Fixed Single–Camera 3D Laser Scanning

Giulio Zausa
Ca' Foscari University of Venice
870040@stud.unive.it

In this report we will discuss how a low–cost 3D Point Cloud can be constructed using a line laser and a fixed viewpoint single RGB camera, building a full reconstruction pipeline using OpenCV and Python, describing the techniques used for camera calibration, laser detection and point cloud reconstruction. We will also talk about the performance of the proposed model, aiming for real–time operation speed.

I. Introduction

The problem of reconstructing a 3D Point Cloud of a solid object has many applications, ranging from entertainment to manufacturing defects detection. This is a well established research topic since decades, and industrial applications are in use since decades. Nonetheless, it's still interesting and useful to research faster and more precise ways for scanning objects using commodity hardware, such as low–res cameras, paper printed markers and cheap lasers. We will see which kind of results can be obtained using low–cost off–the–shelf components.

This project consists in the development of a 3D Laser Scanning software, using a given dataset made by pre–recorded footage and calibration images. Our goal is being able to reconstruct a colored 3D Point Cloud from a series of video frames, capturing the object from a single fixed point of view, with a laser line pointer moving over it.

A. Related Work

Many approaches have been proposed to scan 3D surfaces using structured light, such as [1], [2], [3], [4]. The approach presented in [1], for example, is similiar to ours, since it uses the shadow cast from an handheld sweeping stick. This paper presents an alternative way to perform the same task using an active handheld laser scanner and other easy to find components, while preserving good accuracy and fast processing speed.

B. Input Dataset and Imaging Setup

We will test our scanning software using a pre–recorded dataset, given by the course professor, consisting of 50 calibration images and 4 laser scanning videos. We don't have any prior knowledge about the camera, so we need to infer as much as possible from the calibration images.

The calibration images (fig 2) were taken using the same camera used for the video footage, and they capture the same calibration pattern from different view points, with a resolution of 1296×972 pixels. The calibration pattern fig 1 (included in the project code folder) is made by a chessboard pattern with a missing tile to detect the correct orientation. The chessboard pattern is enclosed in a black outlined rectangle. We will see later how this data allows us to infer the camera projection intrinsics and distortion parameters.

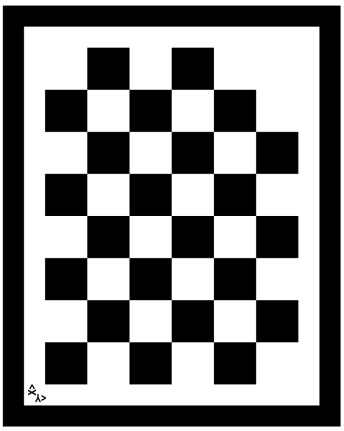

Figure 1: Our printable calibration pattern

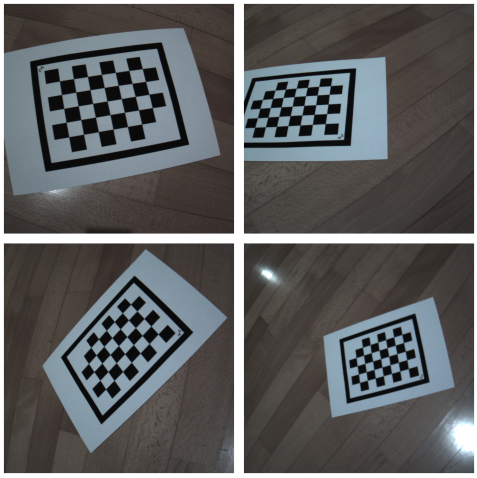

Figure 2: An example of four out of the 50 images used for camera calibration from our dataset

The video frames used for 3D reconstruction are captured with the setup shown in fig 3:

- Two perpendicular printed paper **markers**, with a 23x13 cm rectangle *(printable pdf can be found on the code repository)*;
- A fixed view point RGB 1296 × 972 **camera** *(unknown model)*;
- A red **laser** line pointer, capable of intersecting both markers and the object;
- A 3D **object** to scan, positioned between the two markers.

We can see that all the hardware components are very cheap and easy to find, unlike more costly industry standard solutions. Also, our vision system is monocular: we don't need a stereo calibration, precise positioning and video synchronization.

