Further improvements to a virtual unwrapping pipeline

William Stevens

30th September 2025

Further improvements to a virtual unwrapping pipeline

This report describes progress on an automated segmentation pipeline described previously [1]. Over the last two months work has been done on improving the performance of the pipeline, working out how to process the patch alignment data that the pipeline produces, and comparing the patch growth algorithm with similar parts of VC3D. Example output measuring $100cm^2$ is shown.

Introduction

In previous reports I described a method of using surface predictions for growing flat surfaces based on a physics-based simulation of a flat surface [2]. This method seems to be able to detect at least some areas where the surface predictions are wrong, and avoid those areas.

The first step is to produce a vector field from a surface prediction zarr, where the vector field at each point near a surface points towards the surface. The surface predictions for scroll 1A produced by Sean Johnson in May 2025 were used: `https://dl.ash2txt.org/community-uploads/bruniss/scrolls/s1/surfaces/s1_059_ome.zarr/0/`

The work in this report uses a $2048 \times 2560 \times 4096$ vector field zarr starting at coordinates $x = 2688, y = 1536, z = 4608$.

Next, flat patches are grown, constrained by the vector field. Patch growth stops when stress in any part of the patch exceeds a threshhold, or when the patch reaches a prespecified size limit. After a patch has been grown, a new patch is started by selecting a pseudo-random point on the boundary of all patches grown so far.

Finally patches are stitched together by finding an affine transformation that aligns them using points that they share.

Efficiently representing the boundary of a growing surface

During growth of a surface, seeds for new patches are created at randomly chosen points on the boundary of the surface. In earlier versions of this pipeline, the boundary was identified by maintaining a mask image of the growing 2D surface, using image edge detection to find the boundary, then looking up the 3D coordinates of the 2D boundary point. There were a couple of problems with this:

1. Choosing a random boundary point took time proportional to the area of the growing surface.
2. A mapping from 3D volume coorindates to global 2D surface coordinates must be maintained as the pipeline is running. That mapping is costly to change if found to be wrong.

These problems were resolved by explicitly

keeping track of the points on the boundary in 3D coordinates. When a new patch is added to the surface, only the part of the boundary near the new patch needs to be updated. Also, there is no longer any need to create a 3D to 2D mapping until after the pipeline has finished running, when the mapping can be optimized using all patches. Because no 3D to 2D mapping is needed when the pipeline is running, the 2D reverse-lookup index into the 3D surface that was needed in the previous version of this pipeline is no longer needed.

The part of the pipeline that keeps track of the surface boundary works as follows:

1. Let S denote the set of points in the surface.
2. Let B denote the boundary of the growing surface.
3. Grow a patch P. Let C denote the boundary points of the patch. These are the last-added points of the patch.
4. If this is the first patch, set B = C and goto 8 (Done).
5. From B remove any points that are in P: i.e. erase from the current boundary any points in the new patch.
6. From C remove any points that are in S: i.e. erase from the new boundary any points in the current surface.
7. Set B = the union of B and C
8. Done

Boundaries are represented using the same type of chunked compressed data structure used for representing surfaces (described in the previous report). This is implemented in `bigpatch.cpp`. Random points are selected using `randombigpatchpoint.cpp`. Steps 5 and 6 are implemented using the `erasepoints.cpp` program. Step 7 is implemented using `addtobigpatch.cpp`.

Figure 1) illustrates a boundary growing by several patches.

Ideas for future work on surface boundaries

If a surface grew perfectly with no sheet switches, then the boundary would always be flattenable using an isometric deformation. If a sheet switch occurs, then the boundary may not be flattenable. Testing this (which is probably faster than testing whether a surface is flattenable) could be another way of detecting when and where (i.e. in which patch) sheet switching has occurred.

If any sheet switches are not detected in this way, then it's possible that the 3D representation of the boundary doesn't correspond to the boundary of a 2D surface. If this occurs it isn't necessarily a problem - the main purpose of keeping track of the boundary is to find new places to grow patches where patches don't exist yet. A later processing step should weed out any patches that caused sheet switching.

The compressed chunked storage format is probably overkill for the boundary and this part of the pipeline seems to unexpectedly take several seconds. A simpler representation of the boundary might be faster.

