A not so short guide to all sorts

Dion Moult

December 2, 2015

This document assumes general domain knowledge and technical skills required for this project.

1 Stone

1.1 Quad-based toolpath generation

The details of the quad-based approach are discussed in the published paper[1]. The rest of the documentation is within the code itself, which is included with this document[2]. A simple demonstration of the KRL generation is also included[3], and will be used in the example below. Although the principles may be reapplied to any 3D modeling package, the Python script uses Blender libraries, and so using Blender to visualise and generate the toolpath is compulsory.

The `krl.blend` file includes two objects. The first contains an edge ring which may be used to test a coplanar approach, and another edge ring for a normal approach. The second object can be used as a target vector, to override the coplanar or normal approach. Once generated, the `krl.src` file may be used as an input to the `krl.3dm` Rhino and `krl.gh` Grasshopper file for visualisation with KUKA prc v2.

A useful tip when debugging is to enable debug mode and check indices of the object's vertices, edges, and so on. This can be done easily in the Blender Python console:

```
>>> bpy.app.debug = True
```

Once set, indices in the Mesh display section of the 3D view's properties (`n`) panel can be enabled.

In the first example, assuming

[1]Stereotomy of Wave Jointed Blocks
[2]src/krl.py
[3]src/krl.blend

2 Drones

2.1 Hello, photogrammetry-in-a-nutshell

There are many scanning technologies, each with tradeoffs. There are two umbrella classes of scanning technologies: contact, and non-contact. The former relies on physical contact with the scanned object, and so is not applicable in this project. The latter may be either active or passive. Active solutions emit an electromagnetic wave, and based off its round-trip time, triangulated position, polarisation, or deformation of a rigid pattern, can detect the shape of the surface. Passive solutions merely record the scene under stereoscopic vision, different lighting conditions, or are reconstructed through computer vision. Photogrammetry—our weapon of choice—is reconstructed through computer vision, and has benefits through superior captured textures, capturing a photographic dataset usable by stonemasons for damage anaylsis, has no distance or size limitations, and can use relatively stock hardware.

Photogrammetry belongs to the field of *structure from motion*, or *SfM.* SfM takes a collection of photos as input, processes it, and outputs a 3D point cloud. The processing consists of three major steps: feature extraction, feature matching, and sparse bundle adjustment. The first step identifies distinctive image features that may be common across multiple photographs. This uses the *Scale-Invariant Feature Transform*, or SIFT method, which is an $O(N)$ algorithm, and may be parallelized. The second finds matching features between photos, in an $O(N^2)$ computation. Finally, 3D geometry is estimated from the matched features. This is also practically a $O(N^2)$ process.

The final output from the photogrammetry process is a sparse point cloud. To make it more useful for visualisation, a dense reconstruction step then identifies likely surfaces. From this dense point cloud, the surface may be reconstructed, using a variety of surface reconstruction algorithms which provide a mesh-based output. Although the point cloud and their embedded RGB data is more "accurate" and contains enough colour to visualise the surface, the mesh-based output is desirable as it allows textures to be mapped onto it from the original photo collection. This UV-mapping process can be, and is automated. With an unbiased renderer providing lighting and autogenerated normal and occlusion maps, it can provide a very convincing reconstructed render.

2.2 Upstream documentation

Nothing beats upstream documentation for authority.

vsfm `http://ccwu.me/vsfm/index.html`

siftgpu `http://www.cs.unc.edu/~ccwu/siftgpu/manual.pdf`

cmvs / pmvs2 `http://www.di.ens.fr/cmvs/documentation.html`

pmvs2 `http://www.di.ens.fr/pmvs/documentation.html`

pba `http://grail.cs.washington.edu/projects/mcba/`

MeshLab `http://meshlab.sourceforge.net/`

2.3 Test cases

A Utah teapot equivalent is very useful for testing the reconstruction process. Thankfully, `pba` (`Bundler`) includes two examples which are quick to execute: `examples/ET` and `examples/kermit`. For convenience, the kermit files have been included[4].

For reconstruction, the Standford Bunny and friends that live inside the Stanford 3D scanning repository[5] should be used.

2.4 Scanning images

The best image dataset for photogrammetry has a large (70%) overlap, has an image sequence that shows an incrementally changing viewpoint, and has textured surfaces with lots of unique feature points.

