Refinement of particle-based surface flattening and interpolation

William Stevens

28th February 2025

Refinement of particle-based surface flattening and interpolation

This report describes progress on an automated segmentation pipeline described previously [1, 2, 3, 4]. Over the last two months work has been done on improving particle-based simulations for flattening surfaces and for interpolating holes in surfaces based on the flatness constraints arising implicitly in the particle-based simulations. Work has also been done on optimising some other parts of the pipeline.

Introduction

One of the challenges in reading the Herculaneum Papyri is to work out how to track surfaces in places where there is substantial compression and/or confusion of one surface with another. One way of tackling this is to use flatness constraints: since all scroll surfaces were flat originally, this imposes a constraint on where the surface must lie, assuming that the surface is undamaged. The approach taken here is to use a fast particle-based simulation that models neighbouring particle connectivity (written in C++ using CUDA) to flatten the surface by applying a force on it that squashes it flat. After this, gaps in the squashed flat surface are identified and filled with blank 'hole' particles. Particles in the squashed flat surface are then forced back to their original locations, leaving the hole particles free to move subject to the connectivity constraints of the particle model. The hole particles will adopt a shape consistent with the deformation of the rest of the surface.

Particle based surface flattening and interpolation

Particle based surface flattening is implemented in the file 'surfaceFlatten3.cu'. Surface voxels were obtained from the output of one Sean Johnson's earlier runs. Each surface voxel is converted to a particle in the particle model, with connectivity and forces as described in the previous report [4]. Surfaces are flattened downwards onto a plane perpendicular to the approximate average surface normal of the $512 \times 512 \times 512$ volume.

Figure 1 shows the result of flattening a surface that contains numerous holes. Once the surface is flat, holes are filled in, shown in Figure 2 (the source code for this is in 'resample_flattened_and_holes.cpp'). Then the surface is restored to its original location, so that holes can be assiated with scroll voxels. Source code for this is in 'surfaceUnFlatten3.cu'. The flattened surface can then be rendered. Source code for this is in 'render_slices.cpp'

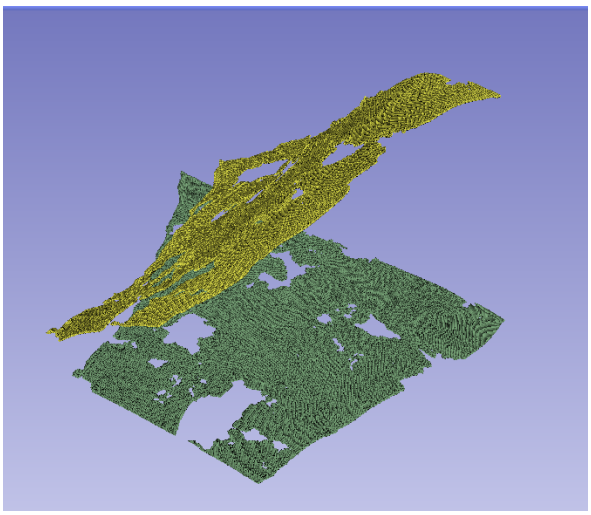

Figure 1: A surface before and after flattening

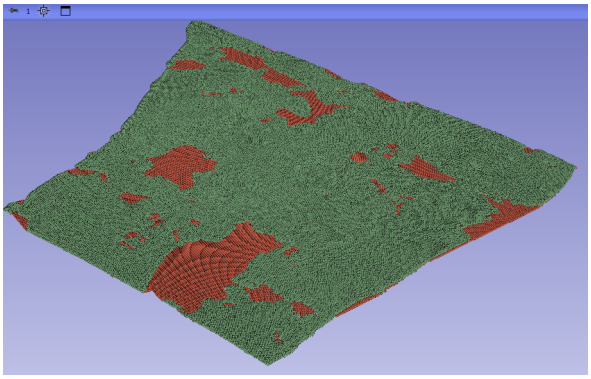

Figure 2: A flattened surfaces where holes in the surfaces have been filled in (coloured in red), but are not yet associated with any scroll voxels.

Joining together flattened surfaces

Once flat surfaces have been obtained from neighbouring $512 \times 512 \times 512$ volumes (overlapping along a 1-voxel wide plane), abutting surfaces can be identified by finding common voxels in the original volumes (implemented in 'neighbour2.py'). Among a collection of flattened surfaces, these common voxels won't necessarily align exactly after the flattening process, so some minor distortion of the 2D surfaces will be needed in order to align them - this hasn't been implemented yet.

Performance

Figure 3 shows neighbouring surface from 9 $512 \times 512 \times 512$ volumes rendered using a simple flat projection (i.e. without doing any flattening or interpolation).

Figure 4 shows the same surfaces after the flattening/unflatting/interpolation procedure described in this report. The dark lines between surfaces are due to the lack of alignment described at the end of the previous section. The difference in surface features are because both surfaces are from slightly difference depths. The central surface of the 9 shown here has an alignment problem that has not yet been investigated. Also only part of the lower right surface is present.