Refining patch positions - solving the problem of misaligned growth forks

In the previous version of the pipeline, alignment and positioning of patches into a growing surface was done one at a time as they were produced. This meant that positions couldn't be changed if they were later found to be wrong. Leaving patch alignment and positioning until after the pipeline has finished means that it can be optimized using information from all patches.

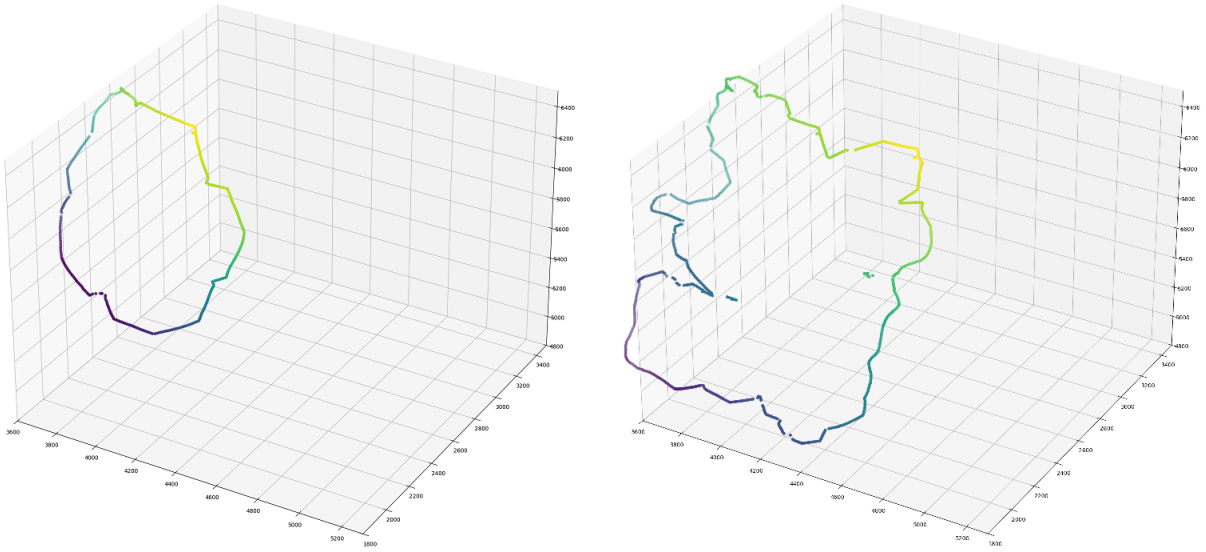

Figure 1: A boundary grows in 3D space after several patches are added to a surface.

Whereas the chunked compressed data structure storing the surface used to contain the global 2D coordinates and 3D volume coordinates of each patch point, it now contains the patch-local 2D coords, 3D volume coordinates and patch number. A separate text file (`patchCoords.txt`) contains affine transformations for aligning each pair of overlapping patches, along with an estimate of the patch radius and data about the variance of each patch aligment transformation. For each pair of overlapping patches, affine transformations aligning them are derived by randomly selecting pairs of overlapping points. The variance of of these transformations is used to decide whether a patch needs to be flipped: if there are lots of overlapping points, but the affine transformations derived from them are all different, it is likely that the patch needs to be flipped. Only if the variance of both the x and y translation is small enough (less than 10) is the patch alignment stored (the median transformation is used). Typically the translation variance is less than 1.

In the previous report, a problem was described where a growing surface could fork and then the two forks could meet again, but be misaligned. This problem is effectively solved by leaving patch alignment and positioning until after the pipeline has finished.

Currently a ball and spring model is used to refine patch position and alignment. The model calculates positions and orientations of patches using information in the `patchCoords.txt` file described above. After initial placement of patches the model is simulated until it reaches equilibrium so that any misalignment between forks is effectively distributed among both paths that lead to the fork. This is implemented in `patchsprings.py`. Figure 2 shows this realignment happening.

Example output

Figure 3 shows about $100cm^2$ produced by the pipeline showing about 10 wraps around the umbilicus. I believe that the holes that appear in each winding are areas where surface prediction had difficulty. The same areas also look problematic in the 2023 PB segments (manual segmentation was also challenging).