It is impractical to manually capture photos of a large and complex object. A more practical approach is either continuous burst photography or video. The former is preferred, as photographs usually have a higher quality than a video, unless RAW video is possible with the camera.

If video is used, the frames may be extracted easily with `ffmpeg`[6]:

```
$ ffmpeg -i /path/to/input.video -r 2 -f image2 -start_number 0 image-%07
    d.jpg
```

The `-start_number` argument makes it convenient to append image sequences from multiple videos. Note that the `.jpg` extension is required for processing by `vsfm`. If image quality issues occur from the JPG encoding, the `-q` argument may be used, or instead bypassed by exporting to `.ppm`.

Additional format conversions are trivially done with `mogrify` from `imagemagick`.

2.5 Non-headless execution

Non-headless execution is trivial and documented on the vsfm homepage.

2.6 Headless execution

The `VisualSFM` documentation[7] and `-h` argument describes the command parameters quite well. In explicit terms, the `sfm` argument invokes headless execution. Additional arguments are appended for the matching process, sparse reconstruction process, and dense reconstruction process, in that order (with

[4] `src/kermit/*.jpg`
[5] `http://graphics.stanford.edu/data/3Dscanrep/`
[6] `https://www.ffmpeg.org/`
[7] `http://ccwu.me/vsfm/doc.html`

some additional options for fine-tuning). An example which performs all processes would be:

```
$ VisualSFM sfm+pmvs /path/to/jpgs/ /path/to/output.nvm
```

When processing large datasets, execution should be batched, and resumable if it fails. Let's say we are merely interested in the matching process, and it terminates before it has finished:

```
$ VisualSFM sfm+skipsfm /path/to/jpgs/^C^C^C
# Now, resume. All steps may be resumed in a similar manner.
$ VisualSFM sfm+skipsfm /path/to/jpgs/
```

Matching results in `*.sift` and `*.mat` files to store the results. Now that matching is complete, sparse reconstruction can begin. Reconstruction results are stored in a `.nvm` file.

```
$ VisualSFM sfm /path/to/jpgs/ /path/to/output.nvm
```

And once sparse reconstruction is complete, we can do the dense reconstruction.

```
$ VisualSFM sfm+loadnvm+pmvs /path/to/output.nvm /path/to/dense.nvm
```

This may be accompanied with the `+subset` argument without any implications. If the dataset is particularly large, it may be desirable to break up the first step into a separate sift (which generates `.sift` files) and matching (which generates `.mat` files) step.

```
$ find . -type f -iname "*.jpg" > photos.txt
$ VisualSFM siftgpu photos.txt
$ VisualSFM sfm+skipsfm ./
```

2.7 Execution times

Processing is not fast. Specifically, a rigorous dataset of a couple thousand images will already take up multiple days[8]. More images are better than less, with non-diminishing gains in reconstruction quality with an increase in the dataset. Before any further optimisations are made, it is important to make sure that the hardware ideally uses SiftGPU, rather than a CPU-based implementation, and uses multi-threaded matching and bundling. VisualSFM provides these features out of the box. A detailed analysis of VisualSFM's approach compared with a cluster based approach is described elsewhere[9].

2.7.1 Preemptive feature matching

A major bottleneck is the $O(N^2)$ image matching process. A solution is a one-time sort of the dataset into subsets to be matched, called *preemtive feature matching*. Changchang Wu has documented the approach[10]. By applying it,

[8] `https://groups.google.com/forum/#!topic/vsfm/PegIJOMOJaE`
[9] `http://ieee-hpec.org/2012/index_htm_files/Sawyer_rev.pdf`
[10] `http://ccwu.me/vsfm/vsfm.pdf`

