Dense Reconstruction using Structured Light

Jente Vandersanden
Brent Zoomers

May 2022

1 Contents of the submission + overview of what we have implemented

We have submitted our implementation code to the provided GitHub repository in which this report is also located. Each of the functions we wrote contains a brief description in the code documentation and illustrates our approach. We also made a directory `/results` that contains all the screenshots for the intermediate results that we have obtained. We have made a full implementation of the base project requirements (week 1 - week 3), including 2 extra's (see last section). We have to mention that the triangulated points that we generate with our own matches are rather sparse (see below). However, when using the matches provided by the teaching staff, the results look plausible.

2 Intermediate results

2.1 Phase unrolling + false color visualization + matches

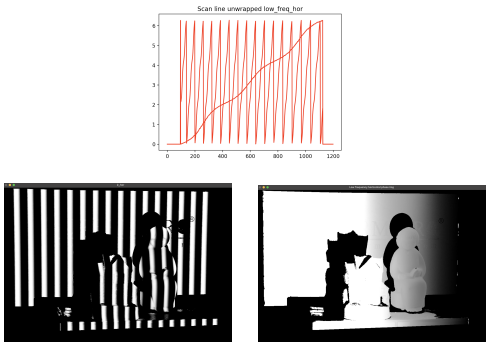

Figure 1: Graph representation of the frequency phase images. Both low and high are represented in this graph. Next, the horizontal high and low frequency phase images respectively. The phase runs from 0 to 2 pi (once in the low frequency image; 16 times in the high frequency image.)

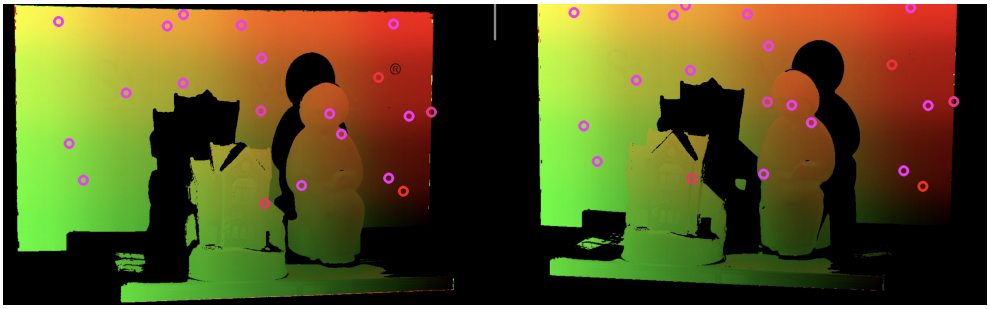

Figure 2: Pixel correspondences based on the false color visualizations of the left and right view respectively. Note that only 20 matches are marked on this example. When we want to find the camera poses through the correspondences between the 2 images, we use all matches. For pixels that somehow matched to multiple pixels in the other image, we took the median of the match coordinates to limit the effect of outliers.

2.2 Camera calibration

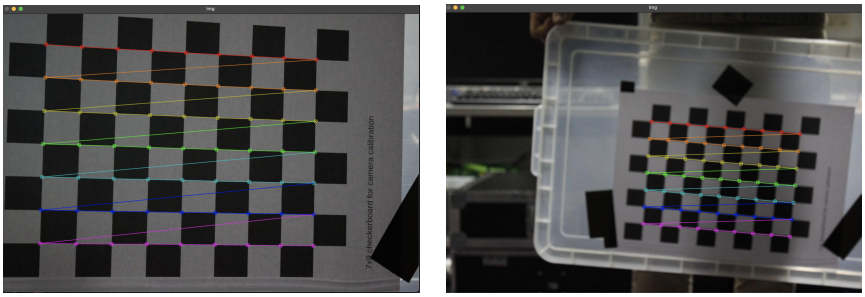

Figure 3: Chessboard pattern matching used for camera calibration. We used the *findChessBoardCorners* and *drawChessBoardCorners* functions to achieve this.

The camera was calibrated using the 2D image points of the chessboard corners and predefined 3D object points (*cv.calibrateCamera*). To undo the apparent distortion on the input images, we have tried out 2 approaches: *cv.undistort* and *cv.initUndistortRectifyMap, cv.remap*.

