Stereo Vision using the OpenCV library

Sebastian Dröppelmann
Moos Hueting
Sander Latour
Martijn van der Veen

June 2010

Contents

1 Preface

Stereo vision is one of the key subjects in current computer vision research. Stereo vision can be best described as taking two viewpoints in a 3D world, comparing the distance between corresponding pixels (representing objects) in both images and taking this distance as a measurement for the real distance of the objects to the camera. Such distance information is retrieved by a dense stereo algorithm of which the output is often a disparity depth map. A disparity depth map is a 2D image where the color of each pixel represents the distance of that point to the camera. In other words, light pixels are near and dark pixels are far.

Depthmaps are interesting because they can be used for various purposes:

3D modeling of 2D images When you take two 2D images of a 3D environment and calculate the depthmap, you can create a 3D model of the scene by using the depth as the third dimension.

Recognizing front objects When you apply segmentation based on the depthmap, you can distinguish objects that are situated in the front of the scene.

Tracking of objects When you have a depthmap it is easier to track an object because you have additional segmentation possibilities. You can create segments of pixels that are near based on the depth of the pixels and their adjacency.

As information about the environment in path planning A depthmap supplies additional information for path planning.

2 Theory

2.1 Camera calibration

Cameras have some parameters that defines how real-world objects are projected on the image plane. The four most important parameters define the *focal length* in x and y direction and the *possible displacement* of the imagecenter away from the optic axis. The focal length ideally is the distance between the *center of projection* ($C1$ and $C2$ in 2.2) and the *image plane* ($R1$ and $R2$). These parameters are summarized in the following (homogeneous) matrix, which is called the *intern camera matrix*:

$$\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

If we choose a real-world coordinate system different from the coordinate system centered at the cameras center of projection, we have another matrix, called the *extern camera matrix*, which defines the transformation from this coordinate system to the camera coordinate system. This second matrix contains rotations in the x, y and z direction, and a translation. This alternative coordinate system becomes handy later on, when we will choose our own coordinate system based on the chessboard. Together, the intern and extern camera matrix form the basis for our mathematical model of the camera, and could be used to calculate the 3D real-world coordinates out of the 2D image coordinates, and vica versa. Calibration tries to determine the parameters in these matrices.

When working with camera's and in particular cheap webcams, often there will be some distortion of the grabbed images due to imperfections of the camera. The two most important distortions are radial distortion, caused by a lens being more spherical than the ideal parabolic lens would be, and tangential distortion, caused by the lens not being fully parallel to the image plane. Fortunately, both distortions can be resonable undone with a simple mathematical reprojection. The distortions can be characterized by a Taylor series of two terms for the lens shape and a reprojection for the skew image plane. For brevety, we omit the mathematical formulas and refer the interested reader to the OpenCV book [**?**].

The (five) parameters needed to undo the distortion can be calculated if we could find a raster of points that lay on straight lines in the real world. For this purpose often a chessboard pattern on a flat surface is used. Based on the distorted raster found in one picture of a chessboard pattern one could determine both distortions and, by using the determined parameters, mathematically reconstruct all original images made with the camera.

When working with *stereo* vision we also need to know the spatial relation between the two cameras. Using the algorithm proposed by Zhang [**?**] we can automate this process by showing both cameras different orientations of the same chessboard. A simple cornerfinding algorithm can recognize the black-white intersections of the chessboard squares, see also section 3.1.

The chessboard used can be of any size N by M. This way for each image containing a chessboard we have $N * M$ points for which we know the image coordinates in both cameras. Using the disparities of these points between both cameras, we can estimate the rotation/translation matrix (called E) which transforms the coordinate system of the left camera to the coordinate system of the right camera, or the other way around. Using this transformation, we could transform the physical coordinates of an object as seen by the left camera to the physical coordinates as seen by the right camera. The OpenCV stereo calibration function will also calculate a matrix (called F) which could transform (2D) image pixel coordinates from left to right image, or vica versa.

We have chosen not to describe the calibration algorithm fully here. See the article by Zhang for more information [?].

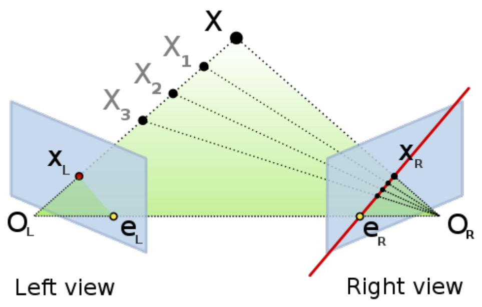


Figure 2.1: Epipolar geometry

2.2 Epipolar geometry and rectification

Epipolar geometry (figure 2.1) is used in stereo vision to limit the searching space when looking for matching points in both images. A point X in 3D space is seen in the left view, which we will call the source image, as a point x, which is on the line between the left camera's focal point O_L and point X. This line is seen in the right view, which we will call the search image, as a line. This is called an *epipolar line*. Given both the cameras internal and external matrices and a point x_A we can generate an epipolar line corresponding to this point in the search image. This constrains the search space to this 1D line. However, this means that for each pixel in the source image, we have to calculate the corresponding epipolar line in the search image. It would be much more convenient if each epipolar line was on the same line as the pixel it corresponded to. It is possible to transform the images in such a way that the epipolar lines are parallel and horizontal, and that process is called rectification.