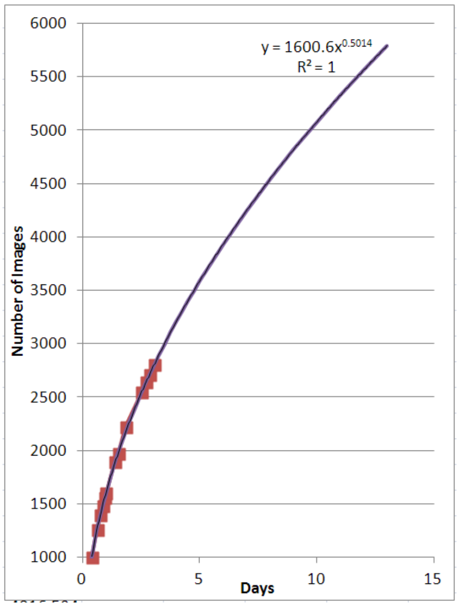

Figure 1: SiftGPU processing times on a CUDA-enabled Geoforce 660

at the sacrifice of potential diminishing returns of output resolution, the image matching turns into a $O(N)$ process. The `+subset` argument of `VisualSFM` enables this feature, and can be applied mid-way through an execution.

```
$ VisualSFM sfm+subset+skipsfm /path/to/jpgs/ /path/to/output.nvm
```

2.7.2 Manual execution of dense reconstruction

Note that this section does not apply for CMP+MVS reconstruction, where the process is significantly different.

`vsfm` does not offer a very detailed breakdown of the dense reconstruction process, which has implications for batching and distributing jobs to a cluster. The dense reconstruction step also has quite a few parameters which may significantly affect execuation time. Although these may be edited in `nv.ini`, they are unlikely to be globally appropriate, and so manual execution of each step is desirable. Finally, there are some options which are simply not exposed, or we would like to iterate through manually to optimise the results.

The dense reconstruction is broken down into 4 steps: `cmvs` preparation, `cmvs`, `genOption`, and `pmvs2`. `cmvs` preparation selects the appropriate matched photos for `cmvs` to execute, performs lens undistortion, and creates the necessary ascii input files. The inputs, processes, and outputs of the subsequent three steps are documented in their respective documentation, and so will not be elaborated here.

We can piggy-back on `vsfm` to generate the appropriate input for `cmvs`. This is an $O(N)$ CPU-intensive process. Once prepared, we can kill the `vsfm` process, and any rogue `cmvs` processes, and proceed manually.

```
$ VisualSFM sfm+loadnvm+cmvs sparse.nvm dense.nvm
[ ... snip ... ]
# images loaded:   n
Load NVM for dense reconstruction
Save to ./dense.nvm ... done
Undistorting n images in model #n
[ ... snip ... ]
^C^C^C
```

The index of images to undistort are provided in `dense.nvm.cmvs/00/list.txt`, so we can easily check to see how many images we are expecting, and whether or not we have finished undistorting all images:

```
$ test -e $(echo $(tail -n 1 list.txt) | sed -e 's/\\/\//g' -e 's/://')
    && echo "yes" || echo "no"
```

Once complete, the `dense.nvm.cmvs/00/` directory holds the formatted `bundle.rd.out` file required for `cmvs` to run. The `cmvs` step is a memory intensive process. Even though it is possible to skip directly to the `pmvs2` step, `cmvs` should be executed first, as it provides clusters for `pmvs2` to execute in parallel.

Nevertheless, `cmvs` can still be daunting task for very large datasets. The memory usage depends primarily on the filtered photo collection in `dense.nvm.cmvs/00/visualize/`. Once loaded into memory, it will start generating clusters. There are two parameters for this process: `maximage` max images for the clusters, and `CPU` number of CPUs or cores in the processing machine.

Although the `cmvs` documentation stresses the `maximage` parameter for controlling memory, a much more significant impact on memory usage can be made by modifying the source images. This is because all source images must be loaded for `cmvs` to calculate the clusters. However, `cmvs` does not simply cluster images blindly[11]. Clustering both determines image neighbours, and following that, cluster neighbours, and additionally filters low resolution geometry. Modifications on the source would have an impact on the latter, but this may be acceptable if the clustering at least makes processing possible.

Let's say we have a memory limit, and we are feeling frisky.

```
$ cp -r ./dense.nvm.cmvs/00/visualize ./dense.nvm.cmvs/00/
    visualise_highres
$ cd ./dense.nvm.cmvs/00/visualize/
$ mogrify -resize 50% "*.jpg"
```

Once memory restrictions are solved, we can proceed. However, `maximage` affects the cluster size, and therefore can result in a (seemingly) infinite loop if a value that is too small is selected[12]. A dumb but valid technique is to attempt

[11] http://www.cse.wustl.edu/~furukawa/papers/cvpr10.pdf
[12] https://groups.google.com/forum/#!topic/vsfm/Iq-ETOwwsdQ

as small an integer n as possible above 1, and if a timeout is reached, attempt again with $n + 1$. Starting with $n = 5$, as used by `vsfm` seems sensible.

```
$ cmvs ./dense.nvm.cmvs/00/ 5 8
```

