Frame Filtering for Videogrammetry

Creating 3D models of an object by filming it

Benjamin PAULMIER
MA-IRIF · Computing Project (PROJ-H402) · Université Libre de Bruxelles
benjamin.paulmier@ulb.be

Abstract **- Photogrammetry is a technology used to measure distances and build 3D models from photographs of an object. Although there are many software for photogrammetry, most do not accept video as an input. Shooting a video to capture frames of an object under all its angles can be more convenient and less time-consuming for a user than taking dozens of pictures around it.**

However, a good photoset for photogrammetry must follow precise criteria in order to work properly. The input images must be in-focus and there must be enough overlap between consecutive frames in order to create an accurate model. Given a video of an object, how can we extract frames and select the most appropriate ones to build a 3D model?

This paper describes an algorithm that was developed to extract and filter frames from a video according to the criteria of photogrammetry. The imaging library OpenCV was widely used to filter out blurry or unnecessary frames and make sure there is enough overlap between successive frames.

As a result, the filtering algorithm produced satisfying results, yielding much better-quality models and in a shorter time than a simple regular selection of frames showcasing the object under all angles. The filtering time accounted for a very small fraction of the 3D modelling time. Testing the algorithm on larger datasets could be needed in order to upgrade it further.

Keywords – video frames, bluriness, 3D modelling, laplacian, bounding rectangle, overlap, homography matrix.

I. INTRODUCTION

Videogrammetry is a technology in which images from a video are taken to calculate the three-dimensional coordinates of points on an object. [banana dense point cloud and banana model). The term 'videogrammetry' is an extension to the principle of photogrammetry: the process of using distinct images of an object to measure distances and determine positions thanks to triangulation. By taking photographs from at least two different locations, the lines of sight can be developed from each camera to points of the object. By intersecting them, we can produce 3D coordinates of the points of interest [1].
A prime example of triangulation use is is the way two eyes work together to gauge distance and see depth.

There are plenty of photogrammetry software available today - including free and open source ones - that take photographs of an object under different angles as an input, and that create a 3D model of said object as an output (which can be used for 3D printing, for example). These models can be very precise and true to the original as long as the given photographs meet the following criteria [2]:

- Good lighting. Hard shadows and uneven lighting are hindering factors because they do not provided many useful features that can be matched for reconstruction.
- No glossy or transparent effects as they make it more difficult to track points of an object across several photos. Spray painting the transparent/shiny areas is a solution.
- Enough overlap on the different photos. To estimate the points' 3D coordinates accurately, many different angles are required: at least 60% of side overlap and 80% of forward overlap between two successive camera positions is needed.
- No moving the object of interest. Its relative position to the background is primordial for reconstruction. Movement of people in the background is also unwanted.
- Fixed lens (no focal change).
- Object of interest stays in the middle of the picture and fills a large part of it. This will result in a more detailed 3D model.
- Good photo quality. Blurry pictures will confuse the feature matching algorithm. It is recommended to shoot with cameras that have at least 5 Mpx resolution.

The objective of this project is to build an algorithm that transforms a video into the ideal set of frames for photogrammetry: extract all frames, remove the blurry or unusable ones, make sure there is enough overlap and keep the final number of frames below a certain value specified by the user. If too many images are given to a photogrammetry software, computing time can be very long.

This frame filtering algorithm lets the user decide on multiple parameters like the desired amount of overlap, number of frames to be kept and the threshold above which a frame is considered blurry. This allows the users to generate photogrammetry-ready datasets that are appropriate for the power of their machine, as most photogrammetry software do not accept video as an input or require very large amounts of computational power to do so. This project will enable users to film an object, moving around and over it, and keep the best frames to make a 3D model out of it in a reasonable amount of time.

II. Material Used

Below is a table showcasing the material that has been used during the development of this project.

Hardware	Programming language and IDE	Imported Librairies	Photogrammetry software

MacBook Pro
2.3 GHz processor
8 GB RAM
No CUDA
Image acquisition :
12 Mpx camera, 30 frames per second
	Python
	Spyder 4.0.1
		`cv2`
		`os`
		`shutil`
		`numpy`
		`time`
			Agisoft Metashape (free version)

Fig. 1 - Material

`cv2` is a library for image processing, which contains a great number of functions used for frame filtering throughout the project. The version used for this project was OpenCV 3.4.2.