Figure 2.2 demonstrates the process of rectification. The points E_1 and E_2 are called the *epipolar points* of both images. All epipolar lines intersect the epipolar point of a given image. However, when the retinal planes of both cameras lie on the same plane, the epipolar points move to infinity and the epipolar lines in one image become parallel to one another. The line between the optical centers of both retinal planes is called the baseline. When the epipolar points lie at infinity, we can see that the epipolar lines are parallel to this baseline. That means that if the baseline is parallel to the X-axis, the epipolar lines are horizontal.

2.2.1 Algorithm

We will give the basic gist of the rectification algorithm here.

1. First both camera matrices are separately factorized into three parts:
    - The internal camera matrix A_o
    - The rotational matrix R_o that gives the rotation between the camera's frame of reference and the world frame of reference
    - The translation matrix t_o that gives the translation between the camera's frame of reference and the world frame of reference

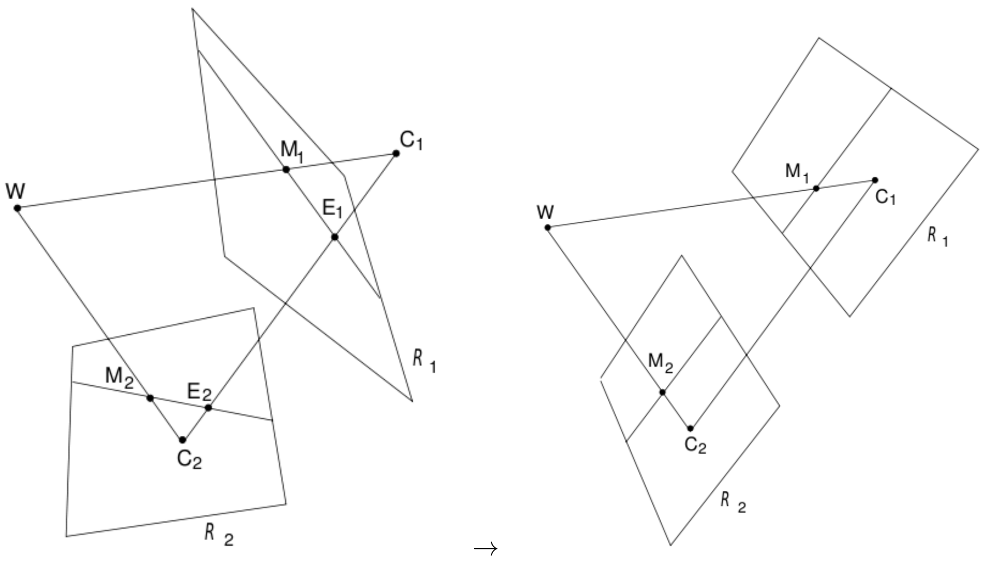


Figure 2.2: Left: unrectified cameras, right: rectified cameras. W represents the point in 3D space, C_1 and C_2 represent the camera focal points, M_1 and M_2 are the projections of W on the retinal planes of the cameras and R_1 and R_2 are the retinal planes of the cameras. In the left image, the points E_1 and E_2 represent the epipolar points of both images. In the right view, these points lie at infinity. After rectification, the epipolar lines are colinear and horizontal

This means $P_o = A_o[R_o \mid t_o]$

2. The new rotational matrix R_n is constructed. This matrix makes sure that in the new 'cameras' reference systems, the x-axis is parallel to the baseline. The baseline is simply the line between the two optical centers, which can be retrieved from P_o.
3. The new internal camera matrix A is constructed. This can be chosen arbitrarily and in this algorithm the mean between both original inner camera matrices $(A_{o1}, Ao2)$ is used.
4. The new camera matrices are now $P_{n1} = A[R \mid -Rc_1]$ and $P_{n2} = A[R \mid -Rc_2]$. The optical centers are unchanged.
5. A mapping is created that maps the original image planes of P_{o1}, P_{o2} to the new P_{n1}, P_{n2}. Because in general the pixels in the new images don't correspond to integer positions in the old images, bilinear interpolation is used to fill up the gaps.

For a complete and in depth description of each step, please see [**?**].

2.3 Dense Stereo

Dense stereo combines the two images you get from the rectification process and takes the position of the pixels in the left image to output the corresponding pixel location in the right image. With this method we calculate the pixel's distance from the camera. The depth is then translated to a depth map where points closer to the camera are almost white whereas points further away are almost black. Points in between are shown in gray-scale, which get darker the further away the point gets from the camera. See Figure 2.3 for an example depth map with the original image. To achieve this, a whole list of algorithms that do the trick is available.[1]

Most of these algorithms are based on 4 principles:

- Graph Cut
- Believe Propagation
- Region Based / Block Matching

[1]A good overview can be found at http://vision.middlebury.edu/stereo/eval/

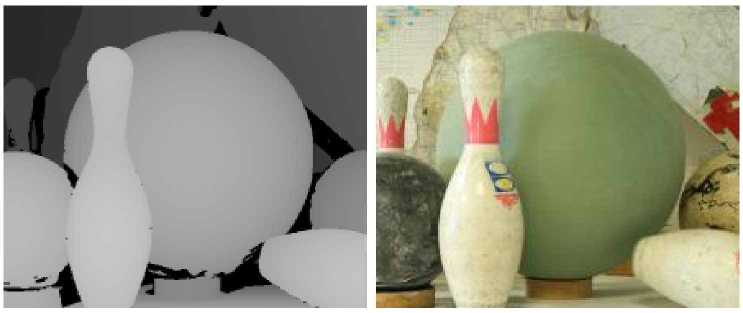


Figure 2.3: A depthmap with the corresponding picture, gray values show the depth of the image

- Dynamic Programming

The algorithms we are using are namely:

- Graph Cut
- Block Matching
- Semi Global Block Matching

These algorithms have to deal with the some problems in their calculations. The main task of dense stereo algorithms is to find the corresponding pixels in the match image starting from pixels in the base image. While doing this the algorithms have to keep track of occluded pixels, which are pixels that are found in one of the two images but not in the other.

