Physics-inspired models for parallelized texture-supported segment-based stereo computation

Diplomarbeit vorgelegt von
Waldemar Kornewald
aus
Kostschi/Kasachstan

angefertigt im 3. Physikalisches Institut
der Georg-August-Universität zu Göttingen

2011
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Introduction</td>
<td>vii</td>
</tr>
<tr>
<td>1 Stereo correspondence</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Epipolar geometry</td>
<td>5</td>
</tr>
<tr>
<td>2 Phase-based disparity</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Algorithm description</td>
<td>9</td>
</tr>
<tr>
<td>Filter bank</td>
<td>10</td>
</tr>
<tr>
<td>Gaussian pyramid</td>
<td>10</td>
</tr>
<tr>
<td>Binocular disparity</td>
<td>12</td>
</tr>
<tr>
<td>3 Segment-based disparity</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Algorithm overview</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Potentially occluded areas</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Segment-edge disparities</td>
<td>18</td>
</tr>
<tr>
<td>3.4 Inner-segment disparities</td>
<td>18</td>
</tr>
<tr>
<td>3.5 Interpolated disparities</td>
<td>19</td>
</tr>
<tr>
<td>4 Algorithm components</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Processing steps</td>
<td>25</td>
</tr>
<tr>
<td>5 Post-segmentation components</td>
<td>33</td>
</tr>
<tr>
<td>5.1 Edge disparity</td>
<td>33</td>
</tr>
<tr>
<td>5.2 Average edge disparity</td>
<td>35</td>
</tr>
<tr>
<td>5.3 Occlusion map</td>
<td>35</td>
</tr>
</tbody>
</table>
## CONTENTS

5.4 Inner disparities ................................. 37
5.5 Interpolation ................................. 39

6 Texture-supported segmentation 43
  6.1 Overview .................................. 43
  6.2 Texture detection ............................ 44
  6.3 Binary segment counting ....................... 48
  6.4 Smoothing ................................. 50
  6.5 Texture component for segmentation ........... 51
    Problem description ........................... 51
    Texture and color components .................. 52

7 Implementation 55
  7.1 Overview .................................. 55
  7.2 Parallelization ............................ 56
  7.3 Python->GPU compiler (py2gpu) .............. 57
    Introduction ................................ 57
    Parsing and PyMeta2 ......................... 59
    Usage ....................................... 61
    Conclusion (py2gpu) ......................... 61

8 Results 63
  8.1 Quality metrics ............................ 63
  8.2 Quantitative comparison ...................... 64
    Baby ...................................... 65
    Cones ..................................... 66
    Lampshade .................................. 66
    Midd ...................................... 66
    Monopoly .................................. 67
    Plastic .................................... 67
    Rocks ...................................... 67
    Teddy ...................................... 68
    Tsukuba .................................... 68
    Venus ...................................... 68
    Wood ...................................... 68
  8.3 Runtime comparison .......................... 68

9 Conclusion 83

10 Future work 85
  10.1 Texture classification ...................... 85
CONTENTS

10.2 Interpolation ............................................. 85
10.3 Speed optimizations ..................................... 85
10.4 Edge disparity smoothness constraints ................. 86
10.5 Integration into segmentation ......................... 86

A Usage of py2gpu ................................. 87

List of Figures .............................................. 91

Bibliography .............................................. 95
Acknowledgments

This diploma thesis would not have been possible without the help of the people who invested time into supporting me:

Prof. Florentin Wörgötter gave me the chance to work on this thesis and provided me with the freedom to explore ideas on my own while also guiding me in the right direction. He is also going to mentor me and two other friends in our future venture. Thank you very much! I would also like to thank Babette Dellen for mentoring me and for all the time she invested into introducing me into the matter and answering my questions and providing suggestions and reviewing my thesis. I’m also very thankful for the help provided by Alexey Abramov who put a lot of time into reviewing this thesis and providing improvement suggestions and discussing issues with the algorithm in general - although he didn’t really have to. My further thanks go to Thomas Wanschik and Johannes Dörr who always provided a helping ear and valuable suggestions and without whom I probably wouldn’t be writing this. Finally, I’d like to thank the whole work group which was a pleasure to work with. The atmosphere was very friendly and I will keep many nice evenings in my memory; talking about Japan and Asian culture with Poramate Manoonpong (who now has his own work group and funding, congratulations!), or joking around and discussing with Tomas Kulvicius (Big T ;) , and everyone else in the work group. Thank you all.
Introduction

Stereo correspondence is an important problem in computer vision and it facilitates solving many important problems in robotics. Stereo vision allows for acquiring depth information from two images. The human vision system accounts for a majority of our sensory input and a three-dimensional understanding of the world is required for many robotic systems. The algorithm presented here is specifically designed for two images (a "left" and a "right" image) taken with a stereo camera.

