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 the computer vision research. Stereo vision can be best described as taking two viewpoints in a 3D world, comparing the distance between the position of an object in both images and relating that to the distance of the object to the camera. Such 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 is directly linked to the distance of the pixel on that coordinate in the original image, in other words if an object is white it is depending on the implementation nearer or further away than a darker object. Our goal is to generate such a depth map from two images taken with two webcams.

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.

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.

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

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

2 Practical problems

2.1 Webcams

We will have to make the two webcams work on Linux. Ideally we would have a live feed from both webcams at all times.

2.2 OpenCV

We will have to get acquainted with the library OpenCV. See section 4.1.

3 Theory

3.1 Camera calibration

When working with stereo vision we need to know the spatial relation between the two cameras, and we need to get rid of radial distortion due to the imperfections of the lens. Using the algorithm proposed by Zhang [5] we can automate this process by showing both cameras different orientations of a chessboard. A simple contrastbased algorithm can recognize the black-white intersections of the chessboard squares. The chessboard used can be of any size N by M. This way for each chessboard we show we have $N * M$ points we know for both cameras. Using the disparities of these points between both cameras, we can calculate a rotation/translation matrix which transforms the coordinate system of the left camera to the coordinate system of the right camera, or the other way around.

We have chosen not to describe the algorithm fully here. See the article by Zhang for more information [5].

3.2 Epipolar geometry and rectification

Epipolar geometry 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 image A, which we will call the source image, as a point x, which is on the line between camera A's focal point and point X. This line is seen on image B, 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 the process is called rectification.

Figure 3.1 demonstrates the process of rectification. The points called 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.

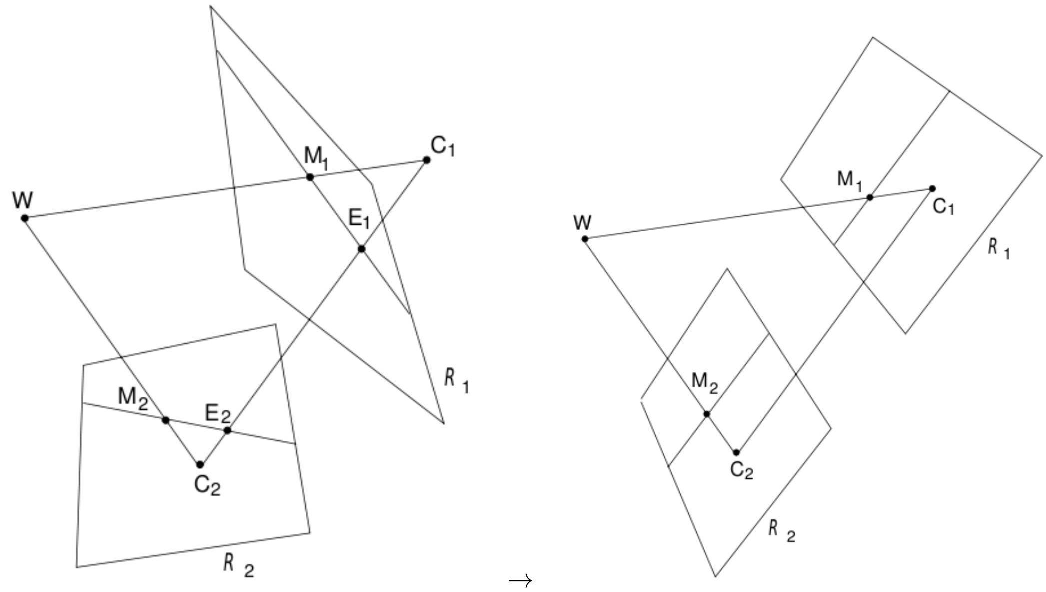


Figure 3.1: Left: unrectified cameras, right: rectified cameras. After rectification, the epipolar lines are colinear and horizontal

3.2.1 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

   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].