2.3.1 Matching

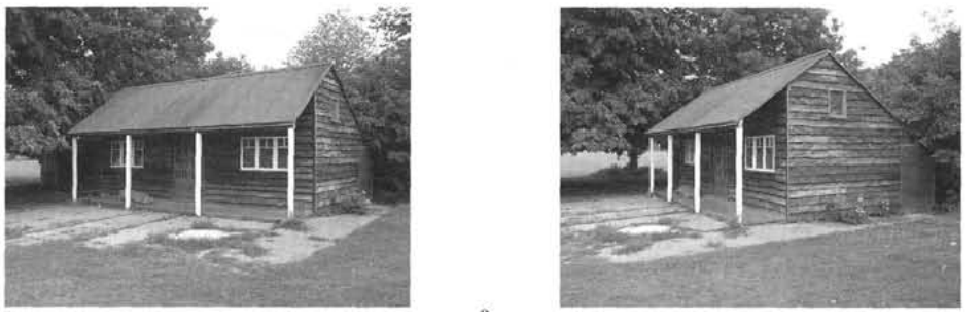


Figure 2.4: On the left image the window is completely on the left of the rightmost pole, whereas in the right image, the window is completely on the right side of the rightmost pole

The main goal of a dense stereo algorithm is matching one point in one image to the corresponding point in the other image. During the matching there are several tasks that the algorithm has to perform. At first it has to compare the epipolar lines of the images pixel by pixel. For every pixel on one line you have to find the counterpart on the corresponding epipolar line in the other image. Often the pixels aren't in the same order, for example if there is a lamp pole in front of a house, things that lie on the left side of the lamp pole in the left picture could sit on the right side in the right picture. See figure 2.4.

Another problem is occlusion of pixels. As we can see in figure 2.5, in both examples pixel P_O is visible for camera O_L, but is occluded in the view of camera O_R. This has to be caught and handled.

Because the OpenCV library that we have chosen has the basic implementation of a *Graph Cut* algorithm from Kolmogorov[**?**], we will start with that specific algorithm. It is not the best algorithm for this task but because of his general usability and it's high profile it is a good algorithm to start with. Furthermore we will test the implementations of the standard *Block Matching* algorithm, which is faster but not that precise. Also we will test a quite new implementation of the *Semi-Global Block Matching* algorithm, which was recently implemented in OpenCV.

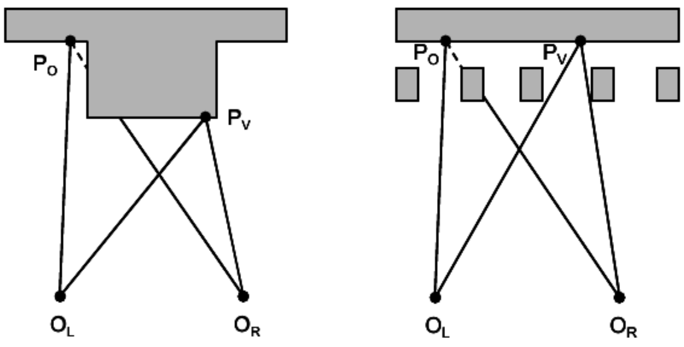


Figure 2.5: Two examples of occlusion problems when working with stereo vision

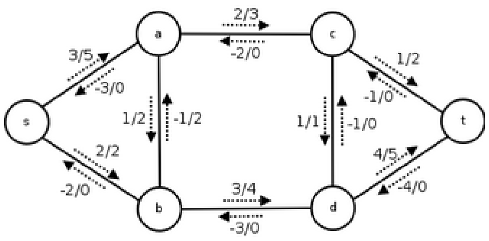


Figure 2.6: Flow network

2.3.2 Graph Cut Theory

Normal stereo matching algorithms try to match a pixel in the left image to a pixel in the right image based on some individual property like color. Although this is a fast and reasonably accurate process it does not deal with interlinear consistency. An algorithm that incorporates interlinear consistency takes in account the interpixel properties like adjacency. This results in a much more accurate disparity depth map. In order to maintain interlinear consistency you want to connect horizontally adjacent pixels and add some sort of weight on that connection to make it expensive to break. Furthermore you want to determine the disparities in the entire picture that disrupt those neighbouring relations as little as possible. [?] states that given some energy function you can construct a graph where the labeling with the lowest energy is equal to the minimum cut on that graph. In that respect you can look at the stereo correspondence problem as a labeling problem where you have to assign disparity labels. We will explain this in various sections below.

Graph theory: flow networks:
Figure 2.6 shows a directed graph $\mathcal{G}$ which can be formaly defined as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$. $\mathcal{V}$ depicts the set of vertices or nodes in the graph, whereas $\mathcal{E}$ depicts the set of edges between vertices.