III. Methods Tried and Used

The developed main algorithm (`extractFrames` in `projh402.py`) takes as input the name of the video on which the user wants to work, a blur threshold above which the user considers a frame to be blurry, an overlap threshold (that determines how many overlapping frames the filter will keep) and the maximum number of frames the user wants in his final, photogrammetry-ready folder. It creates a folder with the right frames, and returns the number of seconds taken to do so.

The algorithm created goes through the following steps, whose purpose and implementation will be detailed later:

- creation of a folder in which video frames will be extracted.
- look at each frame, keep the non-blurry ones in said folder.
- go through the folder once more, eliminating frames where there is too much overlap (unproductive) or return an error message if there is not enough overlap. After this step we are left a folder of good frames, usable for photogrammetry.
- if the number of good frames we are left with is still too high, evenly remove some more.

Before defining the `extractFrames` function, the file contains a long comment giving the user instructions on how to shoot videos efficiently for videogrammetry: having a well-lit environment, no hard shadows, no glossy or transparent effects, no focal change, no moving the object of interest, and trying to keep it in the center of the frames whilst not moving around too quickly to avoid too much blurriness.

A. Filtering out blurry frames

The first step of the algorithm consists in the creation of a folder in which every frame will be extracted and analysed. Using OpenCV's `VideoCapture` function, each frame is written to this new folder and is kept if and only if it is considered not blurry. Getting an accurate estimation of the blurriness is done rather quickly by using the Laplacian operator on the image [3]. It is generally used to compute the second derivative of an image, therefore it will highlight the rapid intensity changes across the image.

If the Laplacian of the image has a high variance, then it has a wide spread of responses, corresponding to both edges and no edges, which is representative of a good, non-blurry image. On the other hand, if its variance is low, there are very few edges in the image, characteristic of a blurry frame. A quick estimation of an image's blurriness is thus obtained by computing the variance of the Laplacian of the image. This will return a floating point value generally between 20 and 1000. Testing this method on a collection of images showed a good threshold to determine whether an image is blurry or not is 150 for the resolution used during this project (1920x1080).

However, this value (`blurThresh`, defined by the user in the algorithm) can lead to discarding perfectly good frames: if one region in the background contains large amounts of blur, an image can be considered 'blurry', but the region containing the object of interest might contain a lot of useful information. To make sure that such frames don't get discarded, the algorithm computes the variance of the Laplacian of the bounding rectangle of the object of interest, as illustrated below.

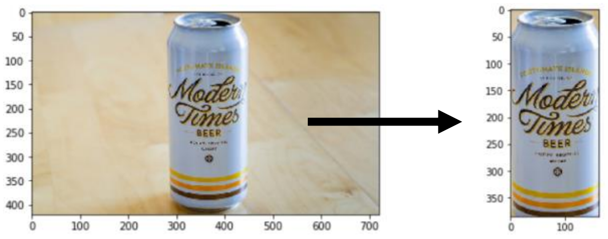


Fig. 2 – Example of a bounding rectangle of an object

Obtaining this result was done by using OpenCV's canny edge detection on the original image (after having converted it to grayscale to save time) followed by the `findContours` function. Once the contours are highlighted, the image is cropped to a rectangle going from:

- the utmost to the downmost point of 'sharp contour' vertically,
- the leftmost to the rightmost point of 'sharp contour' horizontally.

The obtained image might contain more than the object of interest. If the background is contrasted, the bounding rectangle algorithm will find contours located further than the object of interest and therefore include a larger region of the image, a problem we cannot encounter if the background is uniform.

One solution is to apply a gaussian blur first, to smoothen out unwanted contours in the background, leaving the object-background boundary as the only sharp contour left, and thus the only contour taken into account by the algorithm. With highly contrasted backgrounds, heavy smoothening might be necessary to erase the background contours, but it can also remove the actual border between background and object.

To solve this issue, it was chosen to apply a mild gaussian blur, which at best (when the background isn't too contrasted) successfully creates a bounding rectangle containing the object of interest only, and at worst creates a rectangle containing the whole image. This scenario isn't too harmful, considering frames with rich, diverse background will bring lots of information for photogrammetry. In addition, a 'useless computation' of the contours will not waste much time: about 25 milliseconds per frame. Even with 10-minute-long video shot at 30 fps, computing the bounding boxes would take about seven minutes, which is insignificant compared to the time to build a 3D model with such large amounts of data. These values were found using the `boundRectTest.py` file.

