Particle based surface flattening and interpolation, simulated annealing for patch assembly

William Stevens

31st December 2024

Particle based surface flattening and interpolation, simulated annealing for patch assembly

This report describes progress on an automated segmentation pipeline described previously [1, 2, 3]. Over the last two months work has been done on using 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 using an approach based on simulated annealing for assembling patches into surfaces within a $512 \times 512 \times 512$ volume. This approach attempts to maximise an objective function based on total edge-length of joined patches and on patch areas that overlap when projected approximately flatly.

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.

Another challenge which I believe crops up in several different approaches to autosegmentation is to work out how surface patches should be assembled into surfaces. My previous attempts at solving this involved growing surfaces patch-by-patch, evaluating the next best-fitting patch at each step. This has a problem where sometimes a surface will grab the wrong patch, and prevent it from being used in a place where it can contribute to correct surface growth. To get around this I've implemented an approach based on simulated annealing, where candidate patch joining solutions for all patches (typicallys 1000s of patches) in an entire $512 \times 512 \times 512$ volume are scored and penalised in one go - so that a wrongly placed patch is more likely to result in a worse global score than a correctly placed patch.

Particle based surface flattening and interpolation

Particle based surface flattening is implemented in the file 'surfaceFlatten3.cu'.

Each surface voxel is converted to a particle

in the particle model. Any pair of particles closer than 1.45 units are treated as having a connection between them whose length must be maintained during simulation. This is done using a Hookian force based on the separation between the two points, and is implemented in the function 'ConnectionForces'.

If any points get closer than 1 unit then a repulsive is applied to keep them apart. This force is proportional to the square of one minus the inter-particle distance. This is implemented in the function 'ParticleForces'.

A frictional force proportional to particle velocity is applied to all particles, and a constant gravitational force (the flattening force) is also applied. These are implemented in the function 'ParticleMove'.

The numerical values of the force constant parameters are somewhat arbitrary, and are based on success with previous particle-based simulations using the same parameter values rather than physical considerations.

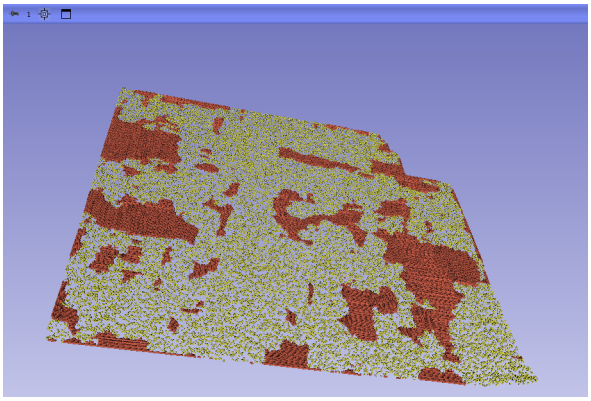

Figure 1: A flattened surface from a $512 \times 512 \times 512$ volume. Holes in the surface have been filled in (coloured in red), but are not yet associated with any scroll voxels.

Figure 1 shows the result of flattening a surface that contains numerous holes. Once the surface is flat, it is straightfoward to fill the gaps with 'hole' particles that don't yet correspond to any voxels in the scroll.

Once gaps have been filled with hole particles, the simulation is effectively run in reverse - particles that correspond to scroll voxels are gradually forced back to their original positions. The source code for this is in 'surfaceUnFlatten3.cu'. Test cases have been written to test this code, figure 2 shows a test case in which a flat square with a square hole in it is forced into a V-shape. The particles that make up the filled hole are constrained only by the connectivity physics. Figures 3 and 4 show how increasing the stiffness of the connectivity force constant improves the extent to which the hole particles conform to the expected shape.

This work is still in progress - a problem to be resolved is that sometimes forcing flattened particles back to their original positions causes unexpectedly large forces to be exerted on hole particles.

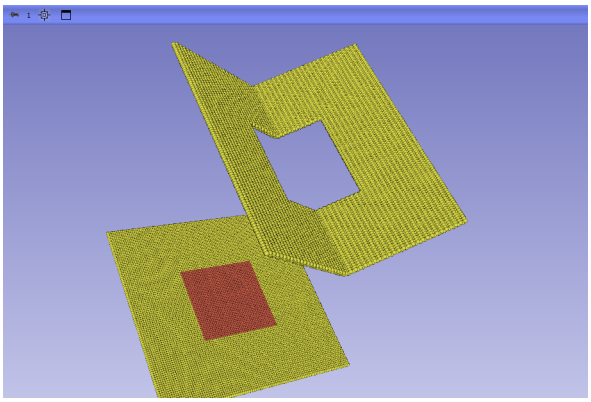

Figure 2: A test case showing a square with a hole in it. The hole is filled in (colour red), then the square is forced into the V-shape, with the hole constrained only by particle connectivity physics.