Once complete, `genOption` can parse the `ske.dat` to create the `option-%04d` files for `pmvs2`. There are far more parameters for this process, and are documented with `pmvs2`. It is encouraged to use the default values by the documentation, not the ones by `vsfm`, and change the third argument, `level` if resource limits are hit. As this is merely a parsing step, execution of the following command will be fast, but has implications with `pmvs2`.

```
$ genOption ./dense.nvm.cmvs/00/ 1 2 0.7 7 3 8
```

Depending on the number of `option-%04d` created, the jobs may then be queued to the cluster. If frisky memory management was applied, now is the time to backtrack our resolutions, as the smaller quantity of images per cluster means that memory should no longer be an issue.

```
$ mv ./dense.nvm.cmvs/00/visualize ./dense.nvm.cmvs/00/visualise_lowres
$ mv ./dense.nvm.cmvs/00/visualize_highres ./dense.nvm.cmvs/00/visualise
```

Now it's just a matter of processing the job queue with `pmvs2`.

```
$ pmvs2 ./dense.nvm.cmvs/00/ option-0000
```

For testing without a cluster at hand, we have to do it all ourselves.

```
#!/bin/bash
for f in ./dense.nvm.cmvs/00/option-*
do
    name=`basename $f`
    pmvs2 ./dense.nvm.cmvs/00/ $name
done
```

2.7.3 Structured sequences

Planning out reconstructions through *structured sequences* can bypass much of the clustering that `cmvs` provides. There are senarios where this is necessary. For example, downsampling the images may not be acceptable for the `cmvs` step. Or perhaps the structure is far too complex and needs such a large dataset that would take too long to process atomically.

In this scenario, essentially the clustering that `cmvs` provides will be done manually. The building must be scanned in portions, and reconstructed in portions, and finally stitched together. When doing this, stitching may be problematic.

There are two stitching strategies: pre-surface reconstruction, and post-surface reconstruction.

2.8 Headless environment

Although `VisualSFM` can be run headlessly, the package has dependencies on a graphics card, some X libraries, and an OpenGL environment. Cheap virtualised hardware—such as Linode—is inappropriate. A CPU-only matching and sparse reconstruction has been tested on a Linode environment, but is not documented here, and nor is it worth pursuing.

2.8.1 Sandbox environment

A Gentoo system is recommended due to the rather bleeding edge condition of the packages and pipeline. This gives us a replicatable vanilla environment that supports and encourages compilation from source. Test setups were done in a `chroot`.

Although the `chroot` may be left as an exercise to the reader[13], a standard setup for a stage3 installation is shown:

```
# Location of the chroot
$ mkdir ./stone
# Note that hardened stage3's have not been tested
$ links https://www.gentoo.org/downloads/mirrors/
$ tar xvjpf stage3-*.tar.bz2 -C ./stone/
$ cp /etc/resolv.conf ./stone/etc

# This mount process should be packaged for convenience
$ mount -t proc proc ./stone/proc
$ mount --rbind /sys ./stone/sys
$ mount --make-rslave ./stone/sys
$ mount --rbind /dev ./stone/dev
$ mount --make-rslave ./stone/dev

# Speak, friend, and enter ...
$ chroot ./stone /bin/bash

# Once inside ...
$ env-update
$ emerge-webrsync
$ emerge --sync
```

For completeness, we also unmount when finished:

```
$ umount -l ./stone/dev{/shm,/pts,}
$ umount ./stone{/boot,/sys,/proc,}
```

2.8.2 Alternative environments

As a quick detour, I'd like to mention alternative black box environments that may help in testing.

[13] `https://wiki.gentoo.org/wiki/Handbook:Main_Page`

A detailed vanilla compilation guide is provided for a Ubuntu 12.04 environment[14]. It is worthwhile reading to gain an understanding of the dependencies and their build process.

Additionally, there is also ArcheOS[15], a Debian-based distribution meant for archaeologists. It bundles other useful utilities, including the Python Photogrammetry Toolkit (PPT) which may be of interest if there are licencing issues with vsfm.