Figure 3: The setup used for video capture

The given dataset is made by 4 videos for a total of 234 seconds (15 FPS, 990 frames total). All the videos use the same setup and capture several laser sweeps done by hand, after a few seconds without the laser in view. We will see how the laser intersection with the object and the markers can help us reconstructing a 3D Point Cloud of our object.

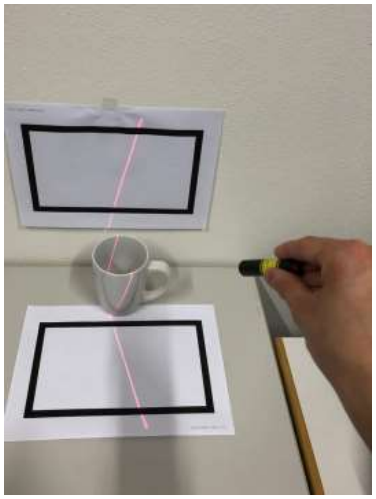

Figure 4: The laser sweeping in action

II. Reconstruction Method

With the given dataset, the first high–level task we try to solve is finding a way to infer depth from the video. The video frames are 2D, they have no explicit depth data, but we need it to build our point cloud. Since the picture is monocular and the camera and the object don't change during the video, the only information we can exploit to infer depth is the movement of the laser that scans the object.

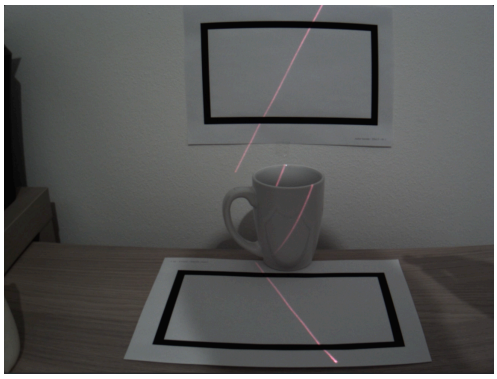

Figure 5: An example frame from the laser scanning video dataset

Looking at the scanning setup, we can identify 3 intersecting planes in three dimensions (fig 6):

- The **ground plane** (δ), identified by the black border of the printed rectangle;
- The **upward plane** (γ), identified by the black border of the other printed rectangle;
- The **laser plane** (μ), identified by all the points that are lit by the laser.

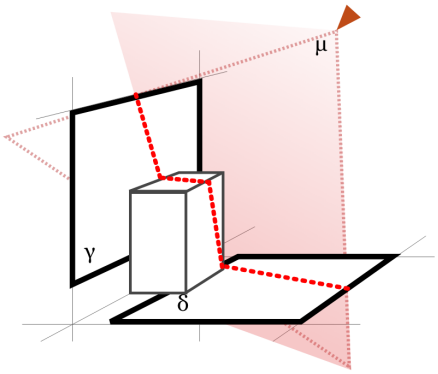

Figure 6: The three planes formed by the markers and the laser

In our video the laser plane intersects the two rectangle markers and the scanning object in several points. We know a 2D projection of these intersections points from the video frames, since the camera can capture the laser light and the surfaces can reflect it towards the sensor. We can try to use this information to infer the 3D position of every point.

We can describe the equation for a projected intersection point p using homogeneous coordinates and the equation for a pinhole camera ([5]). In this equation K is the intrinsics camera parameters matrix, $R\,T$ represents a rigid motion specific for the camera viewpoint (extrinsic parameters) and our (x, y, z) vector is the position of the projection point in world coordinates. We convert our 3D point into an homogeneous 4D vector appending a 1 at the end to perform the projective transformations.

$$p = \mathrm{K}\,(\mathrm{R}\;\mathrm{T}) \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}$$

Once we have the 4D homogeneous projected points, we can convert them back into cartesian coordinates by dividing by the last w component. This way we can establish a relationship between real–world positions and pixel positions captured by the camera. To solve our problem, however, we need the inverse relationship, since we need to infer 3D points from 2D positions to build our point cloud.

