3D Model Reconstruction from Stereo 2D Images
Documentation and Report

Gabriele Galfrè, Arturo Cardone, Federico Sandrelli

December 2018

1 Introduction

1.1 Purpose

For this project, we implemented a system that generates a 3D model of an environment starting from a pair of Stereo Images. The software extracts depth information from the images and is able to do so in different light condition and scene configurations. The depth information is used to create a 3-dimensional scene and the colors of the original image are mapped onto the new space to recreate a faithful representation.
This project has been entirely built in Python 3.6 and, as will be better explained later on, makes use of the OpenCV - cv2 library.

1.2 Structure of this Paper

We firstly introduce one of the most important concepts of the project, which is the creation of the Disparity Matrix. We will then go over a series of steps we took to improve our Disparity Matrix and finally we will see how we used the disparity values we found to map the single pixels of an image to a 3-dimensional space.

2 Implementation

2.1 Overview

We display here a bird's eye view of the software pipeline of our project. Each step of the pipeline will be explored in depth.

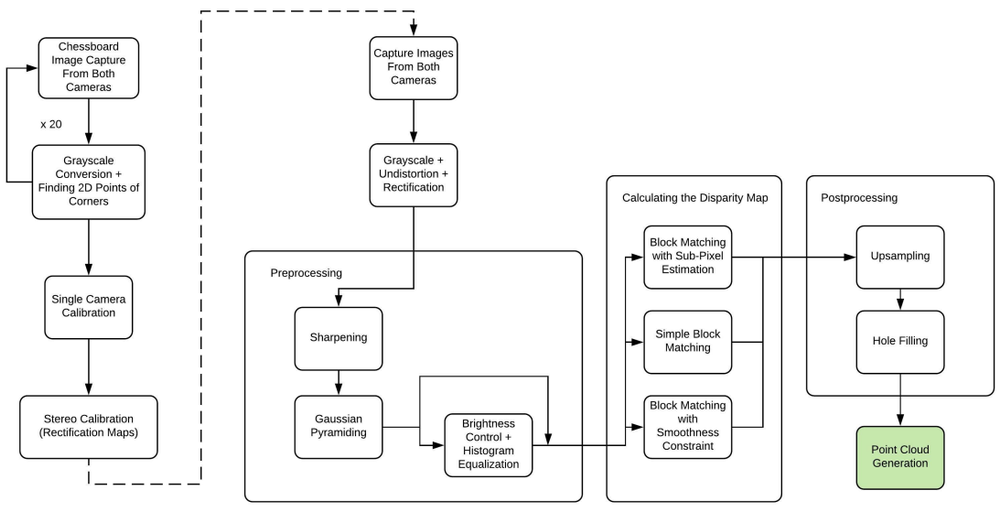


Figure 1: Pipeline

2.2 Calculating the Disparity Map

2.3 Simple Block Matching

To recreate a 3-dimensional representation of our images, we need to estimate the distance of every point in the scene (that corresponds to a pixel in the image) from our cameras. The first thing we need is a Disparity Map. To calculate this we initially implemented a simple block matching algorithm, using the Sum of Absolute Distances (SAD) metrics to match each pixel from the image captured from our right camera to a pixel in the image captured from our left camera. The idea behind the algorithm is to find how much each pixel has shifted horizontally from one image to the other and from this information triangulate the corresponding point in space. The amount of shift is inversely proportional to the pixel's distance from the cameras: objects closer to the cameras will shift more from one image to the other, while our infinity point won't move at all.

The search for the pixel in the second image is performed on the same axis (we will see why in the next section), and extends for k pixels, where k is a parameter we can tune depending on the environment we are shooting in. To identify the correct pixel in the search, the algorithm takes into account a small patch around the pixel (a $d*d$ matrix, where d is a parameter we set) and minimizes the SAD[1] between the matrix of the pixel to be found and all the candidate matrices extracted from the second image. Knowing which image is the left (right) one we can perform the search in only one direction.

[1]Sum of Absolute Differences

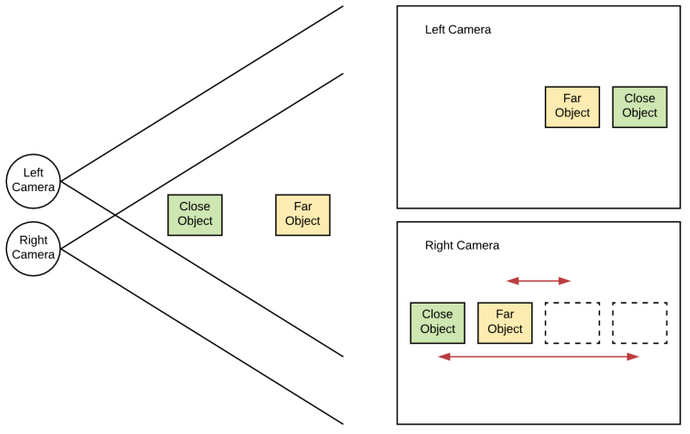


Figure 2: Object Shift due to Camera Perspective

Figure 3: Simple Block Matching[2].

The algorithm uses a very simple metric, nevertheless it performs pretty well if fed with correctly preprocessed images. The SAD metrics suits well to this application for two reasons:

- We need to evaluate each pixel within the image. There is a lot of computation involved and having a simple metrics reduces execution time.
- Given the limited search space of the search a simple metric will often perform just as well as a complex, slower metric.

2.3.1 Subpixel Estimation

The precision to which we can estimate the disparity value of a pixel (and consequently its depth in the scene) is limited by the pixel density of our image. To go beyond this limitation that restricts our disparity to scalar values, we can apply "Subpixel Estimation". Once we have identified our best match with the basic block matching algorithm, we take the corresponding minimum SAD and the SAD values of the neighbouring pixels and estimate a quadratic curve connecting the three. We then compute the minimum of this function by zeroing the derivative and this will be our new disparity value.
Applying Subpixel Estimation yields a slightly smoother depth transition which especially enhances flat surfaces. We can see this for example on the "belly" of the motorbike.

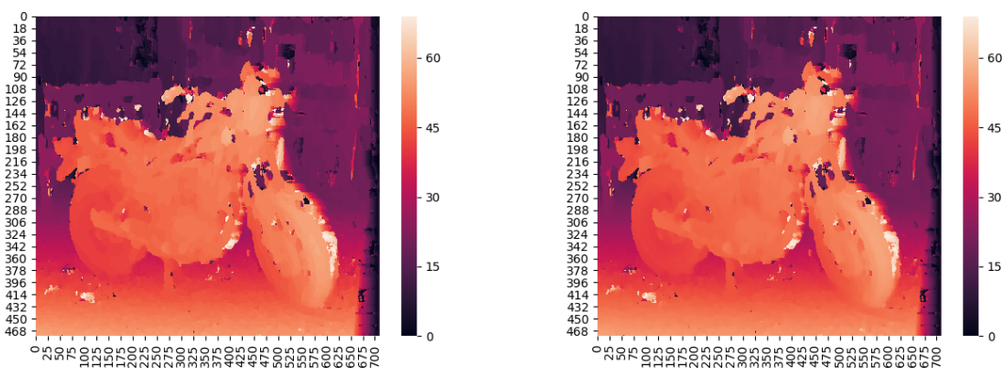


Figure 4: Simple Block Matching without (left) and with (right) smoothing

[2]Source: http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/

2.3.2 Horizontal Smoothing using Dynamic Programming

A second improvement we tried out has been to add a constraint to how much the disparity value of adjacent pixels can differ. Following the approach described in a Mathworks Article we found, we impose a global optimization for the choice of the pixel disparity value, optimizing over the whole line instead of always choosing a local minimum (like basic block matching does). This is achieved adding a cost to disagreement with neighbouring pixels. The overall result is improved and helps in removing outliers, but can also introduce horizontal linear artifacts. In the end we chose this as the algorithm to calculate the Disparity Map that is fed to the following stages of the program pipeline. The algorithm is optimized by using dynamic programming to reduce computational cost.

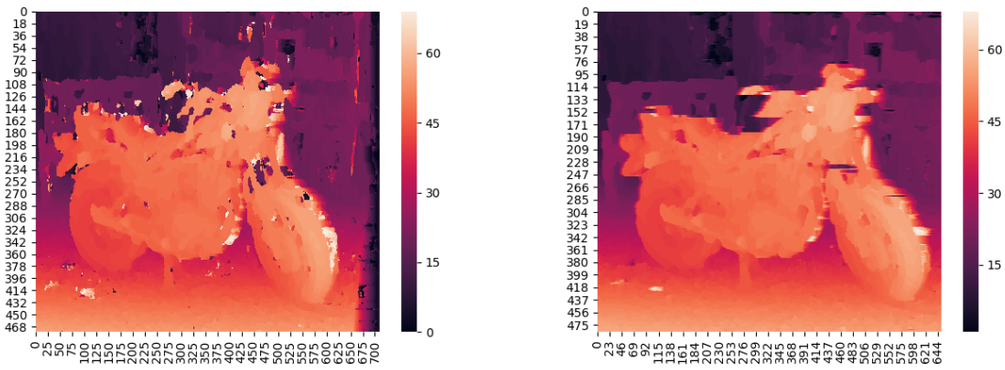


Figure 5: Simple Block Matching without (left) and with (right) sub-pixel estimation

2.3.3 Fast Disparity Map Calculation

One of the bottlenecks of our program is the execution time for the calculation of the disparity map. Python is know for being inefficient in this king of computations and a nested for loop over the image pixels resulted in having an execution time of over two minutes for the full sized images we were using. We propose a matrix based algorithm that parallelizes the computation of dispariy values using numpy arrays.