2.3 Finding the 3D point cloud

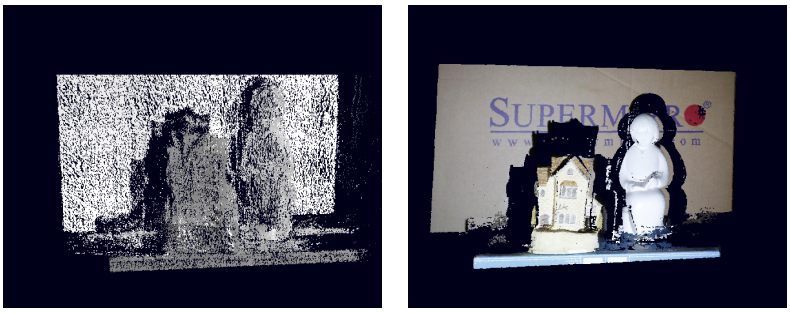

Figure 4: Point cloud visualization. On the left, the rather sparse (and noisy) point cloud generated using our own matches. On the right, the more dense point cloud generated using the teaching staff's matches.

```
Pointcloud center:  [-0.17917095  0.02746372  2.84181992]
```

Figure 5: Point cloud center calculated by taking the average coordinate in the 3 spatial dimensions. This will help us estimate the range in which we should generate depth planes during the plane sweeping algorithm.

2.4 Depth map

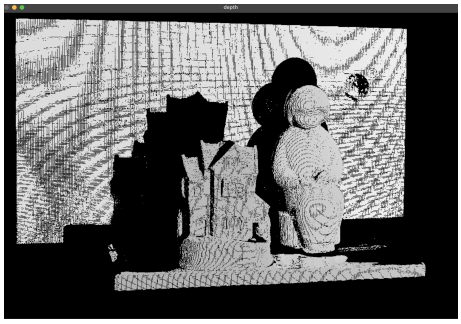

Figure 6: Depth map for the left camera view. The depth map is generated by calculating for each pixel the distance (Euclidean norm) between the camera's position and the 3D point that corresponds to that pixel.

2.5 Warp frame

We have tried to use the *rgbd::warpFrame* function in order to perform an image warp using the depth map information, but encountered some problems here. First of all, the function did not seem to work on opencv-python for Windows. Since one of us worked on MacOS we were able to test it there as well and the function did work on there. However, it delivered rather strange results, assuming that the parameters that we passed are correct, of course. Because these delivered us plausible results in other steps, we assume that this is the case.

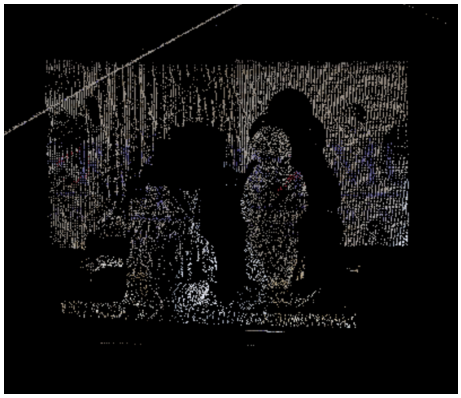

Figure 7: *cv::rgbd::warpFrame* warpedImage output for a hardcoded orientation. We can clearly recognize the scene in this output, however there seem to be a lot of 'gaps' between the pixels in the output. All these 'gap pixels' seem to have been warped to the diagonal line visible in the left top corner of the image.

2.6 Plane sweeping

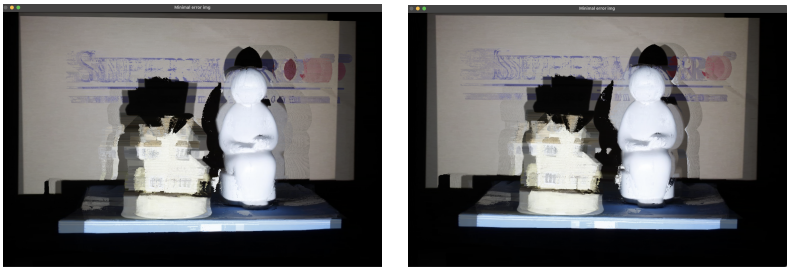

Figure 8: Reconstructed image for 2 intermediate cameras generated by the plane sweeping algorithm. The poses for the intermediate cameras were calculated using interpolation. We have experimented with the amount of depth planes and the range in which to place them. For the images above we used 30 depth planes in the depth range $[2.60, 2.95]$, which we think gives an acceptable result.