An important problem in the area of stereo correspondence is the detection of high correspondences in textureless scenes. Most algorithms only work well on textured images because they provide sufficient structure for matching. In the real world we often have textureless or weakly textured areas like, for instance, a table, a room’s floor and walls, a street, a house, or a metallic industrial environment. Often real-world scenes consist of a mixture of textured and untextured areas which means that a practical "real-world" stereo algorithm has to be able to deal with both kinds of scenes very well.

When talking about texture it is important to recognize that texture depends on scale. The following is an extreme example: When looking at a street from far away it appears smooth (textureless). When zooming into the street it begins to look textured because a street in fact contains very small hills and crater-like structures. Zooming yet further into the street, the hills and craters become so large that they might appear smooth again.

Most existing algorithms can be categorized [29, 28] into the following groups: Local methods use a limited window in which they compare intensity values to find the best match. Global methods use global constraints (e.g. smoothness assumptions) and then try to minimize an energy function describing how well certain areas match in both images within the given global constraints. Some types of iterative methods are in between global and local methods and do not necessarily solve a global minimization problem. Finally, segment-based methods depend on stereo segmentation of the scene in order to achieve better results in untextured areas, but due
to the way most current image segmentation algorithms work this method typically has major difficulties in textured areas.

The algorithm proposed here is based on a segment-based method developed by Dellen et al. [10]. The major contribution of this work is the development of a solution that combines the advantages of the segment-based algorithm for untextured images with the advantages of an algorithm that runs well on textured images, resulting in a more universally usable method. As part of this thesis a new texture-supported segmentation algorithm had to be designed which also required several state-of-the-art image processing algorithms to be combined and including improvements to those algorithms.

Moreover, in order to achieve high performance in spite of the large number of interacting components a new compiler for imaging purposes, named py2gpu, has been implemented which is capable of compiling specifically annotated Python code to run highly parallelized on a GPU (Graphics Processing Unit), allowing to rapidly implement and experiment with high-performance algorithms using a high-productivity programming language.

The first part of this thesis describes the phase-based pre-segmentation disparity algorithm (chapter 2) which is used by the proposed method. Also, the first part explains the original segment-based method (chapter 3). The second part describes the proposed method. An overview of all components and a summary of the changes relative to the original segment-based disparity is given in chapter 4. The following chapter 5 describes the post-segmentation components in detail. In chapter 6 the pre-segmentation and segmentation components are explained in detail. The order of the pre-segmentation and post-segmentation chapters is reversed because the author believes that it is important to first understand the purpose and practical application of the segmentation results before going into the (pre-)segmentation details. Chapter 7 provides a description of the proposed method’s implementation with a focus on the py2gpu compiler. The results of the proposed method are compared against three other algorithms in chapter 8. Finally, the last two chapters provide a conclusion and ideas for future improvements.
Chapter 1

Stereo correspondence

1.1 Overview

The goal of stereo correspondence (or stereo vision) is to reconstruct a 3D model from a scene using two images taken from different perspectives. The underlying concept of stereo vision is that a point has different relative coordinates when seen from different perspectives. In figure 1.1 these coordinates are \( x_l, y_l \) in the left camera and \( x_r, y_r \) in the right camera.

![Diagram of stereo cameras and object coordinates]

Figure 1.1: Stereo cameras taking an image of an object (top) and the corresponding depth map (bottom)
In order to simplify the problem the eyes or cameras are positioned (either by nature or by humans) on the same axis, so that each pixel’s relative positions have the same \( y \)-component in both cameras: \( y_l = y_r \). Normally, the cameras should also have parallel alignment (see figure 1.2) and the images need to be rectified in order to compensate for small deviations from the perfect alignment. This reduces the process to finding an easily calculable disparity \( d = x_l - x_r \) with \( x_l > x_r \) for each point in the left image\(^1\) along the same horizontal line with \( y = y_l = y_r \). Using the disparity value \( d \), the focal length of the lens \( f \), the distance between the two cameras (i.e., the baseline) \( b \), and the physical size (in meters) \( p \) of each pixel on the CCD chip it is possible to calculate the actual distance of each point:

\[
D = \frac{bf}{pd} \tag{1.1.1}
\]

However, when comparing stereo correspondence algorithms one normally does not look at the distance, but rather at a so-called ground truth map which contains the true disparity values for a stereo pair. That is because the final result of a stereo correspondence algorithm is a 2D disparity map.