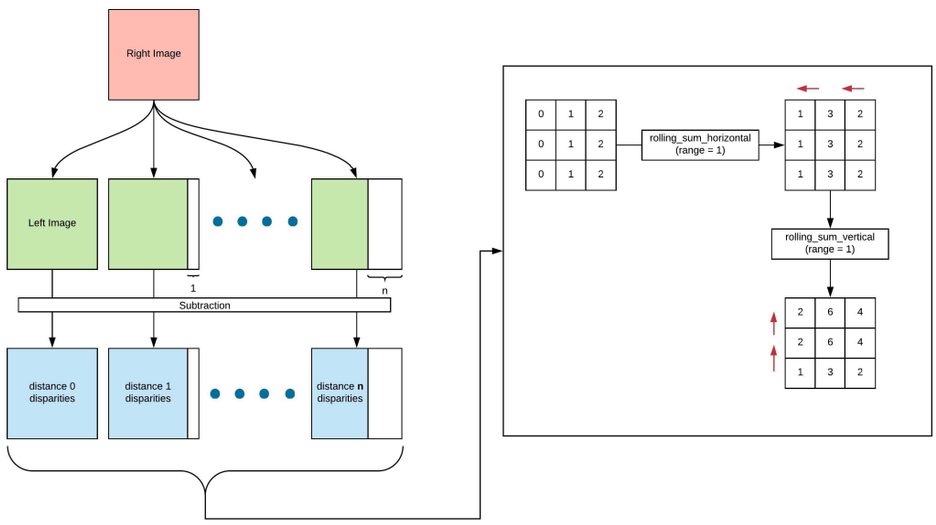


Figure 6: Matrix-Based implementation for Disparity Map calculation

This algorithm takes as input two gray-scale images and operates by creating a set of n matrices of the size of the original images, where n is the horizontal comparison range we are using (usually around 70). Each of these matrices contains the left image shifted by $i \in [0, n]$ pixels to the left (left side is cropped, right side is zero padded). To each of these, we subtract the right image and take the absolute value of the subtraction. At this point, to each pixel of the n matrices we sum the first n values to its right and then the first n values below it. In the image we can see how this operation works on a simple example with $n = 1$. We now have ended up with n matrices where the value of each pixel in matrix $i \in [0, n]$ is the SAD value for the pixel in the same position of the right image, calculated at distance $d = i$. By taking the minimum over the n

[2]Source: http://mccormickml.com/assets/StereoVision/Stereo%20Vision%20-%20Mathworks%20Example%20Article.pdf

matrices we get the Basic Block Matching disparity map. We report here a snipped of the code and the runtime comparisons.

```
    # every matrix in matrices stores the pixel distances with offset = i
    for i in range(0, cmp_range):
        matrices[i] = np.abs(gray_right - zero_left_roll(gray_left, i))

    for i in range(len(matrices)):
        matrices_hsum[i] = rolling_sum_horizontal(matrices[i], block_size)
        matrices_final[i] = rolling_sum_vertical(matrices_hsum[i], block_size)

    disparity_matrix = np.argmin(matrices_final, 0).astype(np.uint8)
```

```
def zero_left_roll(matrix, n):
    out = np.zeros(matrix.shape)
    rows, cols = matrix.shape
    out[:, 0:cols-n] = matrix[:, n:cols]
    return out


def rolling_sum_horizontal(a, n):
    ret = np.cumsum(a, axis=1, dtype=float)
    ret[:, n:] = ret[:, n:] - ret[:, :-n]
    ret = ret[:, n - 1:]
    return ret


def rolling_sum_vertical(a, n):
    ret = np.cumsum(a, axis=0, dtype=float)
    ret[n:, :] = ret[n:, :] - ret[:-n, :]
    return ret[n - 1:, :]
```

Figure 7: Matrix-Based implementation for Disparity Map calculation

Algorithm	Execution Time
Basic Block Matching	$309s$
Basic Block Matching with sub-pixel estimation	326
Block Matching with Horizontal Smoothing (Dynamic Programming)	$306s$
Matrix-Based implementation of Basic Block Matching	$1s$

Table 1: Execution times for calculating the disparity map of the 'Bike Picture' (480 * 720)

2.4 Pre-processing

Many preprocessing steps have been adopted to deal with the following issues:

- Effects of the lenses distortion on the images themselves;
- Efficiency of the matching algorithm when images have high resolution, the block size is not small or the width of the comparison range is relevant;
- Differences in light conditions related to the different spatial position of the web-cams;
- Too low details in the image, making the algorithm less effective.

2.4.1 Calibration and Image Undistortion

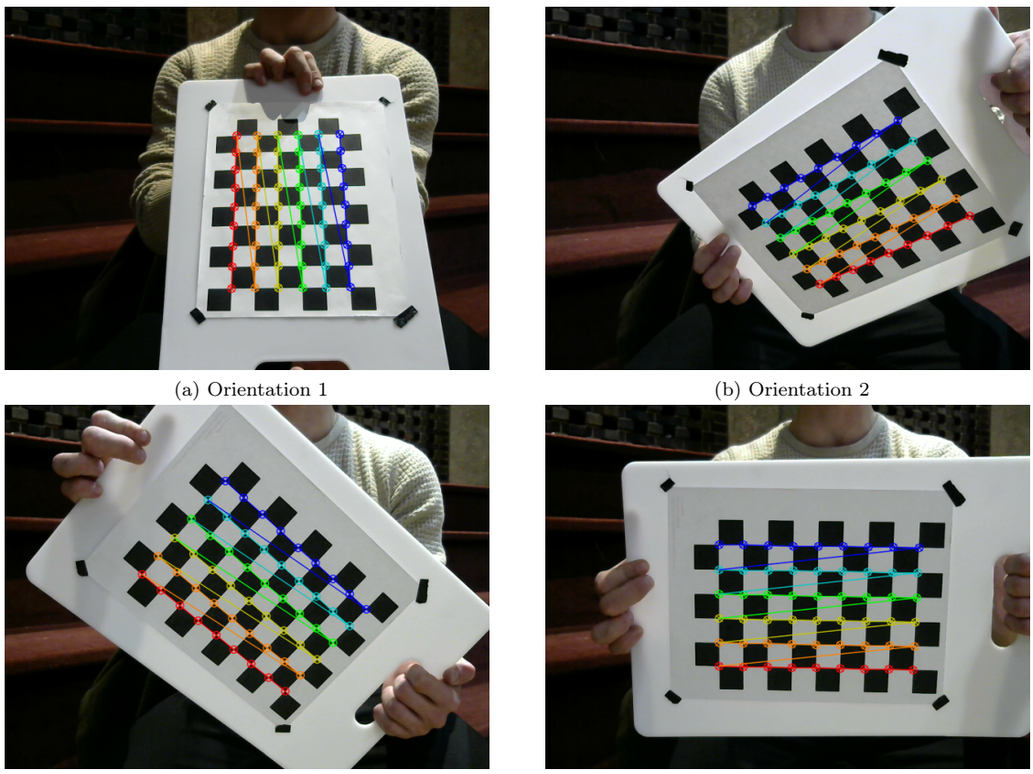
First of all, we need to calibrate the two cameras separately: this is because tangential and radial distortions modify the shapes of the objects inside our pictures. For example, due to radial distortion, straight lines might appear curved. To solve those problems, we need to find the Distortion coefficients and also the Camera Matrix (which contains focal lengths and optical centers).
In our system, in order to calibrate the cameras, we need a set of 3D points and their corresponding 2D points: 2D points are obtained as the image plane location of internal corners of a chessboard (more specifically where two black squares meet each other); 3D points instead cannot be known a priory with precision, so they are chosen in an arbitrary way supposing them to be on the same Z axis distance (0 in our case) with XY coordinates equal to (0,0), (1,0), (2,0) ... (6,9).
This unitary distance can be multiplied by the real-world size in mm of a chessboard square. In this way, we are supposing that the camera is moving around those fixed 3D points, which are mapped to the 2D coordinates on the image plane according to the current camera position and orientation (even if in reality it is the chessboard the one moving around, not the camera). In order to achieve a good calibration, we need to achieve:

• X Calibration: by moving the chessboard left and right, so that the chessboard can be detected at the horizontal edges of the field of view.

• Y Calibration: by moving the chessboard up and down, so that the chessboard can be detected at the vertical edges of the fov.

• Skew Calibration: by rotating the chessboard in various angles in respect to the camera.

• Size Calibration: by moving the chessboard closer and further from the camera.