2.8.3 Basic package compilation and dependencies

Portage, as expected, does not keep vsfm and friends in the main tree. The `cg` overlay provides the related packages.

```
$ emerge -av layman
$ echo "source /var/lib/layman/make.conf" >> /etc/portage/make.conf
$ layman -L
$ layman -a cg
```

There is the usual barrage of keywords, uninteresting USE-flags, and licensing.

```
$ mkdir /etc/portage/package.keywords
$ touch /etc/portage/package.keywords/vsfm
$ echo "=media-gfx/PoissonRecon-6.13 ~amd64" >> /etc/portage/package.
    keywords/vsfm
$ echo "=media-libs/graclus-1.2 ~amd64" >> /etc/portage/package.keywords/
    vsfm
$ echo "=media-libs/pba-1.0.5 ~amd64" >> /etc/portage/package.keywords/
    vsfm
$ echo "=media-gfx/pmvs-2-r1 ~amd64" >> /etc/portage/package.keywords/
    vsfm
$ echo "=sci-libs/cminpack-1.3.4 ~amd64" >> /etc/portage/package.keywords
    /vsfm
$ echo "=media-gfx/vsfm-0.5.26 ~amd64" >> /etc/portage/package.keywords/
    vsfm
$ echo "=media-gfx/siftgpu-0.5.400-r1 ~amd64" >> /etc/portage/package.
    keywords/vsfm

$ touch /etc/portage/package.use/vsfm
$ echo "media-gfx/siftgpu cuda" >> /etc/portage/package.use/vsfm
$ echo "x11-drivers/nvidia-drivers gtk2" >> /etc/portage/package.use/vsfm

$ mkdir /etc/portage/package.license
$ touch /etc/portage/package.license/vsfm
$ echo "dev-util/nvidia-cuda-toolkit NVIDIA-CUDA" >> /etc/portage/package
    .license/vsfm
```

[14]http://www.10flow.com/2012/08/15/building-visualsfm-on-ubuntu-12-04-precise-pangolin-desktop-64-bit/
[15]http://www.archeos.eu/

Note the `media-gfx/siftgpu cuda` above, as this assumes a CUDA environment, and all the 32-bit/64-bit issues and updated graphics drivers that come with it. Resolving this is an exercise for the reader.

In addition to the regular dependencies resolved by portage, `media-gfx/vsfm` requires `x11-libs/libXext` and `media-libs/freeglut`. These should be emerged before `media-gfx/vsfm`.

```
$ emerge -av libXext freeglut vsfm
```

Once compiled, vsfm expects to find sift in `./sift`. Unfortunately, the packaged binary is named differently. This is a simple fix:

```
$ ln -s /usr/bin/SimpleSIFT /usr/bin/sift
```

2.8.4 Using CPU-based sift for a non-CUDA environment

Once SIFT is found, matching and sparse reconstruction should work out of the box. However, if the environment does not have a CUDA-capable GPU, switching to CPU for sift may be necessary. This requires editing the `nv.ini` file, which is generated upon the first run of `VisualSFM`. Once run, it may be found in the counter-intuitive location of `/usr/bin/nv.ini`. It should be modified so that `param_use_siftgpu 0` is set. The configuration is heavily documented inline for more information.

Now that `VisualSFM` is searching for a CPU implementation of sift, we have the choice to use either Lowe's SIFT or VLFeat SIFT instead of SiftGPU[16]. Lowe's SIFT is already packaged including binaries, and so it is a simple matter to download it, and replace the previous symlink in `/usr/bin/sift` to point to Lowe's SIFT.

2.8.5 Patching CMVS and PMVS

Once matching and sparse reconstruction is working, there may still be issues with `cmvs` and `pmvs2`. Gentoo's `media-gfx/pmvs` is incomplete in that it only provides the `pmvs2` binary, and not `cmvs`. Although `media-gfx/cmvs` seems to provide the necessary binaries, they have a bug which I have elaborated upon[17], but not yet figured out an appropriate patch for the overlay.