Graph theory applied to stereo vision:
Figure 2.7 describes the process of constructing a graph $\mathcal{G}$ as discussed in the previous section from a

1. Create a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$
2. $\forall_{p \epsilon \mathcal{P}}$ add p to $\mathcal{V}$
3. add terminals f^0 and f^a to $\mathcal{G}$
4. $\forall_{p,q \epsilon \mathcal{N}}$ add $\{p,q\}$ to $\mathcal{E}$
5. $\forall_{p \epsilon \mathcal{V}}$ add $\{f^0, p\}$ and $\{f^a, p\}$ to $\mathcal{E}$
6. $\forall_{\{p,q\} \epsilon \mathcal{E}}$ set weight w for $\{p, q\}$

Figure 2.7: Steps for constructing a graph from picture $\mathcal{P}$

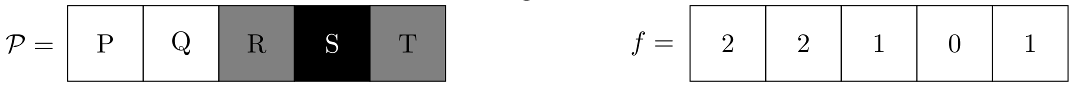
picture $\mathcal{P}$. $\mathcal{N}$ depicts a set of horizontally adjacent pixel pairs.

Weights:

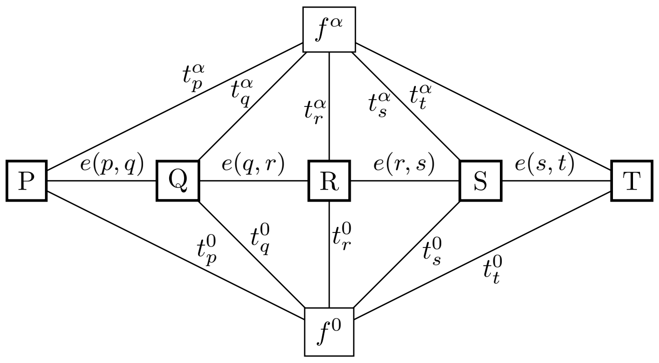
$$E(f) = E_{data}(f) + E_{smooth}(f) + E_{occ}(f) \tag{2.1}$$

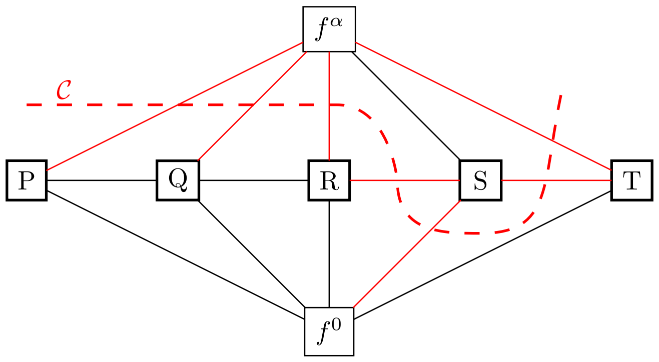
When two adjacent pixels have similar intensities it is likely that they are part of the same object, this means that during the labeling of disparities you would *prefer* to assign the same label to both pixels. The weights in $\mathcal{G}$ encode this information, the higher the weight value the less you want to break the pixels apart.

Figure 2.8: Picture $\mathcal{P}$



$$f_p^C = \begin{cases} \alpha & \text{if} \quad t_p^\alpha \epsilon \mathcal{C} \\ f_p & \text{if} \quad t_p^\alpha \epsilon \mathcal{C} \end{cases} \forall p \epsilon \mathcal{P} \tag{2.2}$$





$$f = \boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}$$

Figure 2.9: Graph-Cut step 1 with $f^\alpha = 1$

For a more in depth description of the graph cut algorithm, see [REFNAARANDERGRAPHCUT-PAPER]. For more details on the implementation in OpenCV, see **??**.

2.3.3 Semi Global Block Matching Theory

Semi global block matching is built on the idea that you use mean data from blocks of pixels as pixel energy and then build up a cost array over the costs of all disparities from the minimum disparity to the

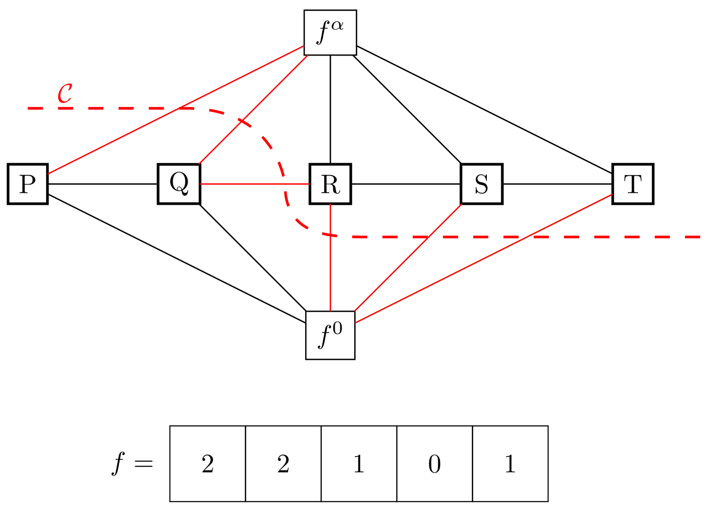


Figure 2.10: Graph-Cut step 2 with $f^\alpha = 2$

maximum disparity for all the pixels. Afterwards a search is done for the minimum path cost in several directions of the image. The minimum path cost is calculated from the pixel energy with penalties for the disparity difference to neighbour pixels along a path that ends in the pixel the disparity is searched for, where the one path with the lowest cost is the right one. This is done on subpixel level. After calculation of the disparities there are some refinement steps to filter out peaks and noise.
The Middlebury page divides disparity algorithms mainly into four parts, where some of the parts are optional. These parts are:

- cost computation
- cost aggregation
- disparity computation/optimization
- disparity refinement

For the SGBM algorithm these steps are filled in as follows:
Cost computation can be done from the intensity or color pixel values. Radiometric differences also have to be kept in mind, so the gradient can also be calculated.
Cost aggregation connects the costs of the neighbourhood of pixels to find the cheapest (thus matching) pixel in the compare image. This is done through a global energy function from all directions through the image.
Disparity computation calulates the real disparity from the previously calculated energy through a winner-takes-all implementation.
Disparity refinements are used to further stabilize the disparity map. This is done via peak filtering, intensity consistent disparity selection and gap interpolation. Also multibaseline matching is done through the fusion of disparities to circumvent streaking and add consistency. Because refinement is a separate part and has nothing to do with stereovision per se, we leave it out in the explanation.

Cost Calculation:
The *matching cost* for the blocks of pixels used for calculating the disparity in the OpenCV implementation are calculated through the implementation of the subpixel algorithm described in the paper of Birchfeld and Tomasi: "Depth Discontinuities by Pixel-to-Pixel Stereo" [?]. This is done by finding the correspondence $I_{m\mathbf{q}}$ of the matching image from the mean intensity of a block of pixels $I_{b\mathbf{p}}$ in the base image with the formula $\mathbf{q} = \rceil_{bm}(\mathbf{p}, d)$ where $\rceil_{bm}(\mathbf{p}, d)$ represents the epipolar line for $\mathbf{p}$ in I_b. The size of the area has influence on the robustness of the disparity map. Larger areas are more robust but fine structures get more blurred, because of the assumption of the same disparity over the whole area which isn't always true. The disparity cost $C_{BT}(\mathbf{p}, d)$ is calculated for a chosen squareblock of pixels $\mathbf{p}$ from $minX$ till $maxX$ (the limits of the block), where d represents the current disparity in the rectified images and x is the pixel in the current block. (2.3) This is done over all disparities between the given minimum

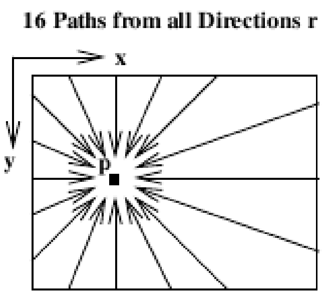


Figure 2.11: Directions of paths considered

and maximum disparity $minD$ and $maxD$.

$$C_{BT}[(x - minX) * maxD - minD + (d - minD)] \tag{2.3}$$

Cost Aggregation:
To smoothen wrong disparities calculated by the cost function and finding the right ones, the Energy of the disparity image is calculated from the sum of all pixel matching costs for the disparity of D. Different penalties, P_1, P_2 are applied for small and large disparity changes (2.4).

$$E(D) = \sum_{\mathbf{p}} (C_{BT}(\mathbf{p}, D_{\mathbf{p}}) + \sum_{\mathbf{q} \in N_{\mathbf{p}}} P_1 T[|D_{\mathbf{p}} - D_{\mathbf{q}}| = 1] + \sum_{\mathbf{q} \in N_{\mathbf{p}}} P_2 T[|D_{\mathbf{p}} - D_{\mathbf{q}}| > 1]) \tag{2.4}$$

The penalty for small changes in disparity P_1 is a constant whereas the penality for higher disparity changes P_2 is also used to catch discontinuities on intensity changes. This can be done through the use of the intensity of neighbouring pixels **p** and **q** in the base image I_b(2.5). But it always has to be ensured that $P_1 \leq P_2$. Now matching is '*only*' a question of minimizing the Energy $E(D)$.

$$\frac{P_2^{'}}{|I_{b\mathbf{p}} - I_{b\mathbf{q}}|} \tag{2.5}$$

Because 2D energy minimization would be a NP-complete problem, 1D Computation is used which can be calculated in polynomial time. To cover the equally important information of different directions of the image, 1D lines from '*all*' directions which end in the pixelblock you are searching the disparity for, are considered (see Figure 2.11) In the Hirschmüller paper 8-16 paths from all directions are recommended, whereas the standard algorithm in OpenCV calculates 5 or 8 directions due to the high memory cost. For our implementation we used 8 directions. For every path chosen the smoothed path cost $S(\mathbf{p}, d)$ is calculated through the traversing path cost $L_r(\mathbf{p}, d)$, were the cost represents the energy formula (2.4) along **an** arbitrary 1D path (2.6). Because the numbers can add up this way to really huge numbers, the minimum path cost of the previous pixel k is substracted. This way the numbers stay smaller and the path doesn't change it's minimum cost way, because the minimum cost of the pixel before is a constant. For precalculation all costs can be saved in an integer array of size $[W * H * D]$ and the aggregated costs are then saved in an equally sized array S.

$$\begin{aligned} L_r(\mathbf{p}, d) = C_{BT}(\mathbf{p}, d) + min(L_r(\mathbf{p} - \mathbf{r}, d), \\ L_r(\mathbf{p} - \mathbf{r}, d-1) + P_1, \\ L_r(\mathbf{p} - \mathbf{r}, d+1) + P_1, \\ \min_i L_r(\mathbf{p} - \mathbf{r}, i) + P_2) - \min_k L_r(\mathbf{p} - \mathbf{r}, k) \end{aligned} \tag{2.6}$$