(a) Orientation 1 (b) Orientation 2

(c) Orientation 3 (d) Orientation 4

Figure 8: Calibration process: identification of chessboard corners using different board orientations

Our code achieves all of this by taking 'n' pictures of a chessboard ('n' is a variable parameter that increases precision of the calibration process, as more sets of points are be obtained) with both cameras. Then, for each one of those pictures, we convert the image to gray-scale and we pass it to a CV2 library function which is able to execute feature detection for those chessboard corners, understand whether a chessboard is present with all of its angles, and return the 2D coordinates of the image plane where those corners have been detected. Eventually we will end up with 'n' sets of 2D-3D point pairs. Those are passed to another library function which computes a Camera Matrix, the 5 Distortion Coefficients and the Rotation and Translation vectors. Those parameters will be used to undistort images when they will be shot for depth map creation. From our experiments, we noticed that a balanced value for 'n' is 20, giving us an acceptable precision in the calibration process, without having to capture too many chessboard pictures.
The main OpenCV function exploited during this process are:

- 'cv2.findChessboardCorners': given the number of corners to be found, this returns the 2D coordinates on the image plane that were detected as chessboard corners.
- 'cv2.calibrateCamera': this function requires as arguments the set of 3D points, the corresponding detected 2D points and the image size, and returns the Camera Matrix, the array of Distortion Coefficients, and also the Rotation and Translation matrices:

$$CameraMatrix = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \quad DistortionCoefficients = \begin{bmatrix} k1 & k2 & p1 & p2 \end{bmatrix}$$

  Where 'fx' and 'fy' are the horizontal and vertical focal lengths, while 'cx' and 'cy' are the focal center coordinates. 'k1' and 'k2' are the radial distortion coefficients, while 'p1' and 'p2' are the tangential distortion coefficients.

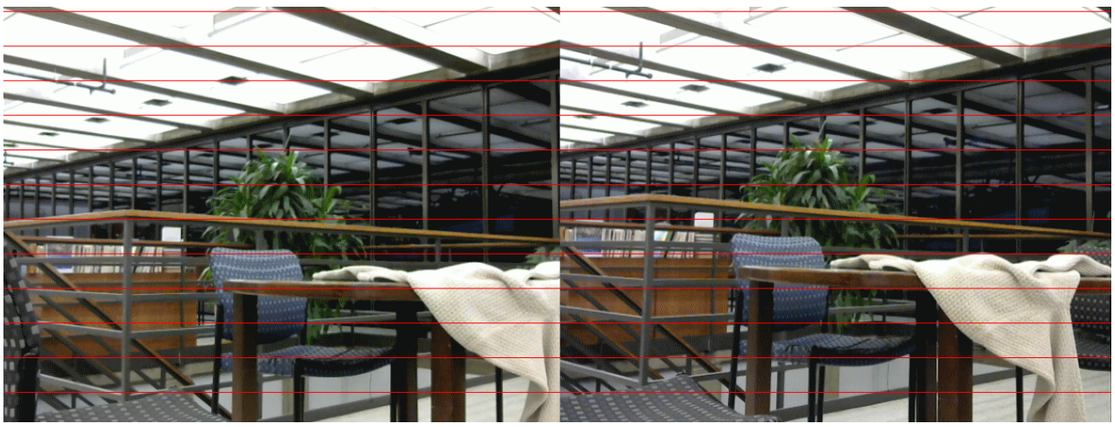
2.4.2 Image Stereo Rectification

Now we have all parameters that we necessitate to calibrate the cameras singularly, but we still need to calibrate the cameras together. By gathering all previously calculated parameters (3D chessboard points, 2D chessboard points for both cameras, Camera Matrix for both cameras, Distortion Coefficients for both cameras), we are able to obtain the Rotation Matrix R and Translation Vector T between the 1st and 2nd camera coordinate systems:

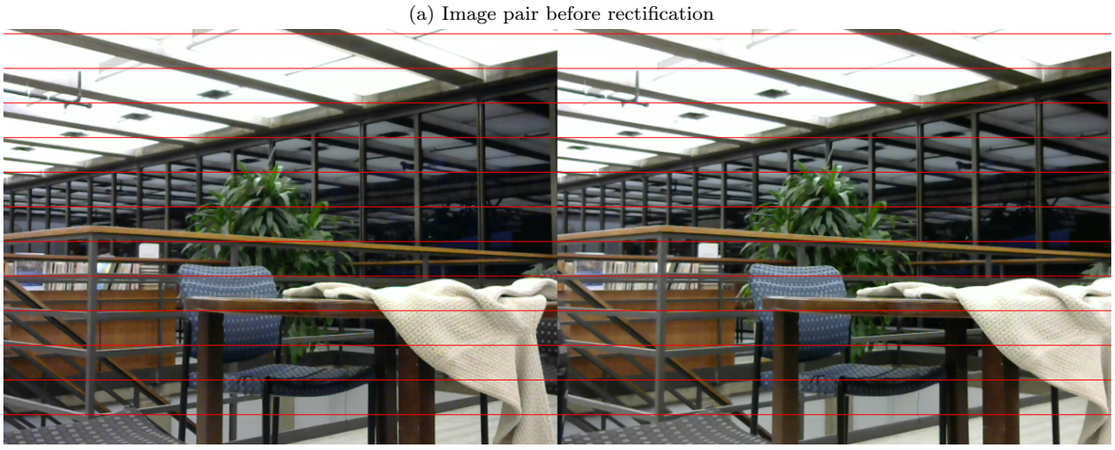
$$R_2 = R \cdot R_1$$

$$T_2 = R \cdot T_1 + T$$

Where R_i and T_i are the computed poses of the chessboard in respect to camera i, while R and T make it possible to calculate the position of a camera relatively to the other.
Then, the Rectification transformation maps are computed using those parameters. Those, together with the Rotation Matrices, are used in order to make both camera image planes the same plane.
This is an essential step, as for the Stereo Matching algorithms that are later used, we need all corresponding objects seen by the two cameras to be on the same horizontal line of pixels. In fact, considering that now



(a) Image pair before rectification



(b) Image pair after rectification

Figure 9: Rectification process: before/after effects of making the image planes and epipolar lines parallel

the epipolar plane intersects an image plane that is unique for both the cameras, Rectification allows for the epipolar lines to be parallel, simplifying the stereo correspondence problem, as our algorithm will just need to search horizontally for the group of pixels that correspond to the same object in the two images, resulting in a disparity value.
The main OpenCV functions that are used in this process are:

- 'cv2.stereoCalibrate': This function requires the Camera Matrix and Distortion Coefficients for both cameras, and outputs the rotation and translation vectors R and T, which are necessary to relate the coordinate systems of the two cameras.
- 'cv2.stereoRectify': using R and T, it allows to calculate the Rotation matrices for both cameras. Once those are applied, the two image planes become parallel.

The effects of rectification can be noticed in Figure 5: the image planes are rotated so that the details can be seen along the same epipolar lines on both left and right images. This can be seen very well if the leaves of the tree are observed in both pairs.

2.4.3 Filtering and Intensity Manipulation

The filtering, as well as all the other preprocessing operations we have applied to the input stereo images have the common purpose of increasing the chance of the matching algorithm to better differentiate between non matching neighborhood, as well as easily recognize matching ones.

The first filter we applied is the sharpening *Laplacian Filter*, which produces images with higher quantity of details, highlighting the regions with rapid intensity changes.

$$LaplacianKernel = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

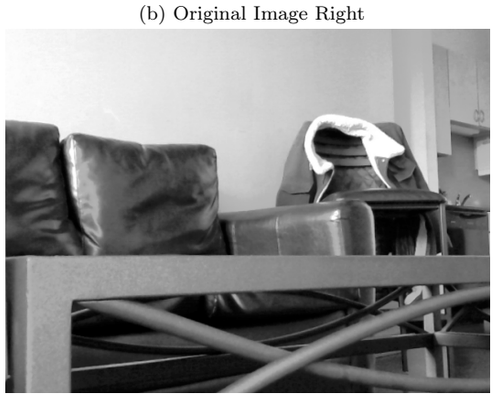
The result is an image that presents an higher amount of information in each neighborhood, allowing a better differentiation between neighborhood that don't have to match.

In addition, in order to enhance the precision of the matching algorithm and allow it to better define the similarity between neighborhoods of pixels, we decided to apply *Histogram Equalization*.
This technique is usually adopted to increase the contrast of the image and also to better exploit the intensity values available by stretching the distribution of pixels to cover them more uniformly.
The main issue we encountered with the execution of this procedure was the presence of colorful objects in one of the two stereo image that are missing in the other one.
The equalization works similarly on the two images only if they share the same intensities' distribution type.