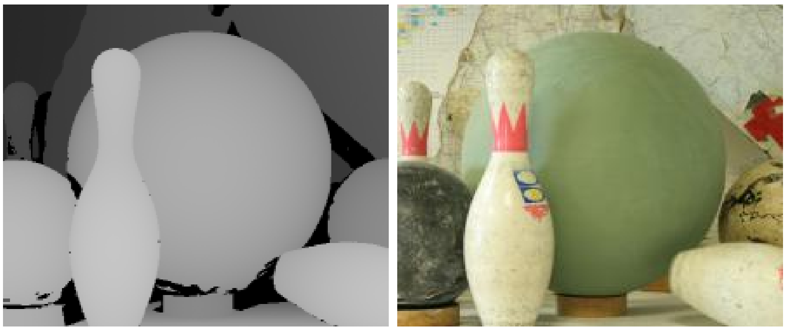


Figure 3.2: A depthmap with the corresponding picture, gray values show the depth of the image

3.3 Dense Stereo

Dense stereo combines the two images you get from the rectification and calculates the position of the pixels in the left image and outputs where the pixel is located 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 3.2 for an example depth map with 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
- Dynamic Programming

These algorithms have to deal with the following problems in their calculations

- Matching points in both images
- Occlusion

3.3.1 Matching

Figure 3.3: 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 such an 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 3.3.

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

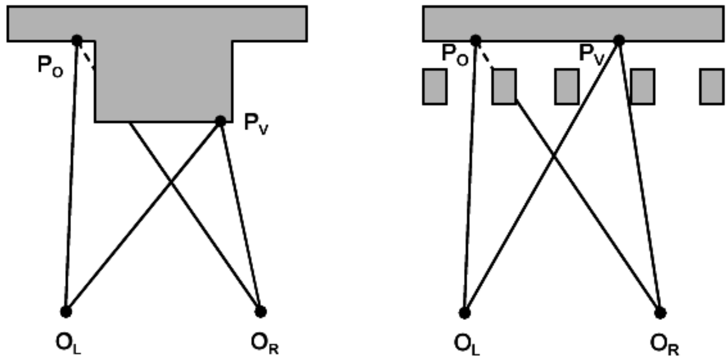


Figure 3.4: Two examples of occlusion problems when working with stereo vision

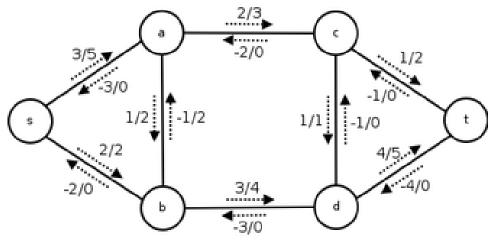


Figure 3.5: Flow network

Another problem is occlusion of pixels. Some things that are visible in one picture can be hidden behind objects in the other picture. This has to be caught and handled. Both examples in figure 3.4 show that the leftmost pixel of the left cam is not visible in the right cam view.

Because the OpenCV library that we have chosen has the basic implementation of a graph cut algorithm from Kolmogorov [3], we will start with that specific algorithm. It is a bit slow and not the best algorithm to handle occlusion, but because it is easy to use we will at first use this algorithm and later replace it with an implementation of the block matching algorithms also provided by OpenCV.

3.3.2 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 the interlinear consistency. The interlinear consistency forces you to also look at interpixel properties like neighbours and will result in a much more accurate desparity map.
In order to maintain the interlinear consistency you want to connect the pixels that are adjacent within a line and add some sort of weight on that connection to make it expensive to break that connection. Furthermore you want to determine the disparities in the entire picture that disrupt those neighbouring relations as little as possible. [ref] 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 correspondance 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 3.5 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:
[h!bt] Figure 3.6 describes the process of constructing a graph $\mathcal{G}$ as discussed in the previous section from a picture $\mathcal{P}$. $\mathcal{N}$ depicts a set of horizontally adjacent pixel pairs.

Weights:

$$E(f) = E_{data}(f) + E_{smooth}(f) + E_{occ}(f) \tag{3.1}$$

When two adjacent pixels have similair 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.

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

Figure 3.7: Picture $\mathcal{P}$

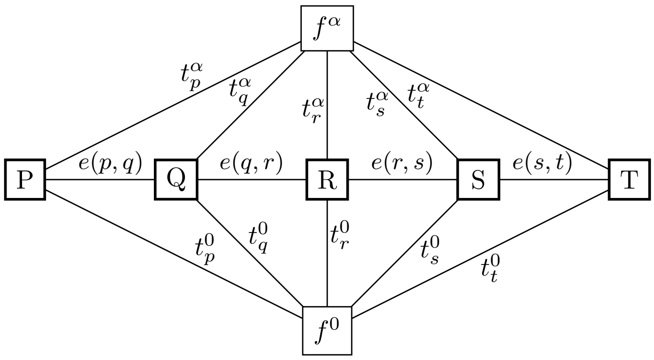
$\mathcal{P} =$

P	Q	R	S	T

$f =$

2	2	1	0	1

$$f^C_p = \begin{cases} \alpha & \text{if} \quad t^\alpha_p \epsilon \mathcal{C} \\ f_p & \text{if} \quad t^\alpha_p \epsilon \mathcal{C} \end{cases} \forall p \epsilon \mathcal{P} \tag{3.2}$$

