Growing flat sheets on scroll surfaces

William Stevens

26th April 2025

Growing flat sheets on scroll surfaces

This report describes progress on an automated segmentation pipeline described previously [1]. Over the last two months work has been done on exploring an approach towards flattening surfaces that builds on previous work of using particle-based simulations to flatten surfaces, but rather than simulate the flattening of a non-flat surface (described in previous reports), it builds a flat surface in place, particle by particle - it can be thought of as being like flood-filling a surface with a ball-and-spring model - where the balls and springs are contrained to be isometric to a flat surface.

Introduction

Previously I have worked on constructing surfaces of scrolls based on the processing of $512 \times 512 \times 512$ voxel volumes using a pipeline that does the following:

1. Identify surface points (initially using basic image processing operations, later using Sean Johnson's surface predications)
2. Flood fill surface points and identify where this leads to an apparent intersection of surfaces - break up the flood-filled surface points at these intersections to produce patches (the DAFF algorithm).
3. Reassemble the patches into surfaces using an optimisation algorithm that penalises intersection (simulated annealing was used).
4. Flatten and render the surfaces, filling in holes by interpolation and working out which surfaces in neigbouring $512 \times 512 \times 512$ volumes abut each other.

The flattening and hole-filling stage worked by squashing a pointset flat, filling the holes in the flat pointset, then unflattening the pointset back to it's original position - the points where the holes were get dragged back to locations consistent with the surface being flat.

When considering how to make this stage more efficient, I first began to reduce the time needed to squash points flat by flattening to a surface that is already a good fit to the pointset. During implementation of this I realised that it might be simpler to replace the flood-fill stage with a stage that grows a flat surface and maintains the flatness all the time, as far as possible.

I think that this ends up having some similarities to the work done by Hendrik Schilling (growing patches from a seed subject to physically-based constraints) and Paul Henderson (fitting a spiral to a scroll). I hope that some of the problems I've encountered and the possible solutions to those problems might also be relevant to the work that others are doing.

Growing sheets on surfaces

A sheet is a flat ball-and-spring model isometric to a flat plane. This is illustrated in figure 1 where the highlighted red-coloured particle is attached by springs (shown in orange) to it's neighbours.

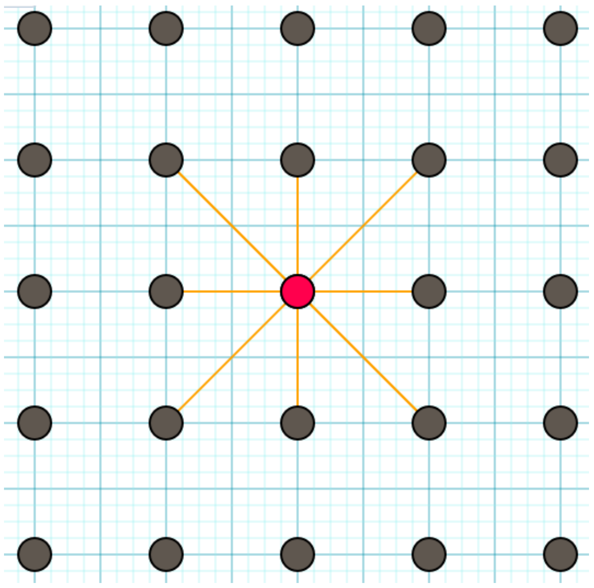

Figure 1: The red particle is connected via orange springs to its neighbours

A 2D array holds the 3D coordinates of each particle that makes up a sheet. Each particle in the 2D array has Hookian forces applied to try and maintain the expected distances of its 8 neighbours. Springs aren't explicitly represented : the 2D array determines the neighbours of a particles and the expected distances between neighbours. The 3D coordinates of each particle give the actual distances between neighbours. Particles are also subjected to forces in a vector field derived from Sean Johnson's scroll surface predictions. This vector field has a resolution of 1 voxel and is calculated as the direction to the nearest surface voxel, or the average of the direction to all nearest surface voxels if several surface voxels are the same distance away. The value of the vector field is zero in surface voxels. After this has been calculated it is smoothed using a 26-neighbour window. Vector fields arrived at in other ways would probably work, this is the only one that has been tried so far. The vector field effectively points towards surfaces, and has a large magnitude just outside surfaces so as to push the growing sheet back towards the surface if it strays outside. Figure 2 shows a slice of the vector field.