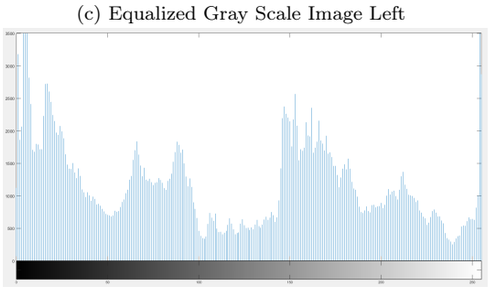
(a) Original Image Left

(b) Original Image Right

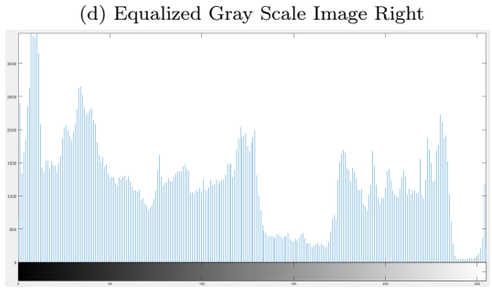
(c) Equalized Gray Scale Image Left



(d) Equalized Gray Scale Image Right



(e) Histogram of Left Image



(f) Histogram of Right Image

Figure 10: Histogram Equalization with colorful object in one picture only

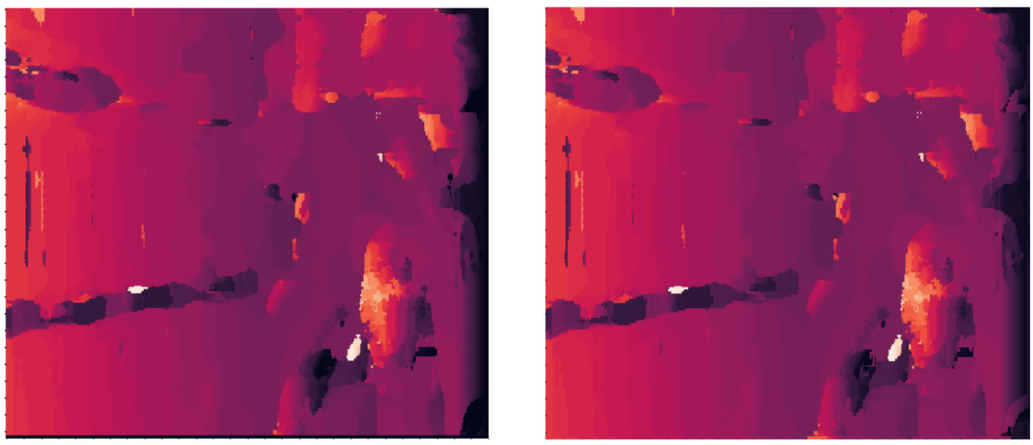
The presence of very bright or dark objects in one of them can seriously change the effect of this approach on one image with respect to the other, obviously deteriorating the matching algorithm results.
In Figures 10 we can observe from the depicted histograms the different distribution of intensities, causes by the presence of the yellow bottle in one of the images only. From the gray-scale images resulting from the histogram equalization, we can notice that the intensities of objects that before were nearly the same, now are slightly alterated.

The solution we adopted simply apply the equalization only if it has been proved that no colorful body is depicted in only one of the pictures, by analyzing the histogram of the stereo images.
The width of the central range of intensities covered by the 95% of the pixel of each image is measured and if they are equal it means that there are no crucial differences in the scenes depicted.

Another issue that could be cause of mismatching is the different light conditions perceived by the two web-cams.
Since they are physically positioned in different space positions they could sense differently brightness values, because some obstacle shadow a light source or some reflecting object is directing more light towards one of the two .
This problem can be easily solved by adding and subtracting an intensity offset respectively to the two images, trying to align their overall average brightness.
This step has been executed right before the application of the equalization, analyzing beforehand the histograms of the images and computing the mean value between intensities, weighting them by the number of associated pixels as follows:

$$M = \sum_{i=0}^{255} Histogram(i) \cdot i$$

Where i are the possible intensity values. Once both the average intensities have been computed the mean value between them is computed and respectively, a positive and negative offset are applied to the images in order to get their average brightness on the same value.
Also in this case the average brightness is strongly influenced by the type of intensities distribution: a colorful object present in only one of the pictures could unbalance the values computed.



(a) Disparity result without Hole-filling

(b) Disparity result with Hole-filling

Figure 11: Hole-filling technique: the effects can be noticed in the lower and right sections of the image

Therefore, as well as the equalization, this pre-processing step is executed only after the checks explained above.

2.4.4 Pyramiding: Downsample and Upsample

One main efficiency issue of the matching algorithm, even with the horizontal search only, was caused by the quantity of pixels that it has to scan while looking for matches.
To reduce the effort required by the algorithm we opted for exploiting the *Gaussian Pyramiding*.
In particular we donwsampled all of the input images before being fed into the algorithm, decreasing by a factor of four the total number of pixels.
Also the size of the range of pixel to check horizontally, as well as the image resolution, has been decrease, reducing by half the number of neighborhood comparison required.

This technique is reducing the precision of the results because the downsampling involve a smoothing of the image through a *Gaussian Filter*. On the other hand we kept the neighborhood size as the original one, so the matches are executed considering four times the amount of neighbor pixels.
An additional good reason to adopt the downsampling is the noise removal, smoothing away all of the lighter noisy pixels.

Once the Simple Block Matching finishes its work, the resulting disparity map is upsampled to regain the original resolution. In order to have disparity values compliant with the original range of comparison, the whole map's values are multiplied by two for all the downsampling steps required.

2.5 Post-processing

2.5.1 Hole Filling

Often, few of the pixels are assigned disparity values of zero in an erroneous way: this might happen because the represented object is too uniform, or simply because the rectification process was not executed in an exhaustive way. Generally, this happens for a very small number of pixels that are surrounded by correctly measured pixels.
In order to mitigate the effects of those wrong values, an 'Hole-filling procedure was applied': for each pixel that is zero-valued, we check its NxN neighborhood (N=5 seemed to be a great value for this task). If all neighbors are zero-valued too, then that means that probably this is actually an area that is supposed to have this value, and as such, we leave this pixel with its original zero value. Otherwise, we select the neighbor with the highest value, and we use it as the central pixel's new disparity value. Average value among non-zero pixels had been tried as well, but seemed to perform worse than simply using the max value.

2.5.2 Outliers removing

Almost always the matching algorithm is not able to find the correct match for particular pixels.
Most of the times it is caused by the presence of monochromatic objects and surfaces or bright light sources that introduce the presence of different neighborhoods very similar to each other.
Another cause of mismatching is the presence of object too close to the camera, requiring an horizontal range size higher than usual to find the matching neighborhood.

From what we experienced in many situations and scenes, the mismatching cause the presence of pixels or small regions of pixels with very high or very low disparity values, that are not correctly representing the supposed match.
These outliers can be simply removed by limiting the range of disparity values accepted by the algorithm to be translated in 3D points later.
Our program handle this outliers removal through a percentage parameter that defines the portion of highest and lowest values of disparity that must be removed before computing the spatial coordinates.

This obviously limits the minimum and maximum depth detectable by our system, but it considerably improve the results.

2.6 Depth-Map and 3D Coordinates

The *Depth Map* is the last step towards the generation of the final 3D model, and it simply represents an association of a real depth measurement (meters) to each pixel's disparity value.
Once we have obtained these values, the 3D model is simply created by projecting in the space the perspective line associated to each pixel and by fixing a point at the depth defined in the depth map.
Here follows an explanation of the attempts and the results obtained.

2.6.1 Reprojection

First attempt consisted in executing the complete *Reprojection* Geometric Transform of the points in 2D pixel coordinates, obtaining the 3D ones.
In order to do this transformation we needed the correct disparity-to-depth reprojection matrix, which consist in a 4×4 square matrix with the following format:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & \frac{1}{T_x} & \frac{c_x - c'_x}{T_x} \end{bmatrix}$$

where:

- c_x is the horizontal pixel coordinate of the principal point of one of the cameras. It is the orthogonal projection of the focal point onto the image plane.
- c_y is the vertical pixel coordinate of the principal point of the same camera.
- c'_x is the horizontal pixel coordinate of the principal point of the other camera.
- T_x is the baseline length, which corresponds to the distance between the two cameras.
- f is the distance between the focal point of the camera and its image plane.

The reprojection is executed as follows:

$$3dHomogeneous = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = Q \cdot \begin{bmatrix} x \\ y \\ dispartity(x,y) \\ 1 \end{bmatrix}$$

$$3dPoint = \begin{bmatrix} \frac{X}{W} \\ \frac{Y}{W} \\ \frac{Z}{W} \end{bmatrix}$$

where:

- x and y are the coordinates of the pixel in the image.
- $disparity(x, y)$ is the disparity value associated to the pixel in position x, y.
- X, Y, Z and W are the 3D real world spatial coordinates, measured in meters in our case.

Our initial approach with reprojection has been adopting the Q matrix returned by the pre-processing rectification step, in particular by the *OpenCV* function *StereoRectify*.
This function is supposed to compute the different values during the rectification process,based on the information obtained analyzing the pictures of the same chessboard views from both cameras.
We obtained poor results, with a perspective angle extremely wide and distance measurement far from the reality.