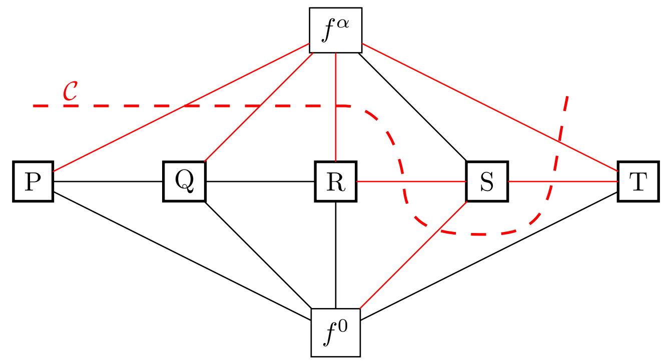


3.3.3 Semi Global Block Matching Theory

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 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

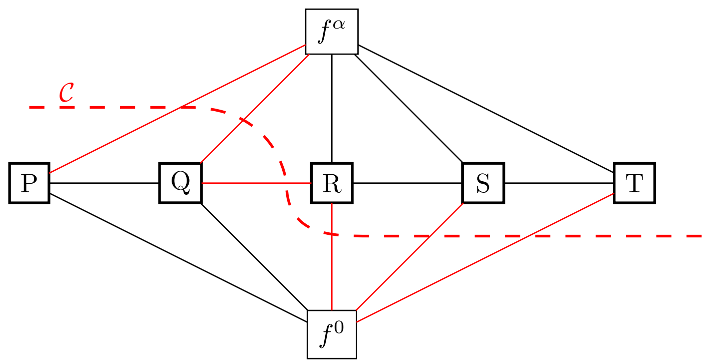
For the SGBM algorithm these steps are filled in as follows:
Cost computation can be done from the intensity or color pixel values. Radiometric differences also have to be kept in mind, so the gradient can also be calculated.
Cost aggregation connects the costs of the neighbourhood of pixels to find the cheapest (thus matching) pixel in the compare image. This is done through a global energy function from all directions through the image.



$f =$

1	1	1	0	1

Figure 3.8: Graph-Cut step 1 with $f^\alpha = 1$



$f =$

2	2	1	0	1

Figure 3.9: Graph-Cut step 2 with $f^\alpha = 2$

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" [1]. 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. (3.3) This is done over all disparities between the given minimum and maximum disparity $minD$ and $maxD$.

$$C_{BT}[(x - minX) * maxD - minD + (d - minD)] \tag{3.3}$$

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,

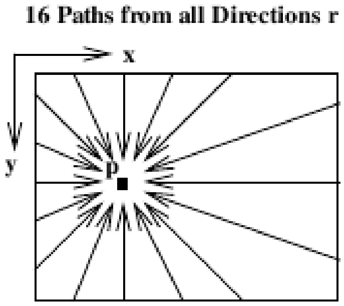


Figure 3.10: Directions of paths considered

P_1, P_2 are applied for small and large disparity changes (3.4).

$$E(D) = \sum_{\mathbf{p}} (C_{BT}(\mathbf{p}, D_{\mathbf{p}}) + \sum_{\mathbf{q} \in N_{\mathbf{p}}} P_1 T[|D_{\mathbf{p}} - D_{\mathbf{q}}| = 1] + \sum_{\mathbf{q} \in N_{\mathbf{p}}} P_2 T[|D_{\mathbf{p}} - D_{\mathbf{q}}| > 1]) \tag{3.4}$$

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(3.5). But it always has to be ensured that $P_1 \leq P_2$. Now matching is '*only*' a question of minimizing the Energy $E(D)$.

$$\frac{P_2^{'}}{|I_{b\mathbf{p}} - I_{b\mathbf{q}}|} \tag{3.5}$$

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 are considered.3.10 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 (3.4) along **an** arbitrary 1D path (3.6). 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.

$$\begin{aligned} L_r(\mathbf{p}, d) = C_{BT}(\mathbf{p}, d) + min(L_r(\mathbf{p} - \mathbf{r}, d), \\ L_r(\mathbf{p} - \mathbf{r}, d - 1) + P_1, \\ L_r(\mathbf{p} - \mathbf{r}, d + 1) + P_1, \\ \min_i L_r(\mathbf{p} - \mathbf{r}, i) + P_2) - \\ \min_k L_r(\mathbf{p} - \mathbf{r}, k) \end{aligned} \tag{3.6}$$

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.