The solution is to compile pmvs2 manually. However, we can still reuse the `cmvs` binary provided by the overlay. The explicit steps (to be executed with due diligence) are shown:

```
# Let's make use of the overlay as much as possible
$ emerge -av media-gfx/cmvs
# Notice this is not pmvs-2-fix0.tar.gz
$ wget http://www.di.ens.fr/pmvs/pmvs-2.tar.gz
# lapack is a dependency for manual compilation
$ emerge -av virtual/lapack
```

[16] `http://ccwu.me/vsfm/doc.html#download_urls`

[17] `https://groups.google.com/forum/embed/#!topic/vsfm/QKZ9MbTKpeE`

```
$ tar -xzvf pmvs-2.tar.gz
$ cd pmvs-2/program/main
$ cp mylapack.o mylapack.o.backup
$ make clean
$ cp mylapack.o.backup mylapack.o
$ make depend
$ make
$ mv /usr/bin/pmvs2 /usr/bin/pmvs2-portage
$ ln -s /path/to/pmvs-2/program/main/pmvs2 /usr/bin/pmvs2
```

2.8.6 Job queues and synchronisation

With a headless setup needing to process many buildings throughout the year, it makes sense to elaborate on synchronising datasets/results to/from the client/server. A 4200 photo dataset, with 3840x2160px dimensions has a total size of 1.2GB. This illustrates the practicality of transferring over datasets to remote hosts.

The transfer process should be done securely, robustly, and explicitly one-way. `rsync` over `ssh` is a natural solution, with potential automation with `lsyncd` (not documented here, but trivial to implement).

```
$ rsync -avz /path/to/dataset/ user@host:/path/to/processor/inbox
```

2.9 Point cloud realignment

The generated point cloud has an arbitrary direction and scale. We need to realign the point cloud using datum points based off a canonical template for each building site.

2.10 Point cloud cleaning

Before the model is useful for visualisation or surface reconstruction, it is important to clean up the point cloud first. Nonsensical points and points unrelated to the scanned object should be removed. This is a visual process, and may be non-trivial to do manually, depending on the site. For example, dust, clouds, specular materials, camera artifacts, or simply poor computer vision can result in patches of clearly incorrect point clouds.

2.10.1 Manual cleaning

Depending on the surface reconstruction process that is planned, there are two approaches to manual cleaning. If a large amount of point metadata should be maintained (e.g. point normals, as required for poisson reconstruction), the cleaning should be done with a program that retains this, such as `Meshlab`. If this is not required, any 3D modeling suite, such as `Blender` would be more appropriate, as it offers much more convenient mesh selection tools. It should be noted that a modeling suite may not be able to render vertex colours, if they are not part of a face, and so may make it much harder to visualise what you are cleaning. Of course, it is possible to use both.

The actual process of selecting and deleting points is trivial, and not documented here.

2.10.2 Automated cleaning

When dealing with messy clouds, cleaning can be a tedious process. In some cases, where the object is concave in all axes, manual cleaning becomes impossible, due to the 2D projection used in the software's selection tools. Using 3D bounding boxes or other forms of cutting planes are inappropriate for any marginally complex building form. Specifically, the problem is difficult in that the errant points may be quite close to a valid surface, and the surface may also be concave.

The problem may be solved by creating a low-poly bounding polyhedron template for each building site, and using a "point-in-polyhedron" algorithm[18] that supports concave surfaces. This is the same as a "point-in-polygon" algorithmn except that axes are intelligently discarded for the test. Each point in the cloud can then be tested against the boundary. If the point lies outside the boundary, it is discarded.

No known software is available that provides this polyhedron containment functionality with any form of usable interface. We will therefore write our own. One library that provides an implementation of this polyhedron test is `MathGeoLib`[19]. Let's compile it:

```
$ git clone https://github.com/juj/MathGeoLib.git
$ cd MathGeoLib
$ cmake
$ make
```

Now we have the linkable static library `libMathGeoLib.a`. Now we will need to deal with two meshes: the point cloud and the bounding mesh (assuming both have been aligned to the datums properly). We can parse the `.ply` format for this.

The `.ply` format[20] is chosen to process for multiple reasons: it is the output of `pmvs2`, it is simple to parse, it explicitly supports custom metadata in the case of future customisation, it is the standard supported by the Standford benchmark, and originally designed for 3D scanning.