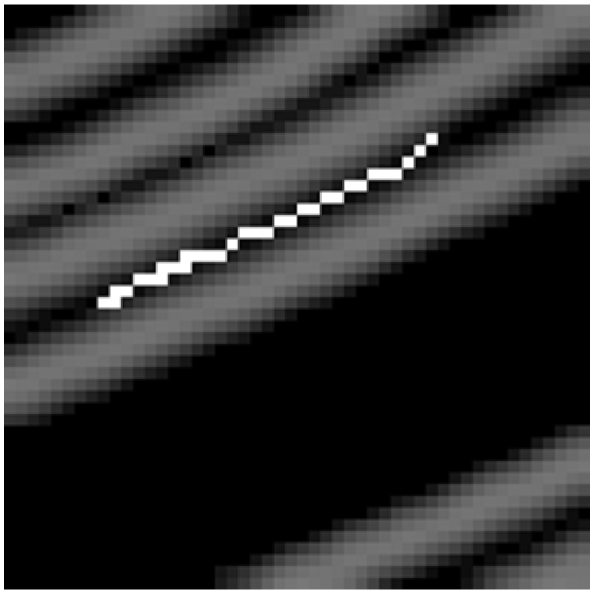

Figure 2: A small 50 x 50 voxel slice showing the magnitude of the vector field surrounding a surface, with a small flat sheet growing within the surface

The representation of the sheet in the 2D array is extended by looking for points that neighbour the sheet and that meet either of these conditions:

1. There is a neighbour in direction D, and also a neighbour in position 2D. The cooridnates of the new point are obtained by projecting a line out from these two particles.

2. There is a neighbour in direction D, and also a neighbour in position 90 degrees from D, and also a diagonal neighbour at the position between these two neighbours. The coordinates of the new point is obtained by projecting a line from the diagonal neighbour through the midpoint

of the line that joins the other two neighbours.

If the new point isn't in a surface voxel then the sheet isn't extended to the new point. If the new point is within a distance of 2 from an existing point then the sheet isn't extended to the point.

Every time the boundary of the sheet is extended, a ball-and-spring simulation is run until either the maximum total force on any particle is less than some threshhold, or a specified number of iterations have been executed. This gives the sheet a chance to bend, move and orient itself to accommodate the newly added particles.

Unless the surface voxel identification is perfect, it is likely that this filling process will eventually lead to sheet switching. Therefore it will be necessary to either detect when sheet switching happens (for example using methods similar to the DAFF algorithm described in previous reports, or possibly using stress analysis illustrated later on), or alternatively grow small patches and then stitch them together using an optimziation method that penalizes sheet switching (similar to the simulated annealing method used previously).

By trial and error, a spacing of 3 voxels between particles in the ball and spring model was found to work. This is likely because it is large enough that the filling step correctly fails at the edges of surfaces (smaller steps made the sheet bunch up so that it had a rippled appearance), but small enough that filling doesn't jump between distinct surfaces.

Unlike my previous work, sheets needn't be confined to $512 \times 512 \times 512$ volumes (although I haven't used larger volumes than that during development), since zarrs are used as the scroll surface data source, and for rendering surfaces.

Rendering a sheet is a simple matter of doing linear interpolation in the square regions between sheet points. A lower resolution rendering can be done directly using the coordinates of each point.

If necessary, normals for each particle can be calcualted based on the averaged directions of the vectors pointing to neighbouring particles. This would be useful for accesing different depths within the scroll surface.