Unfortunately if we try to invert the the relationship, even ignoring the $R\,T$ factor, we end up with missing unknowns, since for every pixel in pinhole camera we have infinite line (fig 7) of corresponding points. If we consider x as our normalized 2D screen position, K^{-1} as the *pseudoinverse* of K and C its null–vector [6]:

$$\mathrm{K}\mathrm{K}^{-1} = \mathbf{I}$$

$$\mathrm{K}\mathrm{C} = \mathbf{0}$$

$$X(\lambda) = \mathrm{K}^{-1}p + \lambda \mathrm{C}$$

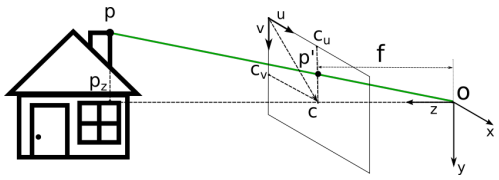

Figure 7: For every pixel in our image, we have an infinite line of corresponding points

Since we have no way to know the λ value directly, to accomplish our task we need to rethink our problem considering also the three planes defined earlier. In fact, we can see that all the backprojected 2D points from the 2D image, using the intrinsics matrix K, intersects the laser plane μ in exactly one 3D point (fig 8). So, if we have a laser–lit point in our picture, and we know the parameters that define the laser plane μ, we can solve the system and determine the precise 3D position of our point.

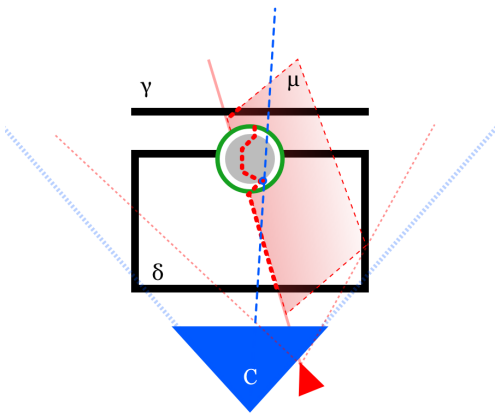

Figure 8: Intersection between the camera ray (in blue) and laser plane (in red)

Since for every laser–lit pixel we have a ray of points (as described earlier), the 3D point determination can be done using the plane–line intersection formula:

$$d = \frac{(\mathbf{p_0} - \mathbf{l_0}) \cdot \mathbf{n}}{\mathbf{l} \cdot \mathbf{n}}$$

Using this method, everytime the laser scans the surface of the object, sweeping back and forth, it gives us information about a slice of the surface, as a set of 3D points. Doing this for every frame we can reconstruct a 3D Point Cloud of our object. This approach is similar to the Structure from Motion method (SfM), even though we are moving the laser instead of the camera.

A. Recovering the laser plane

We have seen how the laser plane can be used to infer the 3D position from an image point backprojection, using the point–line intersection formula. Before

doing so, we need to find the equation of the laser plane frame by frame.

Looking at our scanning setup (fig 6) and at one sample frame (fig 5), we can see how our laser plane intersects almost always the two reference rectangles. Knowing that a plane can be defined by at least three non aligned points, we can use the laser intersection with the reference rectangles to fit a plane. In fact, the laser projection on the lower and upper reference rectangles will always produce three non aligned points, since the two rectangles are perpendicular.

With this method we can easily find the laser plane equation, fitting all the backprojected laser–lit 3D points inside the reference rectangles. To perform the fitting we have used a least–squares method as described in *Least Squares Fitting of Data by Linear or Quadratic Structures* [7]. Our error function for least–squares minimization is the following:

$$E(a_0, a_1, b) = \sum_{i=1}^{m} [(a_0 x_i + a_1 y_i + b) - h_i]^2$$

To find the 3D points to fit we need to backproject them from the 2D image space. To do so we need again to use a plane–line intersection, since they intersects the reference rectangles, but this time we have an easier way to find our plane equations. Since we know the size of the inner black rectangles and the intrinsic parameters of the camera, we can use an homography and a decomposition to find the origin and the normal of our plane, knowing the 2D position of the rectangle corners.