This bounding rectangle function will generally crop out potentially blurry regions on the sides of frames which aren't necessarily bad for photogrammetry, as long as the main region of interest of the image is clear.

If the cropped image passes the `blurThresh` chosen by the user, the original frame (uncropped) is kept in the folder, otherwise it is deleted. At the end of this first round of frame filtering, if there are less than 30 frames in the folder (too few for photogrammetry), the algorithm will return an error message notifying the user that the input video is too blurry. Otherwise, the algorithm continues to the second filtering step which involves overlap.

B. *Overlapping frames*

Many techniques to calculate the side and forward overlap were considered. Computing the difference or the optical flow between two consecutive frames (as shown in *Fig. 3*) gave hardly exploitable results.

Optical flow draws the movement of certain points of the object between two images. By taking into account which translation was the largest, we can estimate the overlap percentage, both horizontal and vertical. However, in order to obtain accurate results, the compared frames must be taken at the same distance from the object, which can be difficult to achieve when filming a video.

The scale of the object of interest remains approximately the same but small changes of distance can impact the results quite heavily, and scale change-aware optical flow algorithms are computationally expensive [4].

On another note, converting the frames to binary images can be useful to compute overlap, however in our case it would mean important loss of information.

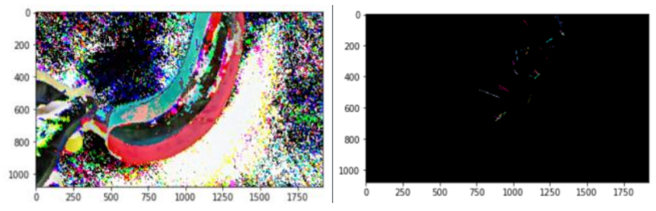


Fig. 3 - difference and optical flow of two frames. The object (banana)'s position was changed slightly.

Instead, homography matrices between consecutive frames were used. The homography matrix between two frames can be used to compute rotation and translation between two images [5]. It is computed by tracking certain matching features between the two images.

The 'overlap filter' of the algorithm works as follows:

- if three consecutive frames have more overlap than needed (according to the user-set `overlapThresh` parameter), the frame in the middle is discarded.
- if the algorithm fails to find sufficient overlap between a certain number of consecutive frames, an error message is returned, notifying the user that the non-blurry frames do not have enough overlap.

Between each pair of successive frames, the overlap percentages, horizontal and vertical, are computed by the `overlapPercentageFLANN` function, also defined in the `projh402.py` source code file. This function uses the translation-related values of the homography matrix to compute overlap. If it fails to find the homography matrix, the consecutive images are considered not similar enough and the counter of insufficiently overlapping photos is incremented.

To compute the homography matrix, two different methods were tried: using OpenCV's SIFT method for feature detection and BFMatcher to do the matching, and using the SURF and FLANN methods for the same purposes. Calling these functions sets up the use of the `findHomography` function on the points that were detected [6]. The SURF-FLANN method proved to be accurate and much faster for a wide variety of test videos and was therefore kept in the final program.

Indeed, SIFT is known to be better than SURF mostly in different-scale images [7]. Two consecutive frames will most probably be of similar scale, sometimes not enough to make for accurate optical flow computation, but sufficient for SURF feature detection. Additionally, FLANN is known to be faster than BFMatcher for large datasets [8].

Once again, if not enough good frames remain, the program invites the user to take a new video or set the blur or overlap thresholds higher.

C. *Final arbitrary filtering and comments*

If the number of remaining frames is higher than the user's specified amount, the algorithm goes through a final step of filtering, evenly removing frames throughout the whole selection until this requirement is met. Filtering frames on quality once again would be unproductive, as all the remaining frames are already good and going further would be computationally expensive.

On a side note, a 'good lighting' filtering was considered but has finally not been taken into account, as it adds a very high number of computations and makes the program much slower. The algorithm can overlook whether images are evenly lit or not by warning the user not to shoot videos in environments with hard shadows and by recommending the conditions described in the ruleset at the beginning.