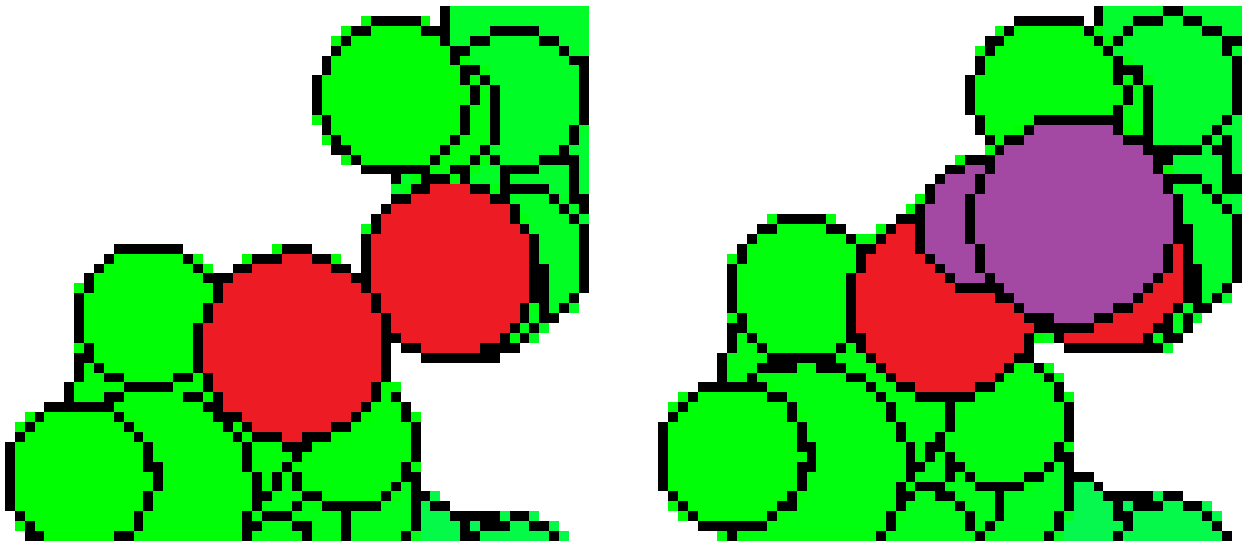

Figure 2: The two red patches and some of their neighbours realign with each other after the purple patches are added.

Figure 3: About $100cm^2$ of surface produced from the pipeline. Occasional patch misalignments occur, which have a downstream effect elsewhere.

Figure 4 shows all of the patches that were used to render this image. Red and purple patches are older than green patches.

Comparison of output with a 2023 Prize banner segment

The area of the scroll shown in figure 3 overlaps with part of the innermost segment of the 2023 Prize Banner region (segment 20251016151002). Figure 5 shows a 1.5cm wide section of the area that they have in common.

Figure 6 shows a comparison of a small region of this area produced by the pipeline described in this report. (A `Show and tell` thread on the Vesuvius Challenge Discord server contains an animation that switches between the two images to help comparison). It is clear that the output from this pipeline is more distorted.

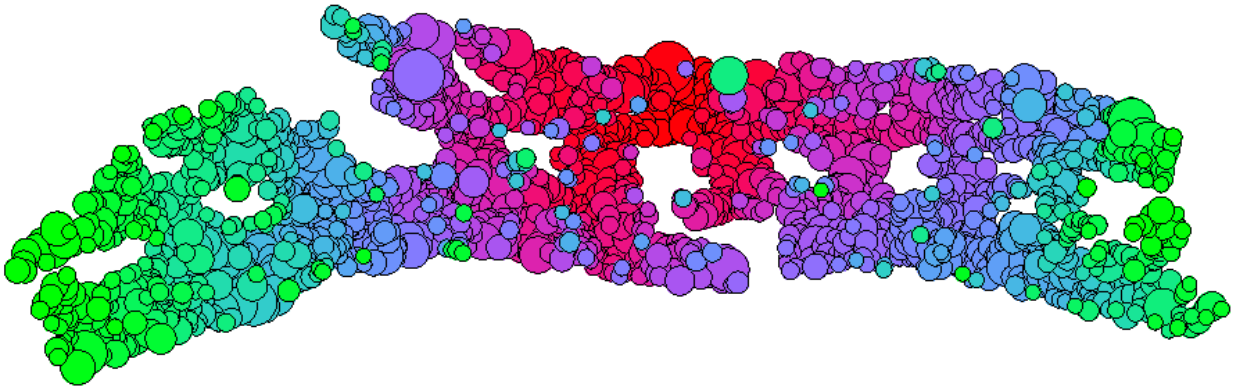

Figure 4: The patches that were used to construct the surface in figure 3.