Supposing we have computed an homography matrix $\mathrm{H} : p \rightarrow p'$ that maps points from reference rectangle coordinate space to image space and K camera intrinsic parameters matrix, we can compute our plane by multiplying the homography with the inverse intrisic matrix K^{-1}.

$$\mathrm{R_1|R_2|T} = \frac{\mathrm{K^{-1}\,H}}{\|\mathrm{K^{-1}\,H}\|}$$

$$\mathrm{R_3} = \mathrm{R_1} \times \mathrm{R_2}$$

This way we have computed the three basis vector $\mathrm{R_1}$, $\mathrm{R_2}$ and $\mathrm{R_3}$, which can be used to compute the plane equation. Before doing that we need to use the Zhang approximation ([8]) to get an orthonormal basis for the rotation: we calculate the singular value decomposition of the matrix $(\mathrm{R_1|R_2|R_3})$ and keep only the U and $\mathrm{V^*}$ part. The origin of the plane will be the translation vector found before, while the normal can be calculated using the singular value decomposition values:

$$\mathrm{R} = \mathrm{U} \times \mathrm{V^*}$$

$$\mathbf{p_0} = T$$

$$\mathbf{p} = \mathbf{R_2}$$

III. Implementation Details

Our reconstruction software is made by two different Python scripts, one for performing the camera calibration to recover the intrinsics and distortion parameters for our camera and another for performing the real scan, taking as input our calibration and a video file.

The scripts are available in our GitHub repository.

Both the scripts are using the OpenCV library to perform common computer vision tasks, for speeding up execution and development.

A. Camera Calibration

As we have seen before, to reconstruct our 3D points we need to recover first the intrinsics matrix K of the camera. We have no prior info about the

matrix, but we can reconstruct it from the calibration images (fig 2). Since the camera also presents radial distortion (fig 9) we need to rectify the frames before performing any measurement. We assume a 5–parameters polynomial distortion model.

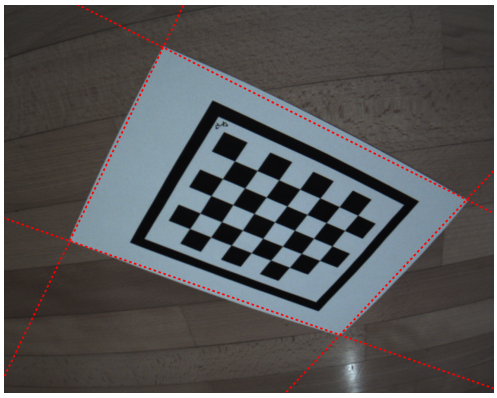

Figure 9: The dataset frames presents radial distortion

Using the method described in "A Flexible New Technique for Camera Calibration" ([8]) we can infer the intrisics matrix and the 5 distortion parameters. In our tests we used the algorithm implementation provided by the OpenCV library (`calibrateCamera()`).

To calibrate the camera we used all of the 50 calibration images provided in the dataset, matching the position of the checkerboard corners with the expected coordinates in the reference marker space. To find the chessboard corners we had good results using the `cornerSubPix()` function from OpenCV, done as descripted in fig 10. The main idea is to crop the black rectangle, exploiting the original pattern and image capture similarity. We suppose the chessboard corner points are in the same position of the original pattern and we use a sliding window to optimize the image point positions with subpixel accuracy. The subpixel corner refinement algorithm is described in *"A fast operator for detection and precise location of distincs points, corners and center of circular features"* ([9]), and aims to find the saddle points of the gray image.

After the calibration process we found the following approximate intrinsic matrix ($\mathbf{K}$) and distortion parameters $\mathbf{d}$.

$$\mathbf{K} = \begin{pmatrix} 1449.060927 & 0 & 611.9387293 \\ 0 & 1483.460784 & 392.2107774 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{d^T} = \begin{pmatrix} -2.4568613490840346e-01 \\ -3.2570741561031419e-01 \\ 8.9675767947581340e-03 \\ 1.9515423269787597e-03 \\ 2.7000527738886521e-01 \end{pmatrix}$$

To validate the quality of the found parameters we calculated the mean reprojection error, that scored 1.0079594964888223, which we found it's good enough to perform some basic 3D reconstruction.

B. Laser Detection

As we explained before, to calculate the 3D points we need a way to find the 2D positions of the laser–lit points in the image, to perform the fitting of the laser plane and to calculate the backprojected rays that will intersect the laser plane. To do so we experimented with a lot of different image processing techniques, and with each one we had different results. To evaluate the performance of each method we compared the found pixels with the expected one and the amount of outliers. The reason why we need to minimize the outliers it's because we found out they introduce a lot of noise in the resulting scan, and most importantly they cause errors the laser plane parameters fitting which precision is fundamental to get a good scan.