Simulated annealing for patch stitching

The previous algorithm for stitching together surface patches within each $512 \times 512 \times 512$ volume grows surfaces one patch at a time, look-

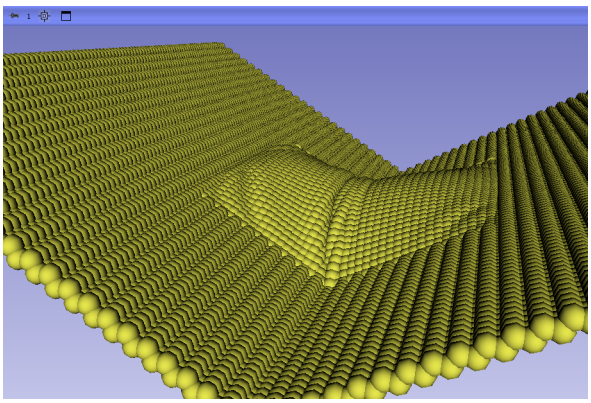

Figure 3: Running the test case with connection forces that are too elastic prevents the hole from adopting the folded shape that the rest of the surface imposes on it.

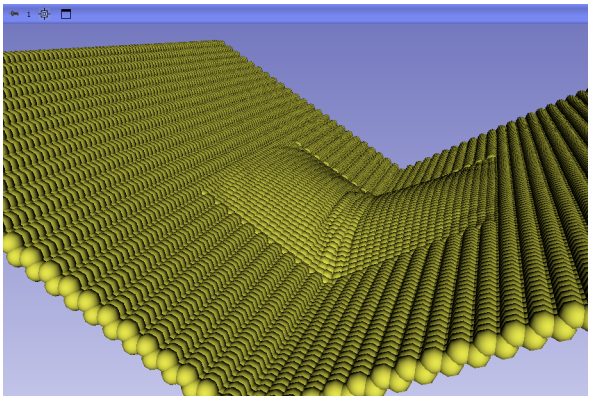

Figure 4: Running the test case with stiffer connection forces gives a better result.

ing for the best match at every addition step. This has a problem where a patch can be the best match for a surface, but would have been an even better match for another surface : the wrong surface stole the patch.

A new algorithm has been written (jigsaw3.cpp) to resolve this problem using a method based on simulated annealing [4]. It works as follows:

1. Work out which patches abut each other without significant overlap. These are all possible neighbours. Typically there are around a thousand possible pairs of candidates that could be neighbours.

2. A boolean connectivity vector represents a possible patch connectivy solution - each element of the vector specifies whether a pair of possible neighbouring patches are connected.

3. An objective function scores a connectivity solution by working out the size of the abutting points of all neighbouring patches (S) and the total number of points within any group of connected patches that project to the same point on an approximately flat surface projection (I). The objective function returns $5 * S - I$: a solution that maximises this score is considered a good solution. (Arrived at by trial and error, after inspecting the results).

4. During the course of simulated annealing the temperature is gradually reduced. A candidate solution is randomly modified, where the extent of the change depends on the temperature. If a better solution is found it is retained.

5. Once the temperature reaches zero, export all of the connectivity groups and call them surfaces. (The terminology in this report is to call the outputs of the patch stitching algorithm surfaces).

Comparing the behaviour of this algorithm (jigsaw3.cpp) with the previous one (jigsaw2.cpp) apparently shows that it produces a larger number of large surfaces within a single $512 \times 512 \times 512$ volume. E.g. in the volume of scroll 1 starting at x=3988, y=2512, z=1500 jigsaw2.cpp output 23 surfaces with > 131072 voxels, whereas jigsaw3.cpp outputs 30 such surfaces.

Figure 5 shows the differences between 4 of the surfaces output by these algorithms. The difference in the rightmost image may be due to a failure of jigsaw2 in using the wrong patch for a surface, which jigsaw3.cpp (correctly) does not try to use.

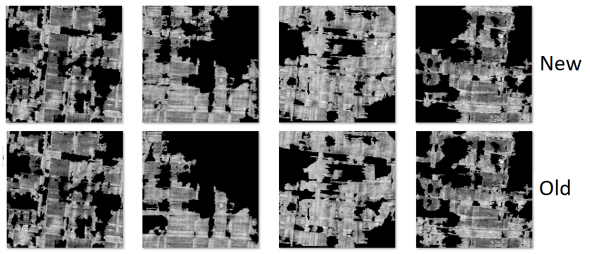

Figure 5: Simulated annealing based patch stitching (jigsaw3.cpp) versus the previous method (jigsaw2.cpp). The new approach seems better.

Performance

The surface flattening step takes about a minute for a large surface (about 300K voxels) on an AWS EC2 g4dn.xlarge instance. The reverse step to interpolate surface positions takes a similar amount of time. There is probably a lot of room for improvement - the algorithm is based on a particle simulation from a much earlier version of CUDA, and I haven't spent any time working out how to make it faster. Also, it probably isn't necessary to turn every single voxel into a particle - randomly picking 1 in 4 or even 1 in 10 would probably still work. So it seems plausible that this could run in a few seconds per surface.

No attempt has been made to optimize the simulated annealing step yet. There is plenty of scope for result caching and parallelism.

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://en.wikipedia.org/wiki/Simulated_annealing`