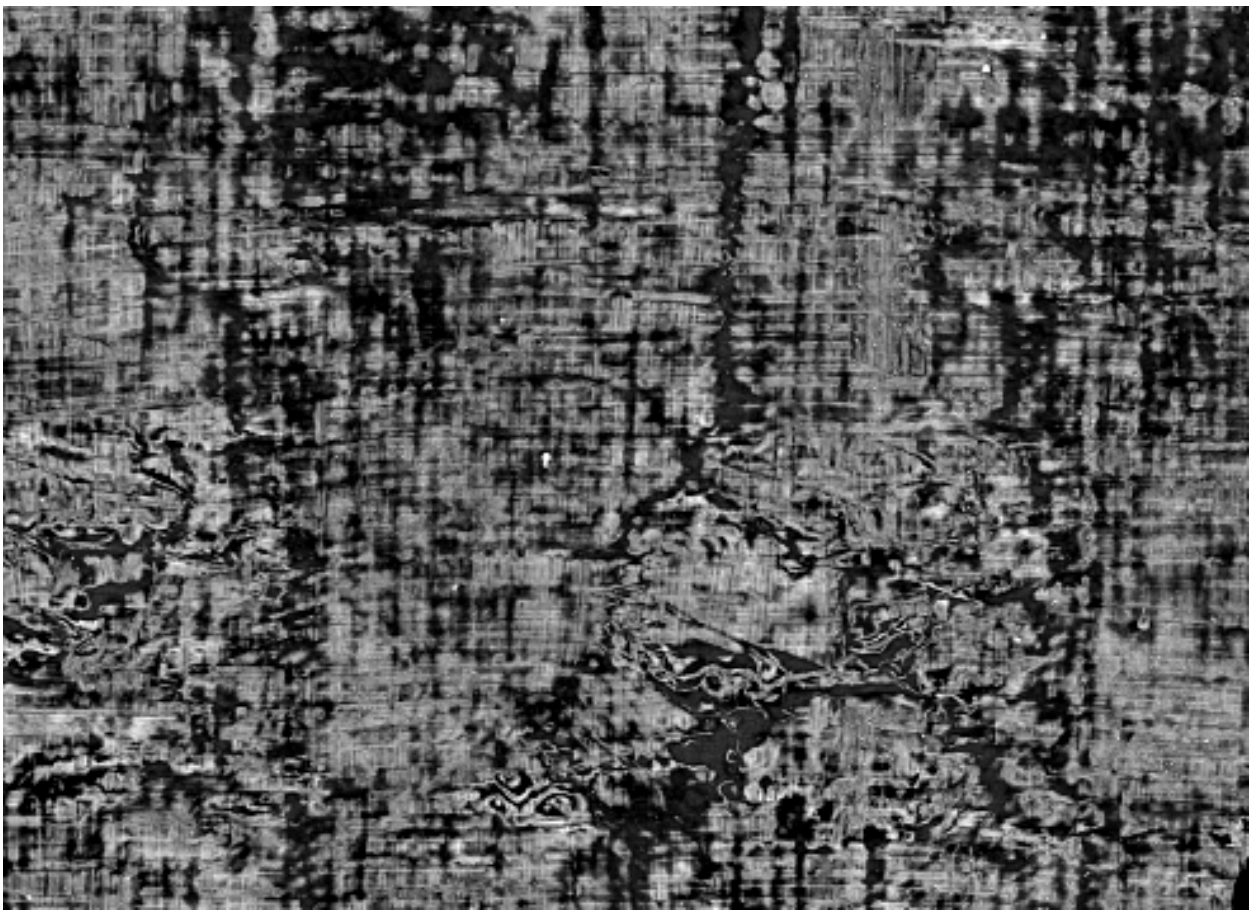

Figure 5: A small area of 2023 Prize Banner region.

Comparison with VC3D

I've done an informal comparison of the patch growing part of this pipeline (file `simpaper8.cpp` with the approach taken by VC3D's `GrowPatch.cpp`. I apologise for any errors or omissions in this comparison.