D. *After running the frame filtering program*

A folder named "good[videoName]Frames" has been created and is ready to be used in a photogrammetry software. Agisoft's Metashape was used during this project. All that remains to be done is open the folder in Metashape, align cameras, create a dense point cloud then getting the final 3D model.

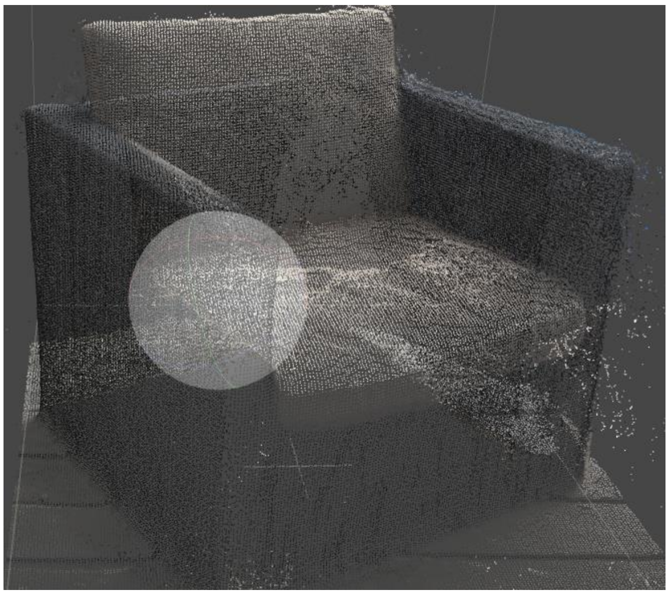


Fig. 4 – Dense point cloud representation

IV. RESULTS

The algorithm has been tested on a dozen of videos, with different values given as blur and overlap thresholds. The videos were shot with a mobile phone camera, filming 1920x1080 frames at 30 frames per seconds. Under Metashape, the parameters used were always the same: medium quality for camera alignment, low quality for dense point cloud, high face count for mesh and texture creation.

Concerning the constants used throughout the algorithm (see `projh402.py` at the end of the article), they were chosen after testing on a variety of images and following recommendations on the OpenCV documentation [9].

On the next page is a table comparing the frame filtering algorithm with an algorithm extracting frames regularly without looking at their quality (source code in `regExtraction.py`, this algorithm will later be referred to as 'naïve algorithm'). The comparison is made on quality and time-related criteria. Unfortunately, the computer used for the project couldn't use Metashapes' automatic treatment for video input. The expected number of frames kept for photogrammetry was between 80 and 150 for each video and each method, to constitute a fair comparison.

Video characteristics	**Quality of frames extracted**	**Time to extract frames**	**Time to build 3D model**	**Quality of the final 3D model**
23 second-long video of a chair shot from 2 meters away. Not very blurry	**FF:** Good **RF:** Uneven, some good frames and some blurry ones	**FF:** 143s **RF:** 7s	**FF:** 9 min **RF:** 12 min	**FF:** Excellent, very close to reality **RF:** Low, presence of waves and imperfections
12 second-long video of a carafe shot from 1 meter away. A bit blurry, and small amounts of shade	**FF:** Average (the blur threshold had to be set low, at 50) **RF:** Poor, many blurry frames	**FF:** 72s **RF:** 8s	**FF:** 7min 30s **RF:** 15 min	**FF:** Medium, shadows in the original video causing imperfections **RF:** Low, parts of the background merged with the carafe
6 second-long video of a carafe shot from 1 meter away. Very blurry, and small amounts of shade	**FF:** None, input video quality was too poor **RF:** Very low	**FF:** N/A **RF:** 5s	**FF:** N/A **RF:** 10 min 10’50” vs 3’52”	**FF:** N/A **RF:** Very poor
18 second-long video of a pear shot from 50 cm away. Not very blurry, moderate amounts of shade	**FF:** Average because of shadows **RF:** Uneven	**FF:** 81s **RF:** 8s	**FF:** 3min 50s **RF:** 11 min	**FF:** low, too much shade **RF:** low, too much shade

Fig. 5 - Performance comparison. FF = Filtered frames, RF = Regularly extracted frames.

Generally, the frame filtering algorithm takes about 5 or 6 times the duration of the video to produce an output, and the result can build 3D models twice as quick as a regular frame selection, these models being more accurate in addition.

Fig.6 shows some screenshots of the carafe model to showcase the differences between the methods used.