Next approach was manually writing the reprojection matrix, defining the values by ourselves.
We assumed that the principal point of the cameras, once rectified, was precisely in the middle of the image.
We also tried different f values, retrieving from the web-cam manufacturer catalogue the average focal distance of our camera model, and searching on the internet for common measures.
All of these attempt ended in similar results as the previous one, so we decided to abandon this approach.

2.6.2 Depth by Measurements

The idea we successfully decided to follow simply aimed at generating a direct association between disparity values and depth measures.
It consisted in generating the disparity map of some scenes, measuring the effective distance of the object depicted from the camera and associating this measures to the corresponding disparity values.
Once we obtained enough pair of values, we simply interpolated them with different type of curves, trying to obtain a general function for disparity-depth association.

The final results have been generated adopting a cubic interpolating curve of the following type:

$$y = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$$

where x is the disparity value and y is the depth measures in meters.
The curve parameters have been generated by the *polyfit* function of the *NumPy* library, and using a minimum of three points.
Several attempts have been tried to obtain associations as close as possible to the original values, and most of the times the resulting curve reached negative values, which represent unfeasible results.
Adding points that resulted out of the interesting disparity range to the set to be interpolated, we managed to lead the curve almost to the desired trend.
Since the disparity values are limited by the choice of the horizontal search range parameter, the interpolating curve is valid only for a predefined set of disparity values.
We tuned our project on a value of 70 pixels for the search range because satisfying in the type of scenes we chose to capture, causing the interpolating point to fit within a range of disparities between 0 and 70.
An example of curve is shown in Figure 12.

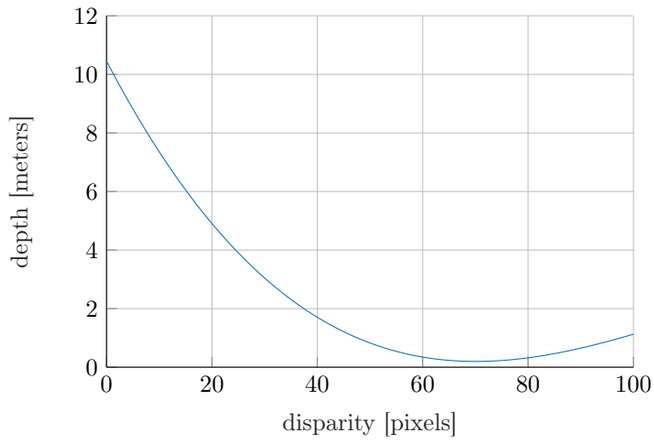


Figure 12: Cubic interpolation curve for Disparity-Depth association

2.6.3 Coordinates by Measurements

Once we have the real depth values, we also need to compute for each pixel the other two coordinates somehow manually.
In order to achieve reliable measurements, we followed another approach.
The idea is based on the fact that we want to avoid to assume the coordinates of the position of the focal point and the image plane of the camera, because difficult to measure, if not impossible, and source of tuning and trial and error attempts.
To be able to project in the space a line for each pixel, without knowing these two objects information, we decided to directly measure the dimensions of the full rectangular scene that the camera is able to capture at a predefined distance.

In practice we captured the image of a white wall at the distances of 0.75 meters and 1.5 meters and used the pictures to draw on the wall itself the rectangles that were fitting in the field of vision of the camera. Then we simply measured height and width of the rectangles.
These measures, with the distances from the wall, have been used to associate to each pixel of both rectangles three spatial coordinates, forcing the central pixel to have x and y coordinates equal to 0. Here follows how the coordinates have been computed:

$$centralX = \frac{WidthMeters}{2} \qquad centralY = \frac{HeightMeters}{2}$$

$$\begin{bmatrix} x_{i,j} \\ y_{i,j} \\ z_{i,j} \end{bmatrix} = \begin{bmatrix} \frac{WidthMeters}{WidthPixels} \cdot j - centralX \\ \frac{HeightMeters}{HeightPixels} \cdot i - centralY \\ DepthMeters \end{bmatrix} \quad \forall i, j \text{ such that } i \in [0, HeightPixels), \quad j \in [0, WidthtPixels)$$

Where:

- $WidthMeters$ and $HeighthMeters$ are the rectangle's dimensions measured in meters.
- $WidthPixels$ and $HeighPixels$ are the image's dimensions in pixels, simply called image's resolution.
- $DepthMeters$ is the measured distance of the rectangle from the camera.
- i and j are the indexes representing the pixel position in the image.

In Figure 20 is shown a graphical representation of this technique After this step we have all the information

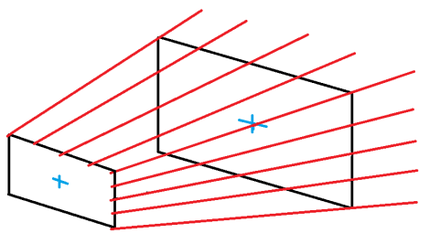


Figure 13: Projected lines

necessary to compute the actual coordinates of the points of our 3D model.
They are computed firstly associating the depth obtained from the previous step as the z coordinate.
Then, we consider the lines passing for the pair of points belonging to the two rectangles and associated to the same pixel and for each pixel we fix the z value from the depth map, obtaining the remaining coordinates x and y.
Here follows how the coordinates are computed:

$$k = \frac{DepthMap(i,j) - RectangleB_z(i,j)}{RectangleA_z(i,j) - RectangleB_z(i,j)}$$

$$\begin{bmatrix} x_{i,j} \\ y_{i,j} \\ z_{i,j} \end{bmatrix} = \begin{bmatrix} RectangleB_x(i,j) + k \cdot (RectangleA_x(i,j) - RectangleB_x(i,j)) \\ RectangleB_y(i,j) + k \cdot (RectangleA_y(i,j) - RectangleB_y(i,j)) \\ DepthMap(i,j) \end{bmatrix}$$

Where:

- i and j are the indexes representing the pixel position in the image-
- $DepthMap(i,j)$ is the real depth in meters computed previously for pixel (i,j).
- $RectangleA_x(i,j)$, $RectangleA_y(i,j)$ and $RectangleA_z(i,j)$ are respectively the coordinates for pixel (i,j) belonging to one of the rectangles introduced above.
- $RectangleB_x(i,j)$, $RectangleB_y(i,j)$ and $RectangleB_z(i,j)$ same as above but for the other rectangle

Each pixel has been translated into a real 3D space coordinate in meters, represented with the same color of the pixel itself.

2.7 3D Model

In order to ease the storing and use of the 3D models, they are saved as .ply files. The name stands for 'Polygon File Format', and their structure consists of 3 main parts:

• Properties section: here we have to declare the data type (float, int32, ...) that describes each of the properties of a point (such as color and position). Also, the number of vertices and faces of the model are to be indicated.

• Vertex section: here we list every single point that has to be represented in the 3D space together with its property values. In our case, we had to indicate X,Y,Z positions, and the RGB color.

• Faces section: same as before, but this time for 3D faces. Not used in our case, as we just represent Point Clouds.

To represent our point cloud, we simply use X, Y and Z positions derived from the Depth Map determination process, and thanks to the 'plyfile' library we convert them to a data type that can be inserted correctly inside a .ply file.
Also, we are able to color those points exploiting the fact that our disparity map is calculated as an offset from one of the two images (right image is the dominant one in our case), and as such we simply use the RGB values of this image, applying them to the depth values calculated for each pixel.

3 Final Results

Here we report some of the models we generated starting both from rectified images ('Bike Picture' and 'Cones Picture') we found on the internet[3] and from images taken from out stereo system (these images are affected by our process of calibration and rectification).