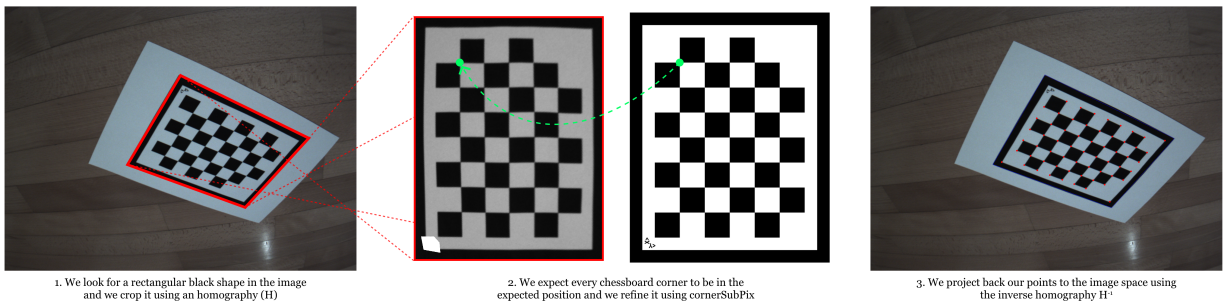

Figure 10: The algorithm used to find the chessboard corners

We found the best results (fig 11) using a simple HSV range thresholding followed by a morphological opening with a square 4×4 kernel. The reasoning behind this method is that the laser–lit pixels have a specific constant hue and saturation that isolate them from the rest of the picture. The morphological opening allows to remove some outliers to have a better result. The HSV ranges picked for the dataset are $H : [150, 200]$, $S : [20, 255]$ and $V : [78, 255]$.

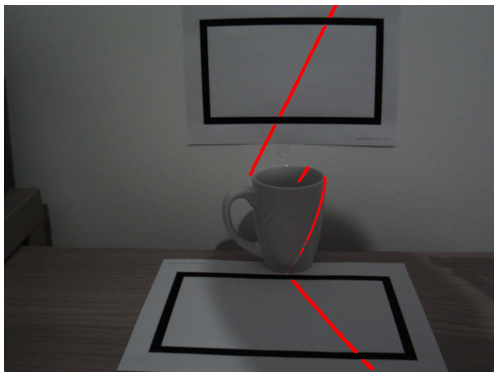

Figure 11: The points (in red) found by the laser detection algorithm result using HSV range thresholding and morphology operators

Since a lot of lower–lit points are missed by the algorithm we evaluated two improvements for the algorithm:

- **Background Subtraction**: we tried to exploit the fact that the camera is still to increase the contrast of the laser–lit points. To do so we subtracted the first frame (that has not any laser point) from all the other frames, but unfortunately this made the detection worse, since it damages the precision of the RGB to HSV algorithm (fig 12);
- **Clustering**: to reduce the amount of outliers given by the widening of the HSV threshold range we tried to use the *DBSCAN* [10] clustering algorithm, even though we found out it made our results worse, since a lot of laser points does not make dense clusters.

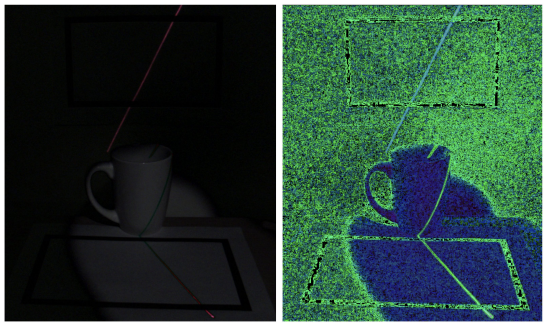

Figure 12: A background–subtracted frame and its corresponding HSV conversion

C. Points Reconstruction

After we have detected out laser–lit points, we can perform the geometric part of the process as described previously. We described an outline of final

algorithm in Alg. 1. A more precise implementation can be found on the paper's code.