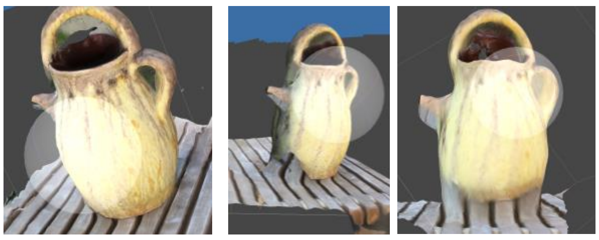


Fig. 6 - Models produced with the frame extracting algorithms. From top to bottom, clockwise: good video and filtering algorithm, good video and naive algorithm, bad video and naive algorithm. The results are decreasingly sharp.

A relevant comment is that on all of the tests that were made, the time spent to extract and filter good frames represented approximately one fifth of the whole 3D model building process time, with that number dropping below 1% if the settings of the photogrammetry software are set higher.

This algorithm can thus be considered as a practicable option to create 3D models from videos, its computing time being acceptable in comparison to the actual modelling. The user can choose the quality of the frames extracted allowed by his or her camera thanks to `blurThresh` (the lower it is set, the lower the quality) and the number of frames wanted with `nbrOfFrames`.

Regardless of the number of frames chosen, the result of the filtering algorithm will generally be better than that of the naïve algorithm, as highlighted by *Fig. 8*. These results were obtained by comparing both algorithms on the same input video, 19 seconds of filming a chair rather steadily. 'TCP' refers to 'total computing time' (frame extraction and 3D modelling), 'Q' stands for quality.

The quality of the final 3D model built with the naive algorithm is often very close to the one obtained with the filtering algorithm when asking for half as many frames, thus taking half as much time to compute.

The final row of the table shows that 20 frames can hardly produce a correct model, justifying the threshold in the filtering algorithm that is responsible for stopping computations and giving an error message when the number of frames after a given step of the filtering is too low.

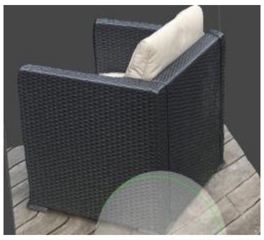
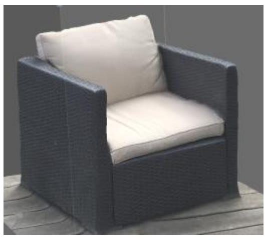
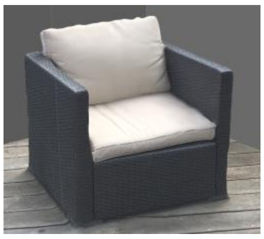
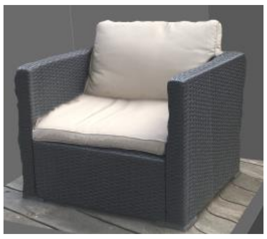
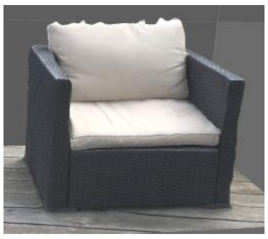
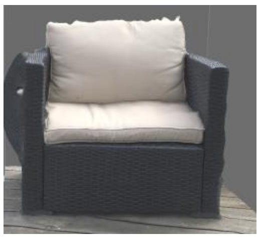
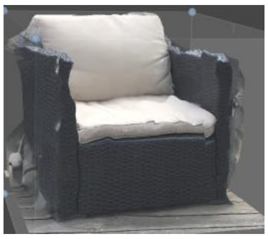
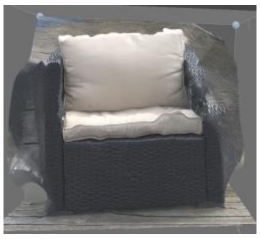
Frames	**Filtering algorithm**	**Naive algorithm**
200	TCP: 17 min Q: Excellent 	TCP: 15 min Q: Very good 
100	TCP: 7 min 30s Q: Very good 	TCP: 6min 30s Q: Good 
50	TCP: 4 min Q: Good 	TCP: 2 min Q: Low 
20	TCP: 3 min Q: Very low 	TCP: 1 min Q: Very low 

Fig. 7 – Algorithm performance given the number of frames used for photogrammetry

V. Conclusion

Photogrammetry is a technique that requires a precise dataset in order to function properly. For casual use, taking a variety of individual photos can prove to be tedious and memory hungry: stopping at multiple angles to take a proper photo is time consuming, and a single photo from a phone camera, if imported uncompressed, can weigh up to a couple of megabytes.