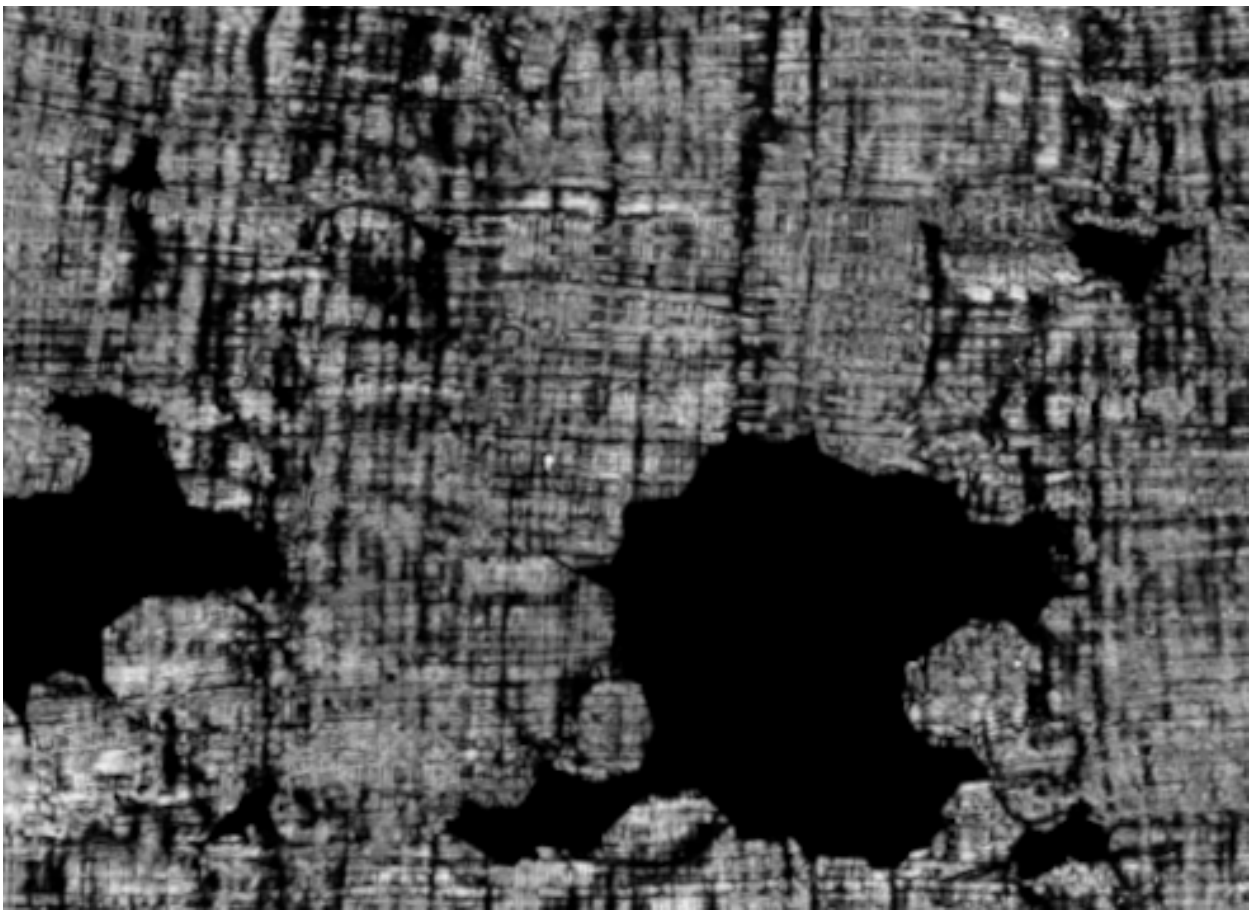

Figure 6: The same region produced by the pipeline described in this paper.

The main difference is that `GrowPatch.cpp` uses a constraint solver (Ceres Solver) to optimize the location of quadmesh vertices, whereas `simpaper8.cpp` uses a dynamic simulation to do the same thing: constraints are translated into forces that act on quadmesh points.

The forces acting on quadmesh vertices in `simpaper8.cpp` come from:

1. A smoothed vector field where voxels near a surface prediction point towards the prediction. This pushes quadmesh vertices into the surface prediction.

