core of a 1.1 GHz Pentium N4200 with 4GB RAM) a single 512 × 512 × 512 volume can be filled with FAFF in 5-10 mins. DAFF takes longer, and currently takes an hour to process a volume in which most surfaces intersect. Smaller regions of intersecting surfaces can be processed in a few minutes. A combination of FAFF followed by refinement with DAFF may perform well - this has yet to be explored. DAFF has not yet been optimised, so once this is done it is reasonable to expect that FAFF followed by DAFF might take 15 minutes to process a volume to a state where distinct surfaces patches are coloured with no overlap.

The surface-patch neighbour-finding algorithm is implemented in Python and hasn't been optimised yet - it takes about 10 minnutes to run. Reimplementation in C and optimization would probably reduce this to an insiginificant amount of time - a minute or so.

From looking at tables of CPU performance, I see that the CPU I am using is at least 10 times slower than the fastest desktop CPUs currently available, so it is not unreasonable to think that faster CPUs or GPU hardware should enable each 512 × 512 × 512 volume to be processed in under a minute. A goal of being able to process a single scroll in under a day seems reasonable.

Limitations, future improvemens, and ideas

Understanding and optimizing FAFF and DAFF

FAFF was implemented as an analogy for a flat blade, larger than small holes in the papyrus, sweeping over a surface, but not being drawn into holes or defects. It is not known whether this analogy is an accurate reflection of what the algorithm actually does. Carefully examining the behaviour of the algorithm in close up around defects would shed light on its behaviour. Looking at places where DAFF succeeds in preventing propagation between surfaces but where FAFF does not might help to refine FAFF. FAFF has a lower computational cost than DAFF.

Sometimes DAFF places plugs in unexpected places. This could be to do with how it would behave if presented with a single turn of spiral with overlapping ends - it would place plugs in the midpoint between the ends of the spiral - i.e. half way along the spiral at the opposite side from where the ends overlap. If it could be made to place plugs closer to where the overlap occurs, this might be better. Also it does not aim to find a minimum number of plugs that prevents intersection, but this would probably be a worthwhile thing to do if it is computationally tractable.

Generally, identifying and classifying the types of defect that can result in surface intersection would probably lead to better algorithm design.

Exploring unfilled areas

Most surfaces found within a $512 \times 512 \times 512$ volume are incomplete. It would be interesting to explore some of the areas of incompleteness and assess whether anything can be done to improve the coverage. A better definition of a surface voxel might improve completeness. Some unfilled areas will be compressed scroll. I haven't yet explored how to handle these areas, and how to define a surface in these areas.

Source code

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

References

[1] `https://github.com/schillij95/ThaumatoAnakalyptor`

[2] Gosper, R.W.: Exploiting regularities in large cellular spaces. Physica D. 10(1-2) 7580 (1984)

[3] Adapting Gosper's Hashlife Algorithm for Kinematic Environments, Stevens W. M. (2010).Proceedings of the 2010 Workshop on Complex Systems Modelling and Simulation (Luniver Press, Frome, UK), pages 75-91. `https://www.cosmos-research.org/docs/cosmos2010-proceedings.pdf#page=85`