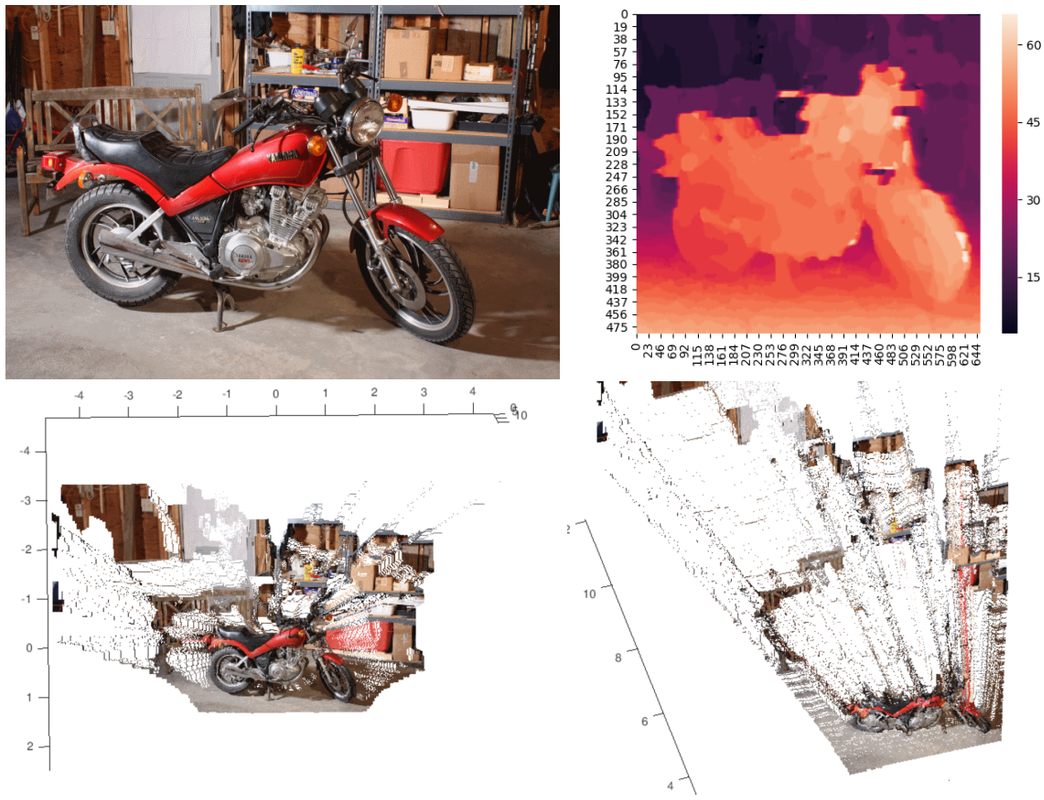


Figure 14: Bike: Rectified Image, Heatmap, PointCloud

[3] http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/

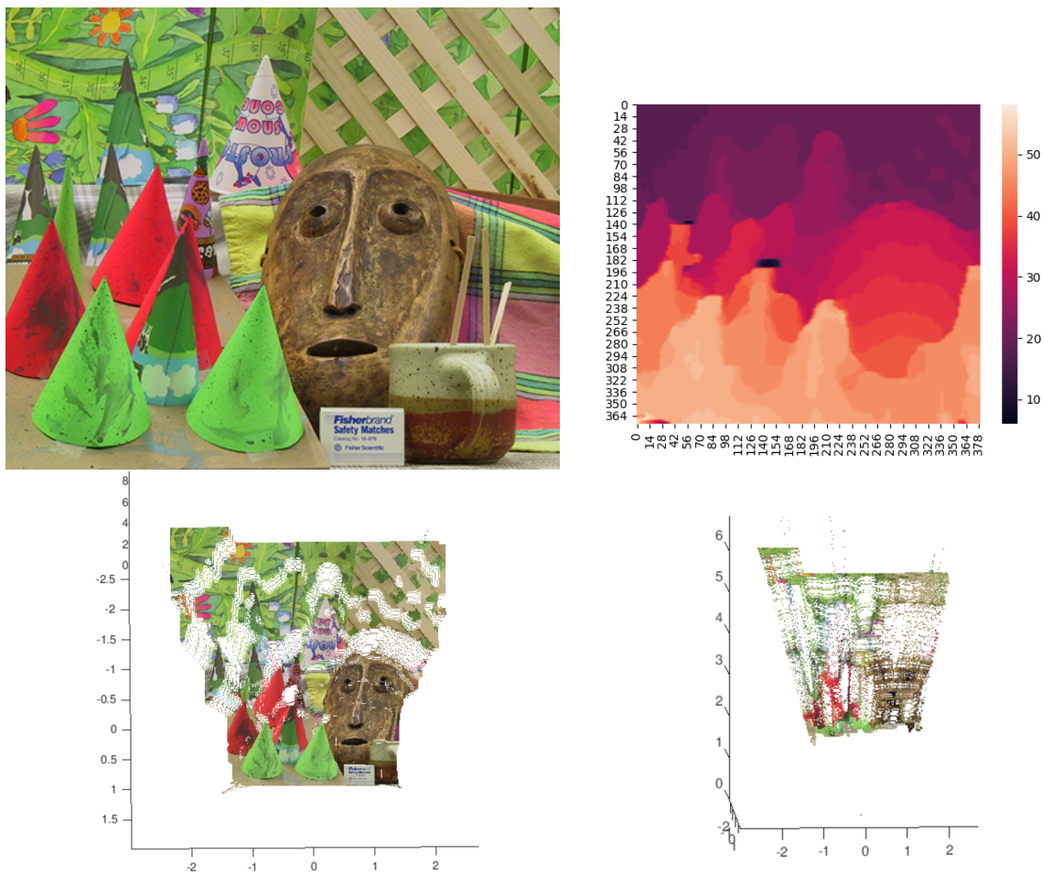


Figure 15: Cones: Rectified Image, Heatmap, PointCloud

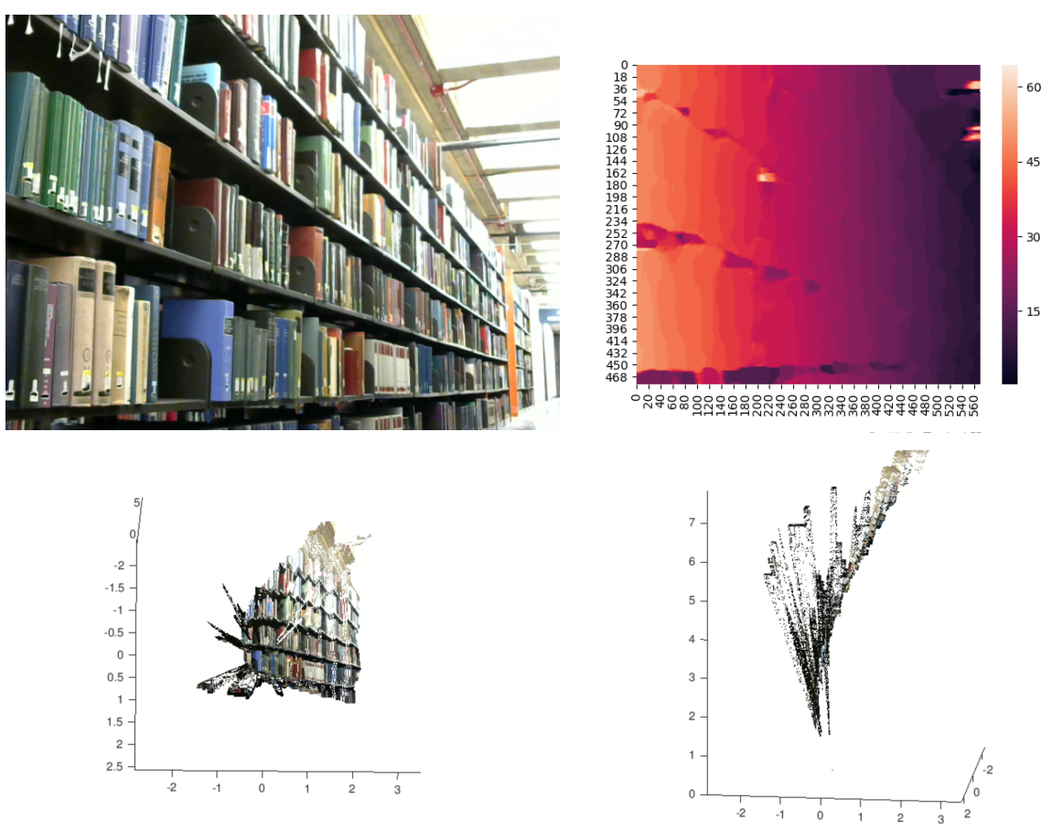


Figure 16: Our image: Rectified Image, Heatmap, PointCloud