3 Extra's

3.1 Neighborhood error metric

We used a neighborhood error metric to hopefully improve the results of the plane sweeping algorithm. To implement this we used a 2D kernel that takes the neighborhood average of each pixel in the difference image. **In this way, we try to prevent outliers from impacting the final result.**

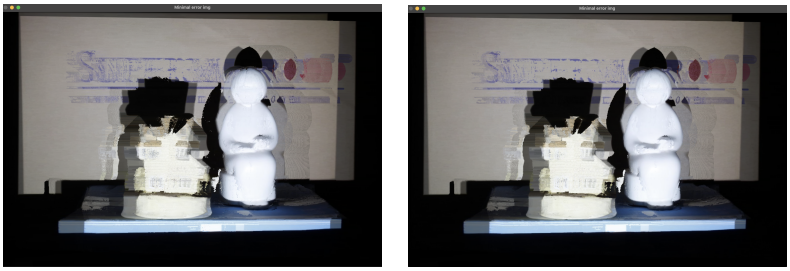

Figure 9: Left: plane sweeping output without using the neighborhood error metric. Right: plane sweeping output taking into account the neighboring pixels when calculating the error. The version that uses the neighbor metric has an overall smoother look.

3.2 Gray codes

The second extra we have implemented is structured light using gray codes. In our implementation, the difference in output between the two versions is that the version using gray codes produces better letters in the background, while the version using sine waves produces a more clear version of the statue. The house contains a lot of noise in both versions.
Another difference between the two implementations is that the point cloud is located slightly different. Using this implementation however, we find dense matches in comparison to the sine wave implementation. This allows us to use our own matches for the rest of the program in stead of the matches provided by the teaching staff.

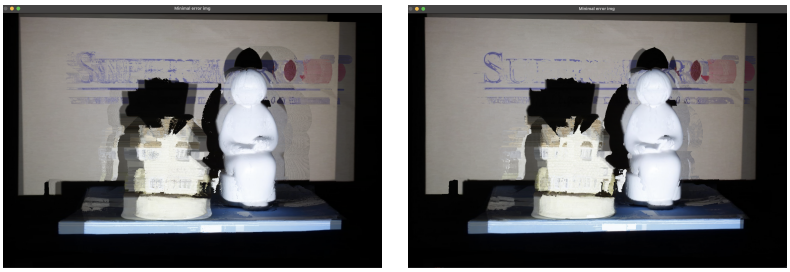

Figure 10: Left: plane sweeping output from sine wave implementation. Right: plane sweeping output from graycode implementation.

4 Encountered difficulties

The first difficulty we encountered was during unrolling the phase images. The result from the low frequency was not a straight line going from $[0, 2\pi]$ which resulted in the step intervals not being of equal size. This in turn resulted in the unrolled image containing artefacts. We eventually "solved" this by using a slightly different formula which results in a false color visualization very similar to the one shown in the assignment.

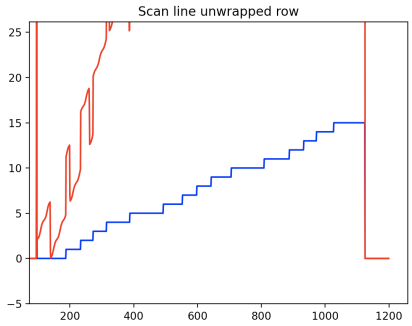

Figure 11: In red, the 'unrolled' phase. We can see that it makes upward and downward 'jumps' of size 2π while in fact it should be a monotonic increasing line. The used step function is plotted in blue. It is clear that the steps are not of equal size, which normally should be the case.

Using the resulting FCV, we determined our own matches which, in comparison to the matches provided by the teaching staff, look very sparse (figure 4). We can still clearly see the scene, but a lot of the details in both objects get lost.

A second difficulty occurred when we tried to use the *cv::rgbd::warpFrame* function. As mentioned in the Discord, this function did not work on Windows. As we have previously mentioned, we tried this on MacOS where we get some output in which we can clearly recognize the scene. The problem we have encountered using this function is that some points get mapped onto a line that is not part of the scene.