4 Implementation

4.1 OpenCV

OpenCV is a library of programming functions for real time computer vision. By ussing this library we can constrain our tasks to integrating various parts of OpenCV and expanding it where possible. If we would not use OpenCV, we would not have enough time to achieve our goal. OpenCV is a C library but has python binding which we will use to decrease the risk of programming errors.

4.2 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).

4.2.1 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 coordinates is returned, see Figure 4.1. The algorithm for this recognition isn't covered here.[3]

Because we are looking for the relationship between the cameras and not the relationship between the cameras 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.

Figure 4.1: The recognized chessboard-points by the OpenCV function 'findChessboardCorners'

4.2.2 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 below.

[2]See stereoCalibrate in the OpenCV documentation
[3]See the OpenCV documentation for the function findChessboardCorners for more information

Figure 4.2: Unrectified images

Figure 4.3: Rectified images

4.3 Rectification

The algorithm used for rectification in OpenCV is exactly as we have explained in the theory section (3.2).

4.4 Dense Stereo Algorithms

During our project we first started with using the standard algorithms that are already implemented in OpenCV. These algorithms are namely:

- Graph Cut
- Block Matching
- Semi Global Block Matching

To test these algorithms we used the Tsukuba dataset from Middlebury [4]. This set with rectified images and their corresponding depth maps is the industry standard for testing and comparing different methods of stereo correspondence.

4.4.1 Graph Cut

The graph cut algorithm is one of the most popular algorithms in the stereo vision depth map generation. It is quite slow though and new algorithms return better results qua speed and depth recognition. We used the standard implementation in OpenCV which is written by use of the Kolmogorov paper from 2003[3]. Our first results almost reach the quality of the depth map provided by the Middlebury homepage[4]. See figure 4.4. Section 3.3.2 talks about the theory behind graph cut.

4.4.2 Block Matching

The Block Matching algorithm is much faster than the Graph Cut Algorithm, but the quality of the results until now are not as good. There are more possibilities to fine-tune the algorithm with pre and post filters which have huge impact on the quality and we still have to find the right configuration to get the best results. See Figure 4.5 for comparison with the Graph Cut algorithm from Middlebury. We couldn't find any good block

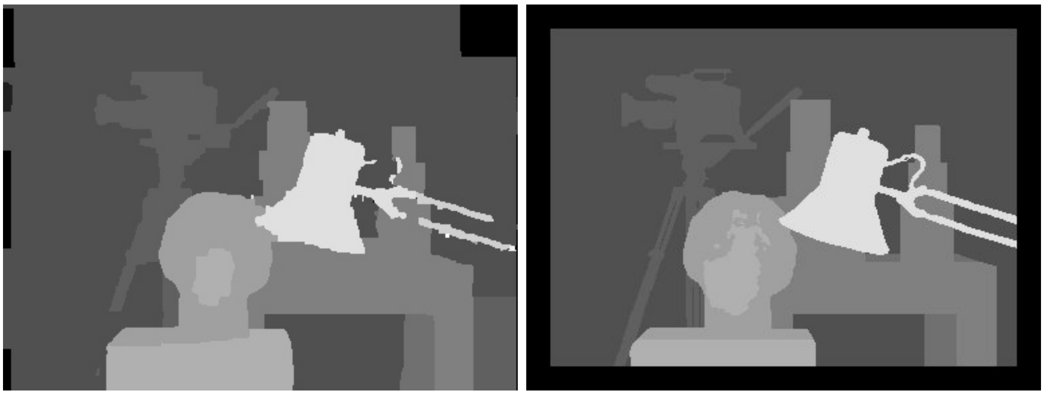


Figure 4.4: left: Our GC depthmap, right: Middlebury Ground Truth. Gray values convey depth, lighter gray is closer to the camera. Black values are “invalid”, that is using the algorithm no valid depth can

matching example to compare with but with the graph cut you will get the idea of how it has to divide the image into different depths.