Preliminary results

Figure 3 shows how a sheet grown in this way sits on a predicted surface.

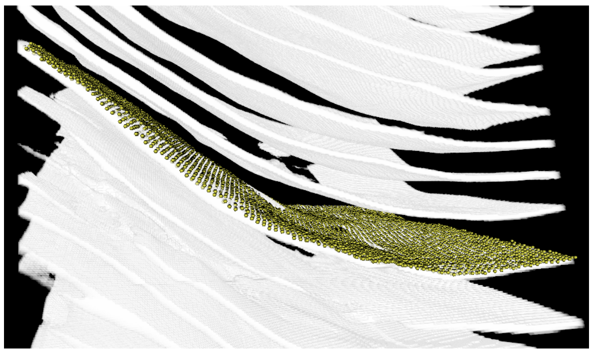

Figure 3: A sheet grown on a surface from scroll 1A

Figure 4 shows the result of growing a sheet on a surface within a $512 \times 512 \times 512$ volume, where the sheet is not allowed to grow outside voxels predicted to be surfaces.

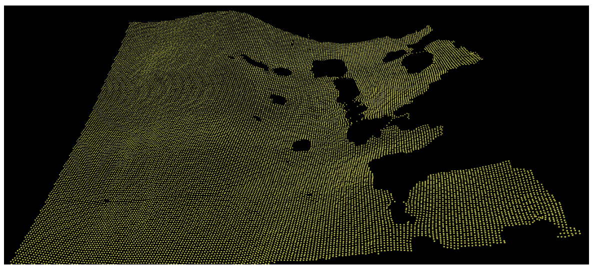

Figure 4: A grown sheet in scroll 1A with centre at approx z=2432, y=2632, x=4352

Figure 5 shows the result of rendering this sheet alongside a plot of the stress at each point. By compariing this with figure 4, it is apparent that there is something anomalous