The output of the algorithm tested on the file `cup1.mp4` from the dataset can be seen in fig 13. Since we have no ground truth Point Cloud data we can't make any measurement about its quality, but we can see from the results how it can scan in a realistic way the convex parts of the objects. We see, however, how it fails to scan all the parts where the laser isn't bright enough, including obviously the parts where the laser has not reached the surface or is not visible from the camera point of view.

Algorithm 1: Scanning process

video = openVideoCapture(filename)
K, d = **cameraCalibration**()
K_0^{-1} = invert(K)
f_0 = video.read()
f_0' = undistort(f_0, d)
$rect_0$, $rect_1$ = **findRectanglePatterns**()
δ = **findRectanglePlane**($rect_0$, K_0^{-1})
γ = **findRectanglePlane**($rect_1$, K_0^{-1})
objectPoints = []
while *video.hasFrame()* **do**
  f_i = video.read()
  f_i' = undistort(f_i, d)
  rays = **findLaserRays**(f_i', K_0^{-1})
  upperRectRays = pointsInside(rays, $rect_0$)
  lowerRectRays = pointsInside(rays, $rect_1$)
  μ = **fitPlane**(upperRectRays, lowerRectRays)
  points3D = linePlaneIntersect(rays, μ)
  coloredPoints3D = **readColor**(rays, f_0')
  objectPoints.concat(coloredPoints3D)
end

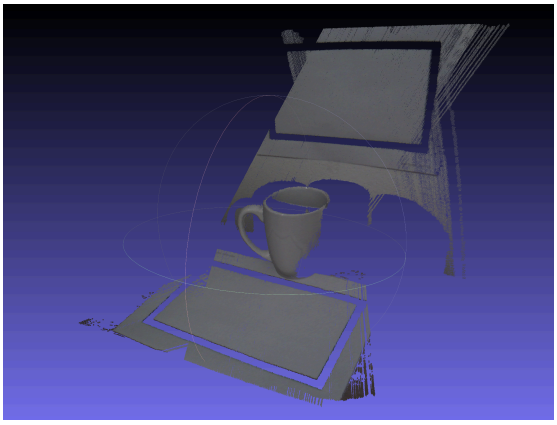

Figure 13: The scanned output in MeshLab, from the `cup1.mp4` dataset (3678321 points)

D. Performance

We made a time–based performance evaluation of our algorithm to see if it can run in real–time speed, as required by the project statement. We used the cProfile Python profiler to analyze the processing time of every part of the program, running the tests with a MacBook Pro *(i7-9750H @ 2.60GHz, 32Gb RAM DDR4, macOS 10.15.5)*.

We divided the time spent in initialization and mesh saving from the time spent in frame processing, to get a better estimate of the average frame processing time (table 14). Subtracting the frame time from the total time, we get an average of 19 seconds per video of fixed processing time.

As we can see from the profiling output (fig 15) most of the processing time is spent in:

- **Line–Plane Intersection** (42%): Difficult to do better in Python, since it involves a very long for loop, it may be useful to use less points
- **OpenCV Undistort** (16%): Unavoidable processing time, already highly optimized by OpenCV in C++
- **Ray Calculation** (9%)
- **Plane Fitting** (7%)

File name	Video length	Total Processing Time	Frames Proc Time
`cup1.mp4`	62s	101s	79s
`cup2.mp4`	45s	89s	66s
`puppet.mp4`	53s	91s	73s
`soap.mp4`	72s	112s	88s