Disparity Computation:
The base disparity map D_b from the base image I_b is caluclated by picking the disparity with the minimum path cost for each pixel. This is done with subpixel accuracy by picking the minimum of a quadratic curve through the neighbouring pixel costs. The disparity map D_m of the match image from I_m is calculated by traversing the epipolar line of pixel q in the match image. The disparity is then again the disparity of the lowest pathcost. To enhance the quality of the disparity map, you can calculate the disparities again but with I_m as base image and I_b as match image. After D_m and D_b have been calculated, a consitency

check is done between the two. If the consistency between the two is too great (e.g. due to occlusion) the disparity is set to invalid. Also unique constrained can be switched on, which enforces one on one pixel mappings.

3 Implementation

3.1 OpenCV

OpenCV is a library of programming functions for real time computer vision. By using this library we can constrain our tasks mostly to integrating various parts of OpenCV and expanding it where necessary. If we would not use OpenCV, we would had to implement a lot of basic computer vision functions, so there would probably be not enough time to achieve our goal. OpenCV is fast library written in C/C++, and has python binding as well, which we will use to decrease the risk of programming errors.

3.2 Calibration

OpenCV has a separate function for calibrating a set of stereo cameras. This function[2] uses as input a list of coordinates of points in the left image and the coordinates of the same points in the world in the right image, and a set of coordinates where these points actually lie in the real world (3D coordinates).

3.2.1 Chessboard points

As input for the calibration, we use the intersection points of a chessboard. These points on the chessboard are recognized by the stereo cameras and a list of chessboard piececorners coordinates is returned, see Figure 3.1. The corners are being found by by first thresholding the image and then trying to locate a grid of black-white transitions of the given count. This gives an approximation on pixel level, which is not very accurate.

Because we are looking for the relationship between the two cameras and not the relationship between a camera and the actual world, we can choose the origin for these "real world points" however we like. As we are using the chessboard, it is very convenient to choose the x and y axes along the sides of the chessboard, so that the first intersection lies at $(0, 0, 0)$ in the real world.

To improve the corner locations, one could use the FindCornerSubPix() function, which uses Harris cornerdetection. To fully cover the algorithm for this correction algoritm would require a lot of linear algebra and isn't covered at that depth here[3]. If the reader has no background in linear algebra, the next paragraph can safely be skipped.

We give a short overview: For each approximated location, which are pixel coordinates, make vectors from some not yet determined subpixel location q to other points p, some pixels away from q. Take the dot-product of this vector with the gradient vector for location p. This dot-product is equal to zero in two situations: if p is somewhere on a square, the gradient vector and thus the dot-product is zero, and when qp is located exactly on an edge, the gradient vector is perpendicular to qp and thus the dot-product is zero. If q is the correct subpixel corner location, then all dot-products are zero. FindCornerSubPix() sets these dot-products to zero and tries to solve the equations, the best solution for q is the new subpixel location.

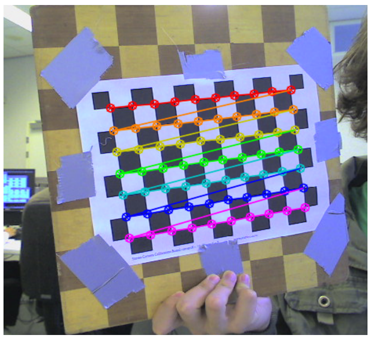


Figure 3.1: The recognized chessboard-points by the OpenCV function 'findChessboardCorners'

[2]See stereoCalibrate in the OpenCV documentation

[3]See the OpenCV documentation, chapter 10, for the function findChessboardCorners for mathematical background

3.2.2 Output

The calibration function outputs a set of camera matrices, a set of distortion coefficients for both cameras (to correct for lens distortion) and a translation/rotation matrix relating the first camera to the second. This output is used by the rectification algorithm explained in section 2.2.