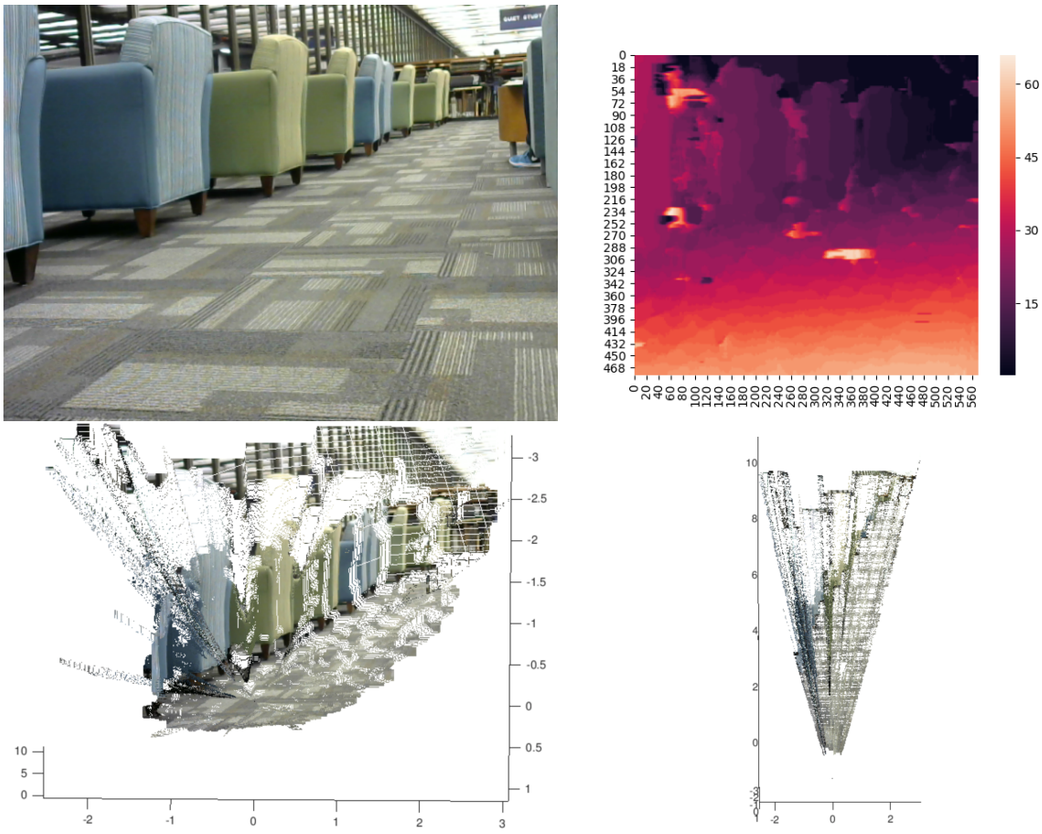


Figure 17: Our image: Rectified Image, Heatmap, PointCloud

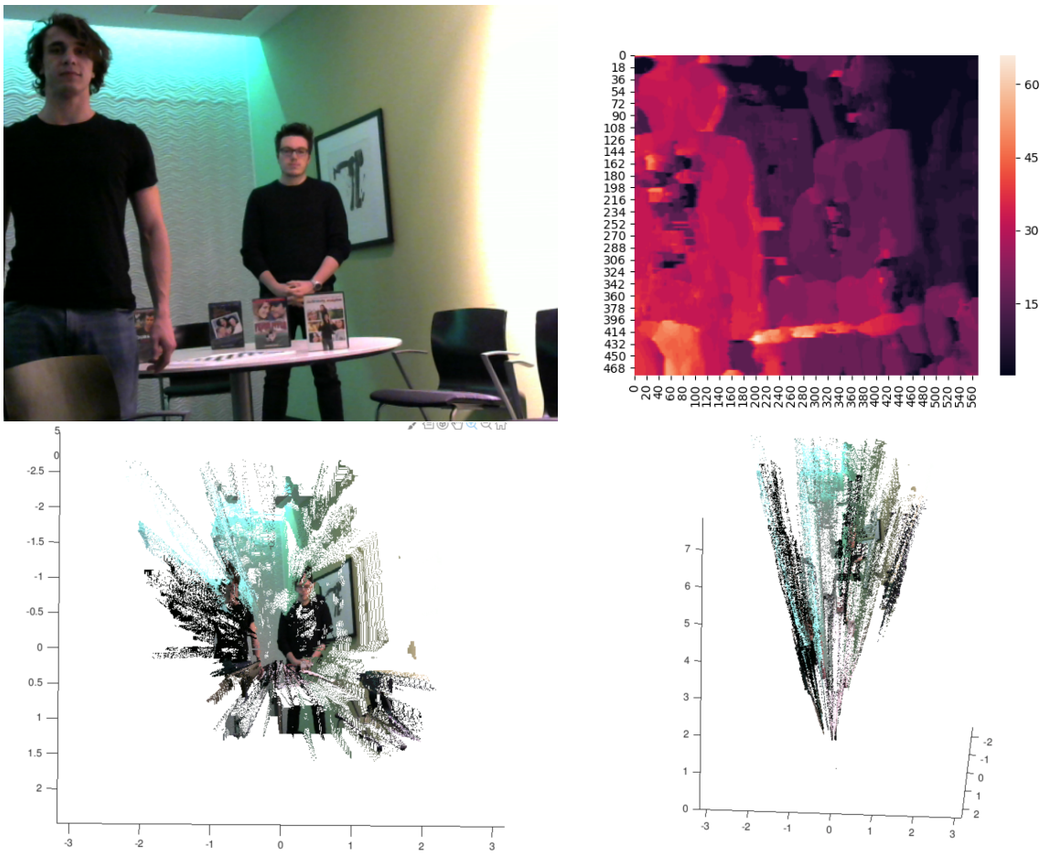


Figure 18: Our image: Rectified Image, Heatmap, PointCloud

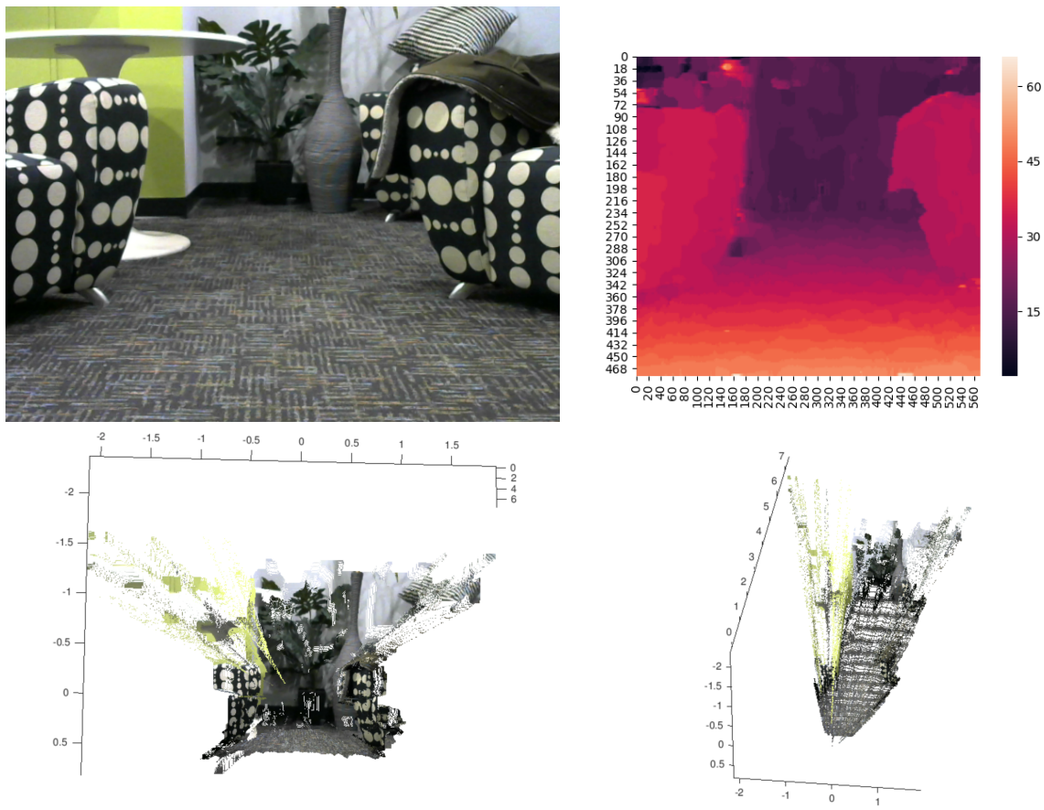


Figure 19: Our image: Rectified Image, Heatmap, PointCloud

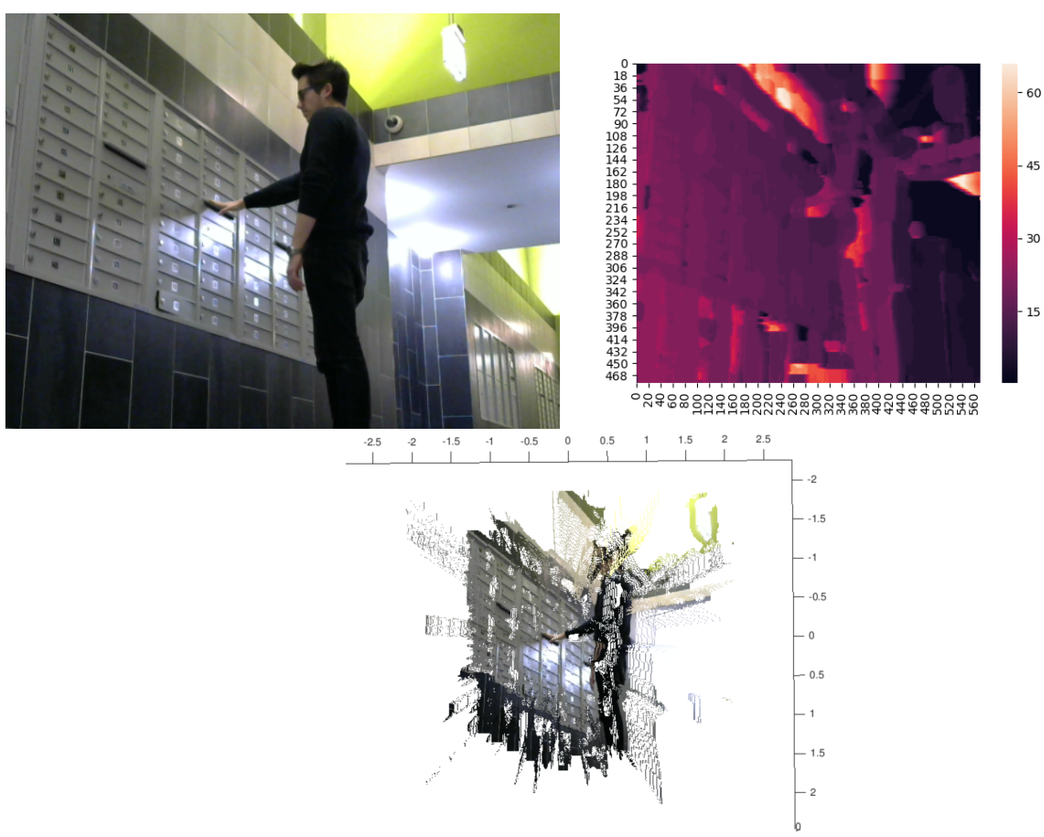
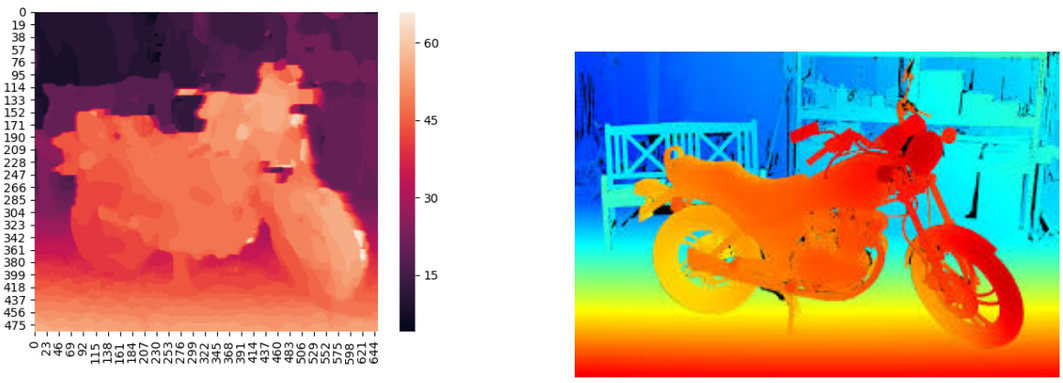