Figure 14: Running times of the algorithm

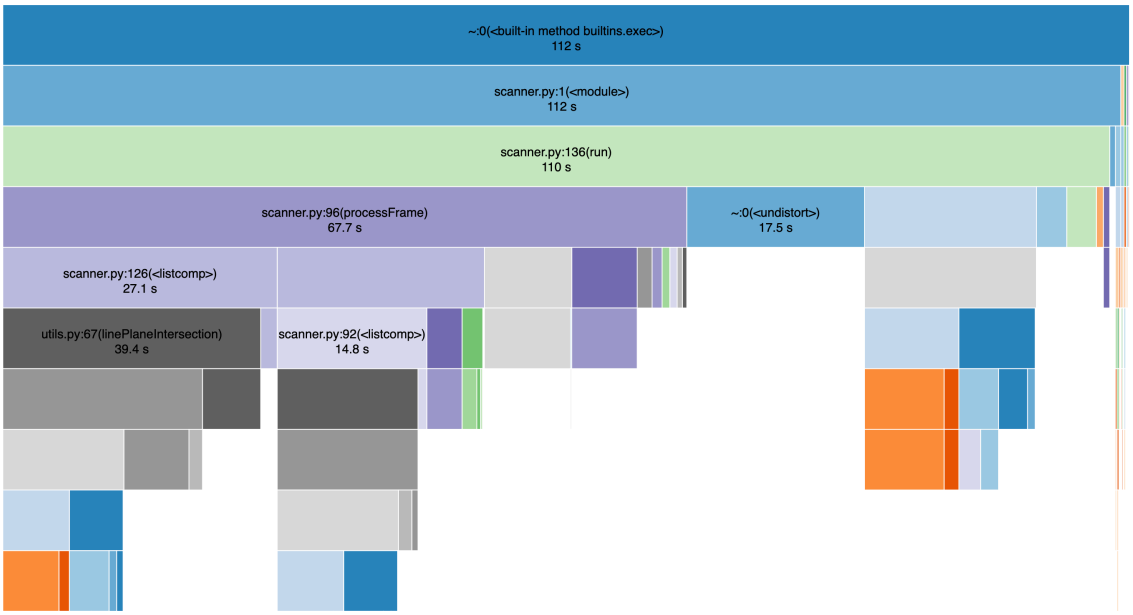

Figure 15: The time profile output from cProfile

Given that the videos are 15 FPS, and considering only the frame processing time, the average processing time per frame is 0.092 seconds, with an average speed of 0.73x.

IV. Conclusion

We have seen how it's possibile to build a monocular fixed–camera 3D scanner using structured light from an handheld laser line pointer, a cheap camera and some printer paper markers. We researched several ways to perform this efficently and precisely, both for finding the 3D points geometry and the laser pointers. We also proposed a Python implementation of our algorithm, testing its performance compared to the video speed. We have seen how it can yield decent results in some cases, while in other cases the fixed camera position makes impossible to scan some surfaces.

V. Future Improvements

As we have seen the main problem is the difficulty of finding the laser points in the image with such lighting conditions. We have explored many possibilities, like background removal or clustering algorithms, but just made the results worse and more noisy.

A solution to this could be building a new dataset with better lighting condition, such as lowering the ambient light, since there's little you can do if naked eye can't even recognize the laser line boundaries.

Another improvement that can be made to the dataset is make the object move or rotate during the scan, keeping the laser line still instead. This could help with bigger objects or concave surfaces, but may require a rethinking of the whole algorithm, since it expects a still camera.

References

[1] J. Bouguet and P. Perona. "3D photography on your desk". In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)* (1998), pp. 43–50.

[2] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. "Real-Time 3D Model Acquisition". In: *ACM Trans. Graph.* 21.3 (July 2002), pp. 438–446. ISSN: 0730-0301. DOI: 10.1145/566654.566600. URL: https://doi.org/10.1145/566654.566600.

[3] H. Nakai, D. Iwai, and K. Sato. "3D shape measurement using fixed camera and handheld laser scanner". In: *2008 SICE Annual Conference* (2008), pp. 1536–1539.

[4] Hiroyuki Aritaki et al. "3D Wand: A Handheld 3D Digitizing Device". In: (Dec. 2020).

[5] Andrew Harltey and Andrew Zisserman. *Multiple view geometry in computer vision (2. ed.)* 2003.

[6] Sebastian Bullinger. "Image-Based 3D Reconstruction of Dynamic Objects Using Instance-Aware Multibody Structure from Motion". PhD thesis. Karlsruher Institut für Technologie (KIT), 2020, pp. 17–19. 163 pp. ISBN: 978-3-7315-1012-3. DOI: 10.5445/KSP/1000105589.

[7] D. Eberly. "Least Squares Fitting of Data by Linear or Quadratic Structures". In: 2018.

[8] Z. Zhang. "A Flexible New Technique for Camera Calibration". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22 (2000), pp. 1330–1334.

[9] W Förstner and Eberhard Gülch. "A fast operator for detection and precise location of distinct points, corners and circular features". In: Jan. 1987, pp. 281–305.

[10] M. Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *KDD*. 1996.