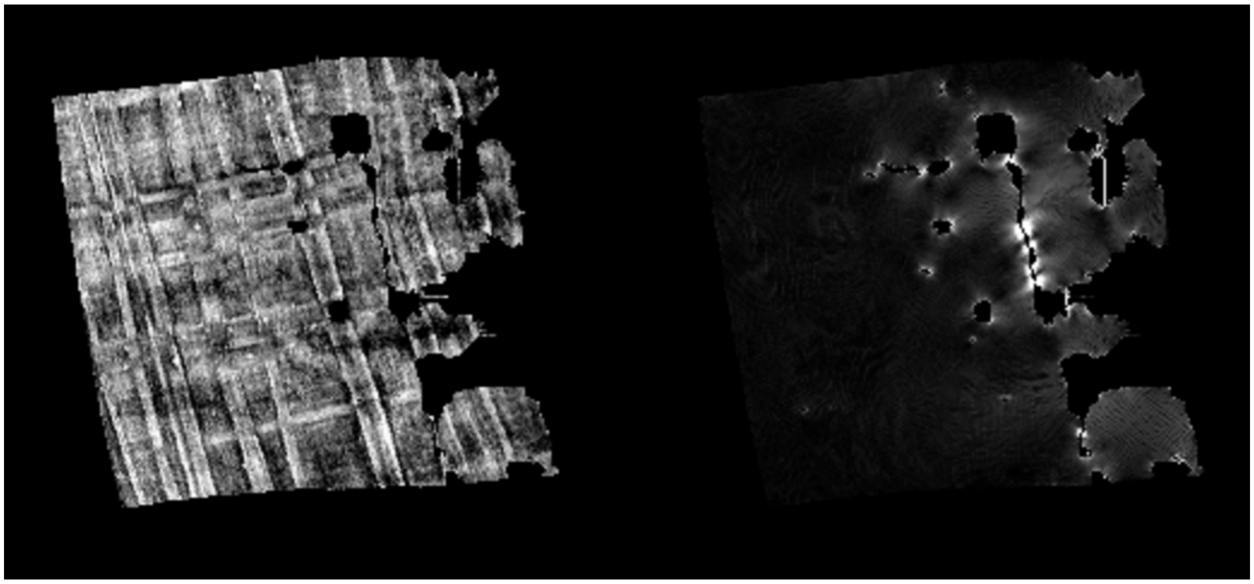

Figure 5: A rendering of the sheet from figure 4 alongside a stress plot

happening near the long void at the middle right of the sheet. This area is also highly stressed, indicating then when attempting to fill the surface with a flat ball and spring model, two points that should be close together in space are in farther apart than expected. I have not yet invesitgated whether this represents damage to the scroll that causes non-flatness, or a problem with the algorithm. Either way, the stress plot indicates approximately where something has gone wrong, and being able to detect this is a good thing.

Next steps and future work

The next thing I want to do is to work out exactly how the stressed areas arise - do they indicate damage to the scroll and/or do they show places where the algorithm is failing?

After that I want to check that the algorithm is capable if filling holes in surfaces. After generating a sheet, if all points in the sheet greater than some specified distance away from a hole are held fixed in place, the vector field is turned off, and sheet growing continues, then the sheet ought to grow over and fill in any holes.

I think that using a priotiy queue for deciding which parts of the sheet to grow would probably be useful - growth could be prioritised in low stress, highly flat areas, and closesness to the seed could also be a factor. This would hopefully mean that large error-free sheets could be grown, and these error-free sheets would impose constraints on the more difficult to grow areas - the more constraints that there are on those areas, the more likely that growth will proceed correctly in them.

Currently inter-particle repulsion hasn't been implemented because it probably has no effect when growing small sheets. But once sheets wrap once around the scroll, impleenting repulsion might help sheets fit correctly into highly compressed areas.

Currently the surface predications are obtained from ZARRs, and rendering also makes use of scroll ZARRs, but intermediate steps such as the vector field calculation use large CSV files - it would probably be better if those steps used ZARR files too.

A CUDA version of the sheet growing program will probably have a large effect on performance, since it should be highly parallelizable. Also because the resolution of the vector field is a single voxel, and the velocity of each particle is much lass than 1 voxel distance per iteration it would probably be efficient to store a copy of the vector field value along with the particle properties, and only update it when the integer value of the particles coordinates changes.

Source code

Source code related to this report can be found at `https://github.com/WillStevens/scrollreading/simpaper`.

There are several programs:

1. `papervectorfield_closest.c` and `papervectorfield_closest_csv.c` : Programs for calculating the vector field. The former uses .zarr files or URLs as sources for the surfaces and uses the vesuvius-c library to access them. The zarr files used were '`https://dl.ash2txt.org/community-uploads/bruniss/scrolls/s1/surfaces/gp_only/old/1213aug/1213_aug_results/1213_aug_erode_threshold-ome.zarr.7z` initially, then `https://dl.ash2txt.org/community-uploads/bruniss/scrolls/s1/surfaces/full_scroll/s1-surface-erode.zarr/` later. This program has been used for computing vector fields for 512 x 512 x 512 voxel regions. The latter `_csv` version uses a CSV file rather than a zarr file as a data source - it was used when running on a system that didn't have zarr support.

2. `simpaper2.py` : A prototype of the sheet-growing program written in python using numpy arrays. This is quite slow and was only used for sheet sizes of about 50×50 particles to help develop the algorithms and refine the constant parameters.

3. `simpaper2.cpp` A C++ version of the sheet-growing program that is a few hundred times faster than the python version and has been used for sheets upto about 170 by 170 particles. Produces a CSV output of the particle positions along with data for the stress plot.

4. `render_from_zarr.c` Render a sheet using the scroll 1A zarr.

5. `render_stress.c` Render a stress plot.

References

[1] `https://github.com/WillStevens/scrollreading/blob/main/report.pdf` to `https://github.com/WillStevens/scrollreading/blob/main/report5.pdf`