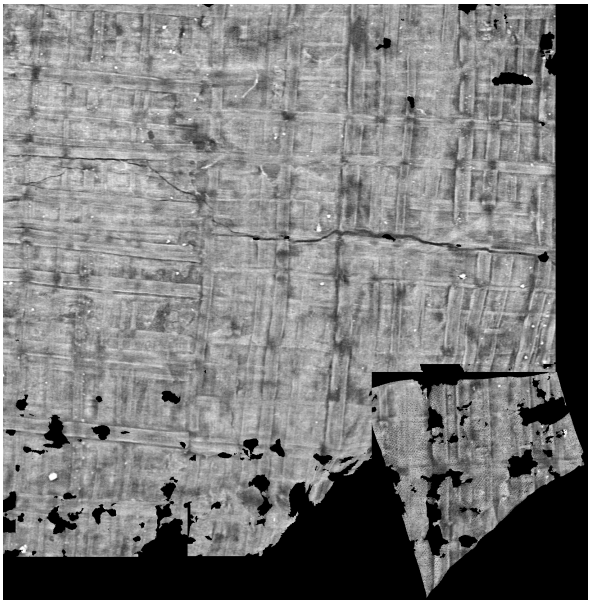

Figure 3: Rendered flattened surface.

The surface flattening step takes about a 10 seconds for a large surface (about 500K voxels) on an AWS EC2 g4dn.xlarge instance. The reverse step to interpolate surface positions currently takes longer, but should in principle take less time - it hasn't been optimised yet. There is probably still a lot of room for improvement - the algorithm is based on a particle simulation from a much earlier version of

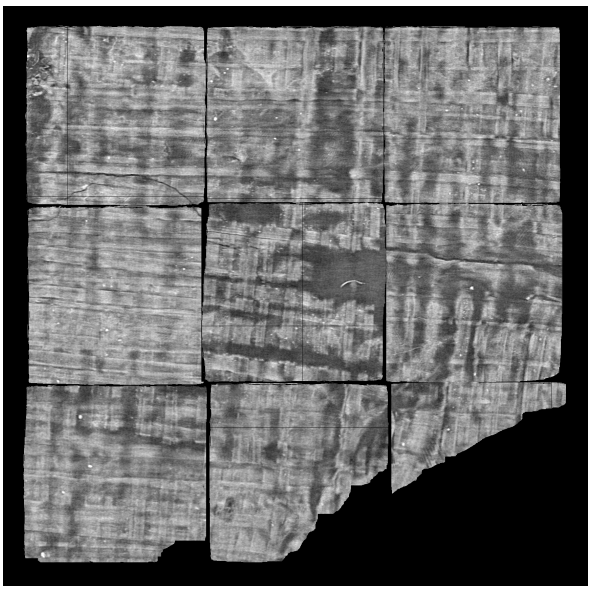

Figure 4: Rendered flattened surface.

CUDA, and I haven't spent much time working out how to make it faster. So it seems plausible that this could run in a few seconds per surface.

The code 'resample_flattened_and_holes.cpp' is not efficient and hasn't been optimised - this is currently the slowest part of the flattening/unflattening process.

Future work

It would be useful to compare interpolated surface with actual surfaces (by deliberately creating holes in surface data) to understand when the this process works and when it doesn't.

Surfaces in the particle model are floppy like a damp cloth, rather than stiff like a piece of paper. The behaviour of the physical model likely affects how well interpolation performs. Knowing exactly what the parameterization should be could be informed by understanding the physical properties of the charred scrolls, and also by examining how well the interpolation performs on known good surfaces.

A performance improvement has been implemented in surfaceFlatten4.cu - rather than squashing surfaces flat, they are flattened to a parellel plane that lies on the centre of mass of the surface to be flattened : some particles are moved up, some are moved down, until the surface is flat. This led me to think that it would probably be a good idea to carry out the flattening and unflattening/interpolation steps in-place, on all surface points within a $512 \times 512 \times 512$ volume at the same time. Doing this would allow neighbouring surfaces to repel each other during the interpolation step, which might give better results, especially in compressed areas.

During the unflattening step (when flat surfaces with filled holes are restored to their original position in a volume), particles bonds are stretched slightly. If there is a lot of stretching, this could imply that the surface is not a genuine scroll surface (e.g. perhaps sheet switching has taken place). It would be worth examining the stress and strain in the particle model to see whether it yields any useful information.

The DAFF algorithm (described in reports 1 and 2) uses flood fill to work out when surfaces intersect, and splits surfaces into patches that can be reassembled in a non-intersecting way. It depends on all surfaces within a $512 \times 512 \times 512$ volume to be roughly aligned in the same direction, within a few tens of degrees. Quite a lot of volumes, especially those towards the centre of the scroll, don't have this property.

The algorithm could be modified to work without this requirement by changing the criterion for detecting when intersection occurs. At the moment this is detected by an attempt to fill a voxel that projects to the same point as an already filled voxel. An alternative criterion could be when a line aligned with the surface normal vector of an already filled surface voxel intersects another filled voxel. This is computationally more expenive, but would work on highly curved surfaces, without any require-

ment the different surfaces within the same volume are roughly aligned with each other.

The whole pipeline is currently written as a collection of standalone pograms that must load their inputs and save their outputs. The pipeline has now reached a state where it would be useful to consolidate it into modules that share memory to avoid a lot of saving and loading of files.

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`

[2] `https://github.com/WillStevens/scrollreading/blob/main/report2.pdf`

[3] `https://github.com/WillStevens/scrollreading/blob/main/report3.pdf`

[4] `https://github.com/WillStevens/scrollreading/blob/main/report4.pdf`

[5] `https://en.wikipedia.org/wiki/Simulated_annealing`