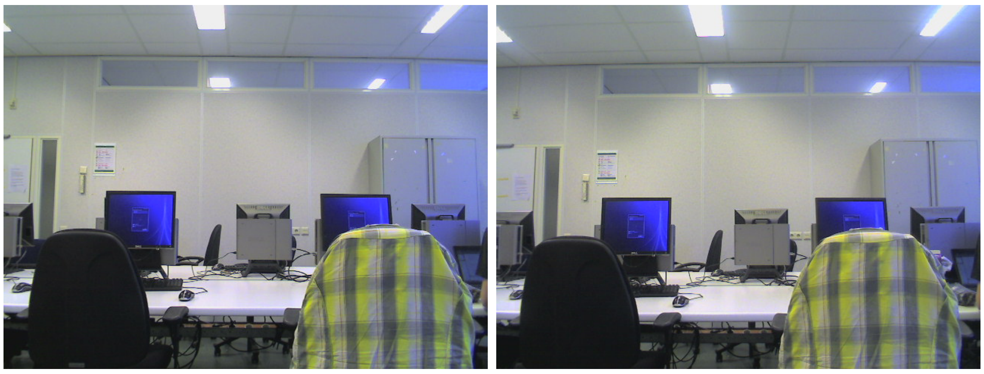


Figure 3.2: Unrectified images

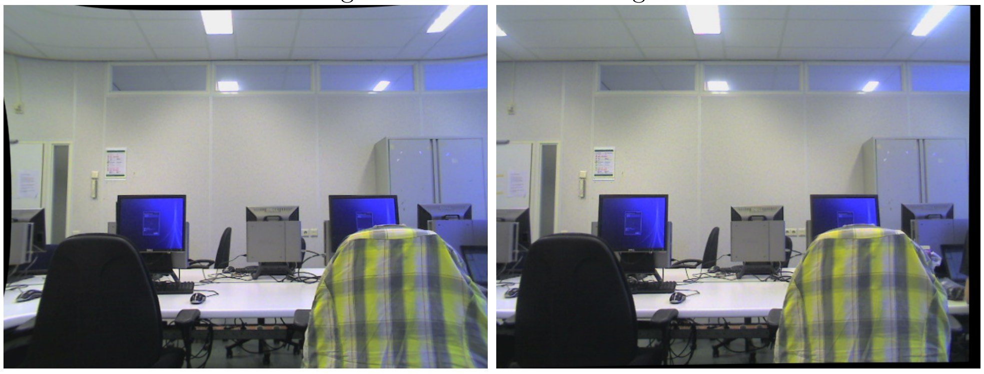


Figure 3.3: Rectified images

3.3 Rectification

The algorithm used for rectification in OpenCV is exactly as we have explained in the theory section (2.2).

3.4 Dense Stereo Algorithms

During our project we first started with using the standard algorithms that are already implemented in OpenCV. As described in the theory part of the paper (see 2.3) we used three algorithms, *Graph Cut*, *Block Matching* and *Semi Global Block Matching.* To test these algorithms we used the Tsukuba dataset from Middlebury [?]. This set with rectified images and their corresponding depth maps is the standard for testing and comparing different methods of stereo correspondence.

3.4.1 Graph Cut

The graph cut algorithm is one of the most popular algorithms in the generation of disparity depth maps. We used the standard implementation in OpenCV which is an adaptation of the Kolmogorov paper [?]. Our first results almost reach the quality of the ground truth depth map provided by the Middlebury homepage[?]. See Figure 3.4. Section 2.3.2 talks about the theory behind graph cut.

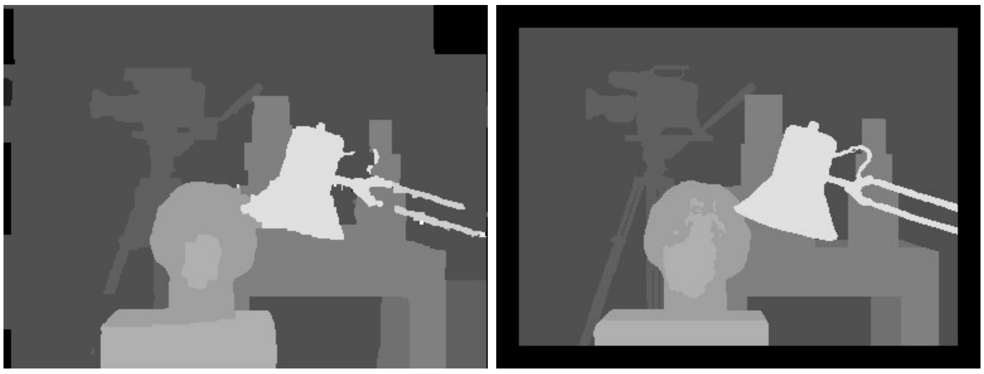


Figure 3.4: left: Our GC depthmap, right: Middlebury Ground Truth. Gray values convey depth, lighter gray is closer to the camera. Black values are “invalid”, meaning that using the algorithm no valid depthvalue could be calculated

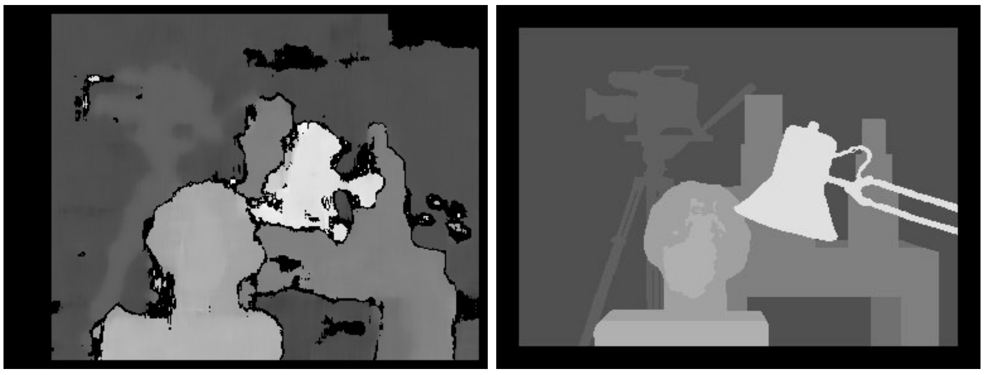


Figure 3.5: left: Our BM depthmap, right: Middlebury Ground Truth

3.4.2 Block Matching

The Block Matching algorithm is much faster than the Graph Cut Algorithm, but the quality of the results is not as good. The quality of the algorithm depends greatly on the configuration of the parameters and the use of pre- and postfilters. See Figure 3.5 for comparison with the ground truth from Middlebury. We couldn’t find any good block matching example to compare with but with the ground truth image you will get the idea of how it has to divide the image into different depths.