The chosen `.ply` parser is `libply`[21]. If `gcc` 4.2.0 is not available to compile it, I have patched it to compile it with 4.9.3, and included the patch with this document[22]. Now we can compile it:

```
$ wget http://people.cs.kuleuven.be/~ares.lagae/libply/ply-0.1.tar.gz
```

[18]`http://erich.realtimerendering.com/ptinpoly/`
[19]`http://clb.demon.fi/MathGeoLib/nightly/`
[20]`http://people.cs.kuleuven.be/~ares.lagae/libply/ply-0.1/doc/PLY_FILES.txt`
[21]`http://people.cs.kuleuven.be/~ares.lagae/libply/`
[22]`src/libply.patch`

```
$ tar -xzvf ply-0.1.tar.gz
$ cd ply-0.1/
$ ./configure --prefix=/home/foobar
$ make
$ make install
```

Note the location of the installed library files, as they will be needed to compile our point cloud cleaner.

It should be noted that the `.ply` parsing (and other functions) library used by `MeshLab` is VCG[23]. It may be beneficial to rewrite the parser using VCG to minimise file handling quirks. Usage of VCG is apparently straightforward with the inclusion of their header files (no apparent static or shared libraries), but I was unable to get the PLY importer headers to compile smoothly, probably due to the cygwin environment. If it is rewritten with VCG, their import class supports all major mesh file formats.

2.10.3 Automated cleaning optimisations

It should be noted that a convex hull test is much more computationally efficient. Similarly, the boundary test depends on both the number of faces in the bounding polyhedron and the number of points in the point cloud. If the need arises, there should be multiple passes: a roughing pass that uses convex hulls, and then a more detailed pass.

2.11 Surface reconstruction

As described, mesh output is usually desirable. Before the mesh surface can be reconstructed, Depending on the resolution and reliability of the point cloud, different surface reconstruction techniques may provide better results. Two are detailed below: *Poisson surface reconstruction* and *Metaball surface reconstruction.* Two other techniques which may be appropriate for open object scans are *Greedy meshing*[24] and *Delauney*[25].

2.11.1 Poisson surface reconstruction

Poisson surface reconstruction is incredibly popular with very good results. It is based off the Poisson equation, and the details are elaborated in the original paper[26] by Microsoft. It is the de-facto reconstruction method for reliable point clouds which represent closed objects.

`MeshLab` provides

[23]`http://vcg.isti.cnr.it/vcglib/index.html`

[24]`http://www.pointclouds.org/documentation/tutorials/greedy_projection.php`

[25]`http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Triangulation_3/Chapter_main.html`

[26]`http://research.microsoft.com/en-us/um/people/hoppe/poissonrecon.pdf`

2.11.2 Metaball surface reconstruction

2.11.3 Headless surface reconstruction

2.12 Texture mapping

2.12.1 Axis mappings between formats

3 Video

3.1 Panoramic video

Panoramic video creation is essentially the same as panoramic images except that they are performed for each video frame. Two regular cameras are placed angled apart so that there is an overlap in the field of view with shared feature points. Common feature points in each frame are detected, aligned to determine perspective rules, and the resulting template mask is applied to merge the two images together. This template mask generation does not have to occur for each frame if the rig is stationary — it is calculated once at the beginning and reused. Once the images are stitched, they may be reprojected using a variety of projection types.

The fundamental libraries used by most panoramic wrappers is `Panorama Tools`[27], which contains a suite of utilities, similar to how `imagemagick` is packaged. Panoramic stitching is not a perfect science, and so ideally requires visual confirmation. For this, `hugin`[28] is recommended as a cross-platform GUI wrapper.

3.1.1 Synchronisation of video

A loud sound, or alternatively a motion which is within the field of view overlap may be used to synchronise the start times of the video.

3.1.2 Headless panoramic

It is possible to merge the photos completely headlessly. It is assumed that the two footages are already converted into individual image frames with the same frame rate, and are synchronised so that c̃amera1/n.png is captured at the same timestamp as c̃amera2/n.png.

Once the frames are prepared,

```
pto_gen -o project.pto camera1\0001.png camera2\0001.png
pano_modify project.pto --canvas=AUTO
cpfind -o project_cp.pto project_mod.pto
autooptimiser -o project_align.pto -a -l -s -m project_cp.pto
./process_frames.sh
```

[27]http://panotools.sourceforge.net/
[28]http://hugin.sourceforge.net/

process_frames.sh does the actual stitching, and is a simple bash loop that goes over all frames.

```
#!/bin/bash
for f in camera1/*.png
do
    name=‘basename $f‘
    nona -o output/$name -m PNG project_align.pto camera1/$name camera2/
        $name
done
```