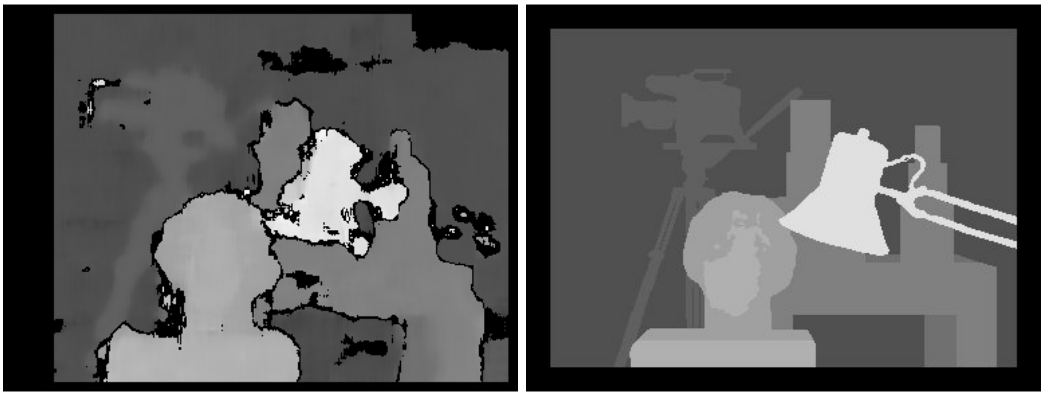


Figure 4.5: left: Our BM depthmap, right: Middlebury Ground Truth

4.4.3 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 faster than 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 4.6 for comparison with the GC algorithm. The tuning of the algorithm was one of the most important steps to get good results. The options you could tune were:

- 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 for a size of 9 to keep

the calculation faster and we didn't get much better results with higher values.
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. With a value of 32 we had good results so we kept that.
Original at last defines if you want to use 5 or 8 paths for the cost aggregation. We used the original suggested 8 paths so we turned on the boolean.

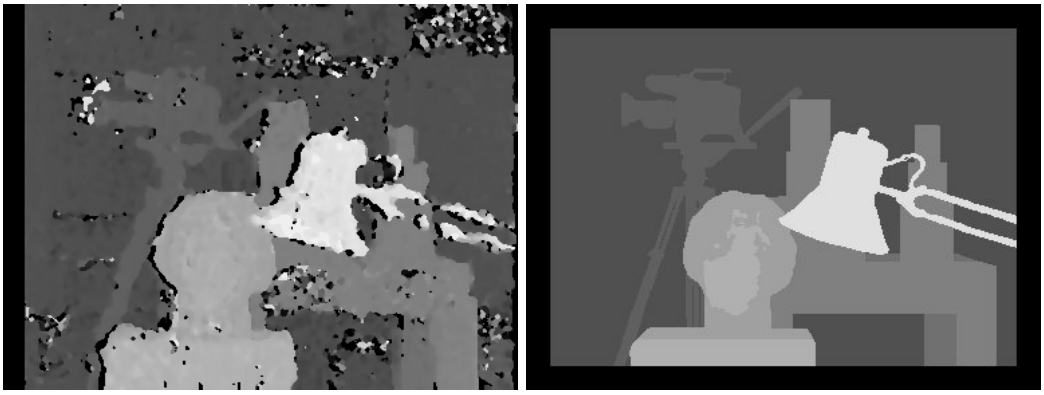


Figure 4.6: left: Our SGBM depthmap, right: Middlebury Ground Truth

5 Planning

- 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

6 Tasks

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

References

[1] S. Birchfield and C. Tomasi. Depth discontinuities by pixel-to-pixel stereo. *International Journal of Computer Vision*, 35(3):269–293, 1999.

[2] A. Fusiello, E. Trucco, and A. Verri. A compact algorithm for rectification of stereo pairs. *Machine Vision and Applications*, 12(1):16–22, 2000.

[3] Junhwan Kim, Vladimir Kolmogorov, and Ramin Zabih. Visual correspondence using energy minimization and mutual information. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 1033, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Daniel Scharstein and Richard Szeliski. vision.middlebury.edu. `http://vision.middlebury.edu/stereo/`, 2010.

[5] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *iccv*, page 666. Published by the IEEE Computer Society, 1999.