3.4.3 Semi Global Block Matching

The Semi Global Block Matching algorithm is quite new in the OpenCV library. It was implemented in version 2.1 which is the current version at the time this article is written. It is more precise and almost as fast as the standard block matching algorithm but the python binding is not yet completed and integrated into OpenCV. Because of this we wrote this part in C++. Just as the standard Block Matching the algorithm still needs to be tuned right to return the optimal depthmap. See Figure 3.6 for comparison with the ground truth. The tuning of the algorithm is one of the most important steps to get good results. The tuneable parameters are:

- min and max disparity
- SAD size
- PreFilterCap
- disp12MaxDiff
- uniqueness
- speckleWindowSize and speckleRange
- Original

min and max disparity are the minimum and maximum disparity values that can be found. Between these the best one is chosen for the disparity image. In our implementation we set the minimum disparity to 0 and the maximum disparity we chose the image width/8, because it returned the best results
SAD size is the size of the blocks which is considered for the Energy calculation of the algorithm. The manual suggests values between 5 and 21 pixels. The value has to be an uneven value. We chose a size of 9 to keep the calculation as fast as possible without too much loss of quality.
PreFilterCap is the value for the Tomasi cost function to cap the values at [-PreFilterCap, PreFilterCap] intervals. In the OpenCV implementation the values given can vary between 0 and 63. We chose for 63 because then you had the minimum noise inside the image.
disp12MaxDiff is the maximum value in the left-right disparity check. We set it to 2 to get results that where acceptable. More didn't make a difference and less gave much noise.
uniqueness is the switch to enable the uniqueness check. We switched it on.
speckleWindowSize defines the maximum region which is considered as speckle / peak. In our configuration we set the value to 100. A value less gave too much noise and a higher value didn't really change that much. OpenCV suggests a value between 50-200.
speckleRange defines the disparity difference that is considered as speckle. This has to be a value devidable by 16 and the OpenCV manual suggests 16 or 32. A value of 32 gave good results.
Original at last defines if you want to use 5 or 8 paths for the cost aggregation. We have chosen the suggested 8 paths configuration.

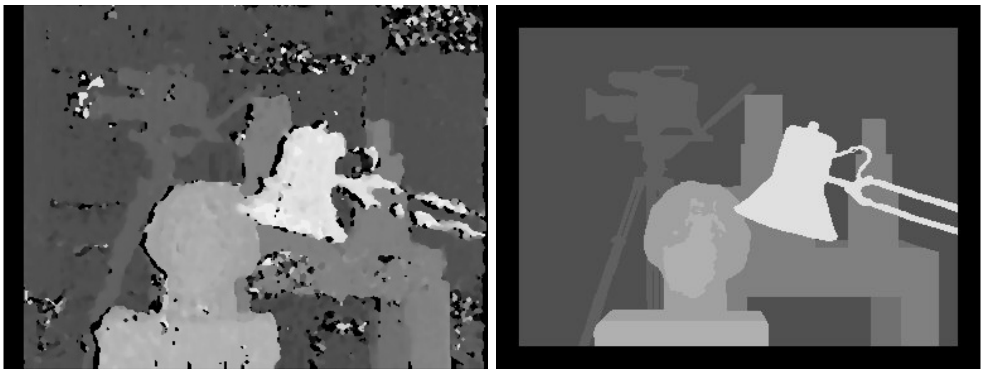


Figure 3.6: left: Our SGBM depthmap, right: Middlebury Ground Truth

4 Practical problems

4.1 Webcams

We had to make the two webcams work on Linux. We had to compile our own drivers in order to receive a live feed from both cameras at the same time. Furthermore, because the webcams could not be moved relatively to each other after calibration, we created a rig in which both cameras were fixed.

4.2 OpenCV

We were completely unfamiliar with the library OpenCV, which we have used for all of our calculations. We were quite accustomed to MATLAB, so it took us a while to switch to programming linear algebra in OpenCV. First off, it took us a while to get it working on various systems. The library is written in C, but because we used Python bindings, it was quite hard to debug errors thrown by the library functions.

Furthermore, in order to be able to use the SGBM functionality implemented in the latest release of OpenCV, we had to compile our own build of the library. As of yet, there are no binary packages available for this release. See section 3.1 for a more in depth description of OpenCV.

5 Applications

6 Discussion

7 Planning

This was our original planning from which we did not have to deviate greatly.

- Week 1
  - Reading literature
  - Getting webcams to work
  - Choosing dense algorithm
- Week 2
  - Implementing
    - Dense disparity map algorithm working
    - Camera calibration using epipolar geometry
    - Rectification of images
  - Halfway report
- Week 3
  - Fine tuning camera calibration
  - Cropping of rectified images
  - Fine tuning parameters of dense stereo
  - Completely understand the algorithms
  - Depth map normalize
- Week 4
  - Optimizing and testing
  - If there's enough time left
    - Generate 3D image of environment
    - Remove background using dense disparity map

8 Tasks

- Martijn and Moos
  - Camera calibration
  - Epipolar geometry
- Sander and Sebastian
  - Finding corresponding points
  - Generating depth map

9 NOTES MARTIJN (DELETE LATER)

- Calibratie beschrijft nog steeds niet echt hoe de stereo matrices (F en E) worden berekend, dat wil Rein eigenlijk wel. Maar daar wordt het stuk wel weer langer van.
- De demo programma's worden nu helemaal niet beschreven. Voorstel: sectie 'Applications' toevoegen en kort progjes beschrijven, + screenshot
- Er mist nog Conclusie.
- Er mist nog Reflectie.