Importing a video, which requires much less effort to shoot and weighs about 20 megabytes for a few hundreds of frames, is an appealing option to get images of the object of interest. However, since photogrammetry is very expensive computationally speaking and requires high quality images, it is necessary to select only the best frames from the video before feeding them to a 3D modelling software.

The algorithm created in this project filtered out all blurry and unnecessary frames whilst assuring the working conditions for photogrammetry are met: sufficient overlap and frame quality. Mathematical objects like the variance of the Laplacian of an image and homography matrices between different images were used to evaluate blurriness and overlap, thus allowing the program to keep the best frames only.

When comparing the algorithm's performance with a simple regular selection of frames throughout a given video, it was highlighted that better-quality models are created, and in a shorter time, when frames are filtered according to their quality. A rule of thumb is that the filtering algorithm usually requires twice as few frames to produce same-quality models.

During the creation of a 3D model using this algorithm, the time taken to filter out images only accounts for approximately 20% of the entire computing time, and this number can drop to under 1% if the reconstruction settings of the photogrammetry application are set much higher.

Using a video and extracting frames using the method described in this article can therefore be considered as an interesting, ergonomic and user-friendly way to quickly create 3D models of objects.

More computational power to test the algorithm on much larger amounts of videos would have been a way to find out how to upgrade the algorithm more, highlighting which types of videos need to be addressed and giving a better idea of the algorithm's performance. However, due to the Covid-19 pandemic computer resources at the university were unreachable and all of the testing had to be done on a personal computer. Further testing could prove to be useful to bring positive changes to the algorithm.

Testing the algorithm with different cameras could also yield more recommended values for the blur threshold set by the user. 150 was chosen here for a 12 Mpx camera after trying out multiple values on different input videos, but higher resolution cameras would most likely allow the user to set that threshold higher. Testing with aerial videos of much larger objects also remains to be done.

To close this article, it would like to thank the Laboratory of Image Synthesis and Analysis of the Université Libre de Bruxelles, project coordinators Olivier Debeir and Arnaud Schenckel for offering such an interesting project. A link to the GitHub repository containing the source code for the algorithms is available after the 'References' section.

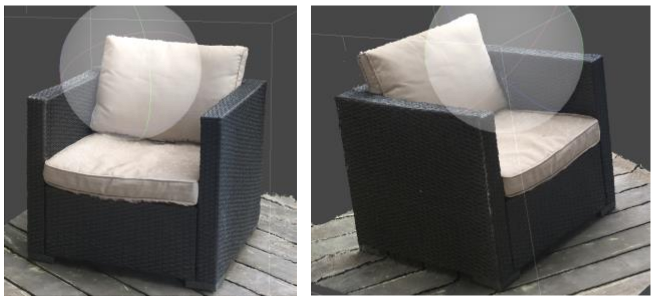


Fig. 8 - Screenshots of the 3D model of a chair obtained with the filtering algorithm

References

[1] GIS Resources, "What is photogrammetry ?", Jan. 2014

[2] Mitko VIDANOVSKI, "What makes good photos for photogrammetry ?", Nov. 2014

[3] Adrian ROSEBROCK, "Blur detection using OpenCV", Sept. 2015

[4] Li XU, Zhenlong DAI, Jiaya JIA, "Scale Invariant Optical Flow" - Dept. of computer science and engineering, The Chinese University of Hong Kong

[5] Ezio MALIS, Manuel VARGAS, "Deeper understanding of the homography decomposition for vision-based control" - RR-6303, INRIA. 2007, pp.90.

[6] Satya MALLICK, "Homography examples using OpenCV", Jan. 2016

[7] Mistry, Dr & Banerjee, Asim, "Comparison of Feature Detection and Matching Approaches: SIFT and SURF" - GRD Journals- Global Research and Development Journal for Engineering. 2. 7-13, 2017

[8] Unknown, Feature Matching

[9] OpenCV documentation

GitHub repository Link

The GitHub repository containing the files `projh402.py`, `regExtraction.py`, `boundRectTest.py` and a .psx file (to be opened with metashape) can be found here.