Figure 20: Our image: Rectified Image, Heatmap, PointCloud

4 Testing

4.1 Testing disparity calculation

Given that some of the images that were used in order to test the disparity map calculations were retrieved from the Middlebury Stereo Dataset [4], it seemed a good idea to compare the performace of our algorithm against the disparity results that can be found on the dataset for every image.

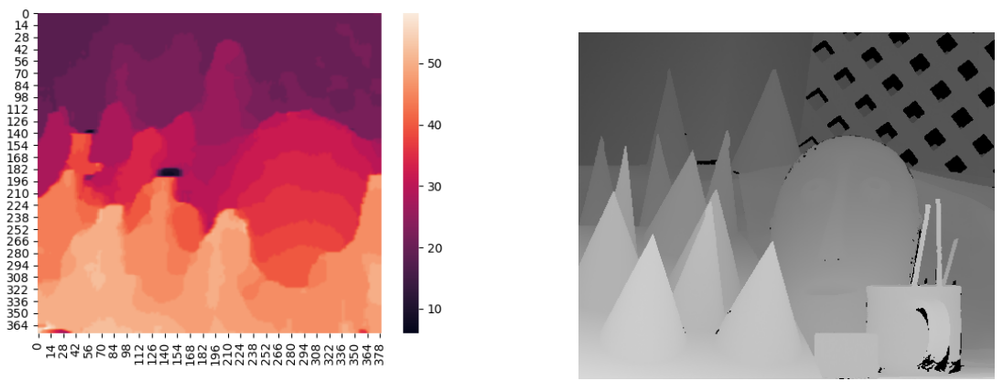


(a) Our disparity results

(b) Disparity ground truth

Figure 21: Comparing the disparity map results using the 'Motorcycle' image from the Middlebury dataset

If we look at the case of the 'Motorcycle' example, it is possible to notice very easily that the benchmark is a bit more defined in the subject contours than our result: this is because of the pyramiding process on one side, which reduced the image's resolution in order to improve performance, but also because of the sub-pixel estimation process on the other (used to help in the outliers removal).
Besides that, the results seem to be very satisfying, being able to capture with very high precision the floor, the bike, and the background: the floor, even if pretty uniform, is correctly identified in its gradual departure from the camera position. The same goes for the motorcycle, where the disparity is obtained with such a precision that it is possible to notice the small various mechanical elements of the bike itself.



(a) Our disparity results

(b) Disparity ground truth

Figure 22: Comparing the disparity map results using the 'Cones' image from the Middlebury dataset

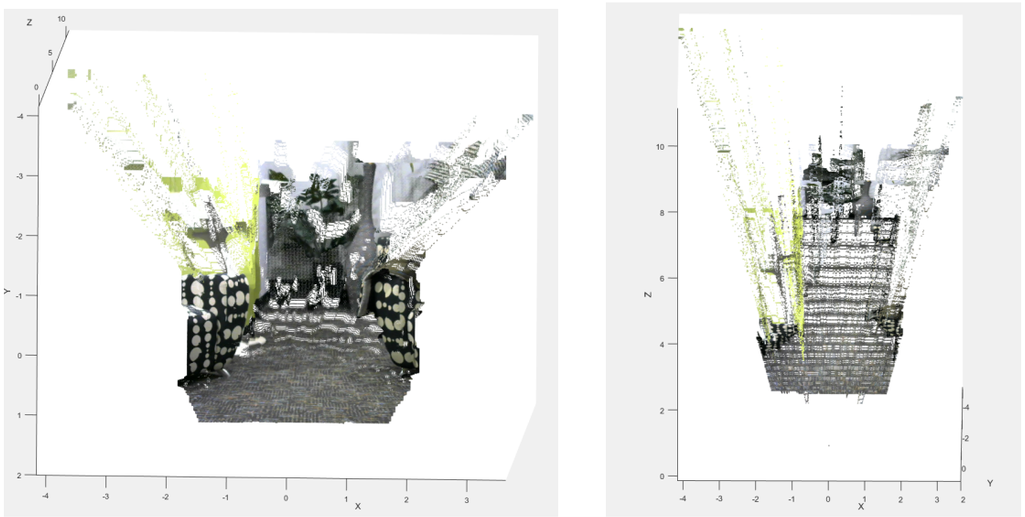
In the case of the 'Cones' image, we can notice again how the disparity values were correctly calculated for every single subject present in the picture. The ground truth is only more defined than our model thanks to an higher resolution.
Every single cone in the image was represented with a gradually lower disparity, just as it happens in the ground truth: no depth details were lost in the process.
Summarizing, the precision of our disparity calculation algorithm is extremely high, but it is only penalized by the reduced resolution necessary to improve general performance and reduce noticeably outliers.

[4] vision.middlebury.edu/stereo/data/

4.2 Testing 3D models

The model obtained from the pictures we captured have been tested simply through the comparison of the distances and sizes of objects measure in meters in the point cloud with real measurements taken on the field.
The result obtained are realistically accurate, representing quite well the scene depicted and interpreting correctly even extreme perspectives.
In Picture 23 is shown an example of scene we captured. It represent a couple of 0.8 meters high armchairs positioned at a distance of approximately of 3.5 meters from the camera and some objects in the background, positioned against the wall, that is 7.2 meters away.
In Pictures 24 we can see two views of the 3D model of the same scene. Exploiting the X, Y and Z axis depicted, it is possible to roughly measure the features expressed above, .
The estimated values of the armchairs height, their distance from the X axis and the distance of the background wall from the same axis are respectively 0.7, 4 and 8 meters.
This same level of precision is achieved in all the scene we captured and the results are quite satisfying.

Figure 23: Original image



(a) Sight from above

(b) Front sight

Figure 24: Views of the 3D model

5 Future Development and Possible Applications

There are many real world applications for the technology we have explored with this project.

The rise of autonomous vehicles, be these cars, planes or drones, has increased the demand for a fast, robust, reliable and cheap solution for the analysis of the spatial environment in which these vehicles must move. Stereo vision is being explored as a possible way to achieve good results with very inexpensive and easily available hardware. An example of this is the SkyDio R1[5] drone that flies autonomously avoiding obstacles by using a 360 degree stereo analysis of its surroundings.
In this area of application, the speed of the system are crucial for a real time reliable result that will allow vehicles to react quickly. We have seen that great improvements can be achieved when correctly encoding an algorithm. Future developments of this project could include a more in depth analysis of the execution with possible implementations in lower level languages such as C++ to have more control over the execution paradigm and speed up the calculations.
Other professional applications exist such as 3D mapping of spaces: building interiors, construction sites or even wider geographic areas (for example areas affected by natural disasters). In these cases, speed is not as relevant as maybe accuracy might be. For these kind of implementations we should focus a=our attention to the improvement of the qualitative result of our algorithm. More preprocessing and postprocessing could allow us to extract better defined models with smoother surfaces and less outliers.

Apart form improving the existing functionalities, it would be interesting to add something new. An example could be to implement a system that allows us to iteratively add points to our point cloud (the 3D model) as we take images from different angles. This would allow us to generate complete 3 dimensional objects without the limitation of a single viewpoint.

[5] https://www.skydio.com/product/