2. The 8 neighbours of a quadmesh points, acting to maintain the correct distance between them.

Previous versions also had a force that would maintain distances to four neighbours spaced 2 unit distances away to resist bending. This was removed because it didn't seem to make any difference.

`GrowPatch.cpp` has constraints corresponding to these, plus some constraints that try to:

1. Make the patch normals point in the expected direction.

2. Orient the local x,y axes of the patch in a particular direction.

The quadmesh spacing used by `GrowPatch.cpp` is typically 20 voxels. I have tried quadmesh spacings of 3,4 and 6 with `simpaper8.cpp`. I settled on 4 so that it

will be easy to make the outputs compatable with anything in VC3D that expects a spacing of 20 by downsampling. `simpaper8.cpp` uses about 100Mb of RAM to produce a patch 225 quadmesh vertices in diameter. Most of this is used to cache vector field data. The vector field zarr uses a chunk size of $32 \times 32 \times 32$ voxels to minimise the amount of RAM that `simpaper8.cpp` uses. The rate limiting part of `simpaper8.cpp` is the inner loop that calculates neighbour-distance forces.

GPUs are not used by `simpaper8.cpp` but this is something I'm likely to implement soon. I experimented with implementing the time consuming inner loop of `simpaper8.cpp` in PyTorch and it was possible but seemed slower (on a CPU) than I expected, which makes me think that a direct CUDA implementation would be better than GPU PyTorch, even though less portable.

I haven't done a proper area / hour / compute performance comparison of the two approaches yet but my first impression is that they are similar (but this ignores the work that goes into producing the vector field zarr that `simpaper8.cpp` needs - I am effectively treating that as being amortized over the many runs that are carried out during development of the pipeline). I also haven't done any assessment of quality, but I believe that `simpaper8.cpp` currently produces lower quality patches because ripples within patches can be observed in some places.

Future work

I'm currently working on making the output from the pipeline compatible with VC3D to help visualize where it is going wrong.

The surface in figure 3 was produced from a single thread on an EC2 g4dn.xlarge instance in 8 hours. I beleive (from a comment on Discord) that scroll 1A is 9.3m long by 15cm high, so $100cm^2$ is 0.7% of the scroll surface. A 35-fold improvement in the performance of the pipeline would enable scroll 1A to be processed in a day. This seems possible using more threads and/or CUDA.

Patch positioning and alignment needs more work and could perhaps be better handled as an optimization problem if suitable objective and penalty functons can be found.

Surface rendering is currently only approximate, at the moment it is used mainly to help spot anything going wrong with the pipeline. It doesn't take account of the small changes in distances between patches that can occur after initial placement in the ball and spring model. In future it will need to do this - the ball and spring model will effectively be used to output a distortion field for each patch that will ensure consistent rendering at the joins between patches (assuming that this is a soluble problem when more than two patches meet in the same place).

I have thought of a better way of producing the vector field zarr that will enable the guassian filter size to be reduced (cheaper to compute) but still flatten out ripples. Currently voxels outside a surface prediction point to the surface prediction, but voxels inside point nowhere (those near the edge of a surface prediction have a small inward pointing vector after guassian smoothing). If the inside voxels were set to point in the opposite direction to the nearest empty voxel(s), then there would be no areas within the surface prediction not subject to a force. Hopefully this means no gap in which ripples could form. Smoothing with a small window size would make sure that no vector feld discontinuities arise.

Source code

The source code for this pipeline has been put into a single standalone folder here: `https://github.com/WillStevens/scrollreading/pipeline4`. This is a self-

contained folder that contains all of the code needed to run the pipeline (including code to produce the vector field zarr). This folder won't be significantly altered (bug fixes only) once this report is produced - future work will be done in a new `pipeline5` folder.

The only depedencies are blosc2 and libtiff-dev.

I am in the process of moving the various parameters that are needed for the pipeline into parameters.json and documenting them there. Currently only the quadmesh size and volume coordinate and size parameters are located there.

References

[1] `https://github.com/WillStevens/scrollreading/blob/main/report.pdf` to `https://github.com/WillStevens/scrollreading/blob/main/report8.pdf`

[2] `https://github.com/WillStevens/scrollreading/blob/main/report6.pdf` to `https://github.com/WillStevens/scrollreading/blob/main/report8.pdf`