\(^1\)or the right image; this depends on which convention is used
1.1. OVERVIEW

$d(x, y)$ (see figure 1.1) consisting of disparity values mapping each pixel in the left image to its corresponding disparity value and thus to its position in the right image. The ground truth map is like a "true" or "reference" disparity map which contains the real disparity values which have been actually measured, for instance, with a laser. An example of a disparity map is shown in figure 1.1 where the intensity corresponds to the disparity value. Black means the disparity is 0, i.e. the pixel coordinates are the same in the left and right image and thus the pixel corresponds to a point that is very far away. White means "maximum disparity" which in an 8-bit gray-scale image is equivalent to $d = 255$ px, i.e. the pixel coordinates in the right image are shifted by 255 px along the x-axis, relative to the corresponding pixel’s coordinates in the left image. Such a pixel would belong to a point that is very close to the cameras. In general, with parallel camera alignment the disparity is inversely proportional to the distance of the corresponding point.

In practice, there are two variants of disparity maps: **Sparse disparity maps** contain disparity values for only a small subset of all pixels. In most cases this subset consists of values with high confidence. **Dense disparity maps** contain disparity values for all pixels (possibly with the exception of occlusions). Dense disparity maps also contain values with very low confidence. The reason for choosing a dense disparity map is that subsequent processing steps often require a complete disparity map and they can deal with disparity errors much better than with a complete lack of information in case of a sparse disparity map.

The difficulty of stereo correspondence algorithms lies in the matching process which tries to find the left pixel’s corresponding position in the right image. This is particularly difficult with reflective, translucent, and untextured surfaces. Depending on their pureness, reflective surfaces can look like a mixture of the reflected content and the reflective surface itself which results in conflicting hints about the point with highest correspondence. Since stereo correspondence algorithms normally have a winner-takes-all strategy they only accept one matching point. Translucent surfaces are very similar, however they can be even more complicated because, for instance, colored glass surfaces look not only a combination of the transmitted light and the colored surface, but additionally also contain reflections. Therefore, glass can have three correspondences! The third type of highly difficult surfaces are untextured (textureless) surfaces. These provide no hints or a very small number of hints against which a match can be made. Most algorithms require a lot of texture to produce high-quality results and perform poorly in untextured regions.
Stereo correspondence algorithms often consist of a subset of the following steps:

1. calculation of matching cost
2. aggregation of matching cost
3. calculation of disparity with possible optimization
4. refinement of disparity

Stereo algorithms can be categorized into four groups [29, 28]:

**Local methods** compare intensity values within a window. This process is applied in the neighborhood along the x-axis. One such algorithm is sum-of-squared-differences (SSD) [14, 21] for which the matching cost is the squared difference using the gray-scale intensity value \( I_l(x_l) \) of a pixel at the position \( x_l \) in the left image \( I_l \) (the right image is \( I_r \)) and a possible disparity \( s \):

\[
S_D = (I_l(x_l) - I_r(x_l - s))^2 \tag{1.1.2}
\]

In the aggregation step the sum of all squared differences within a square window with constant disparity is computed. The final disparity for each pixel is the \( s \) value which has the best (lowest) aggregate SSD value. Another common alternative is to use the sum-of-absolute-difference (SAD) [16] which defines the cost function as:

\[
S_A = |I_l(x_l) - I_r(x_l - s)| \tag{1.1.3}
\]

**Global methods** take global constraints like the overall smoothness into account and solve a global optimization problem for the whole image. Global methods often do not have an aggregation step, but instead calculate the disparity directly based on a global matching cost function and smoothness constraints. Many such methods are formulated as an energy minimization problem where the goal is to find the disparity \( d \) with minimum energy

\[
E(d) = E_d(d) + \lambda E_s(d) \tag{1.1.4}
\]

where the data term \( E_d(d) \) measures how well the disparity \( d \) agrees with the input image pair and \( \lambda \) is a term controlling the amount of smoothness in the optimal solution and the smoothness term \( E_s(d) \) represents the
1.2. EPIPOLAR GEOMETRY

smoothness assumptions of the algorithm. In order to not make the com-
putation of $E_s$ not too algorithmically time consuming the term is often
limited to neighboring pixels

$$E_s = \sum_{(x,y)} \rho (d(x,y) - d(x+1,y)) + \rho (d(x,y) - d(x,y+1)),$$ (1.1.5)

where $\rho$ is a monotonically increasing function of disparity difference.

Iterative methods are in between local and global methods. This class
of algorithms does not use a global function which needs to be minimized.
Iterative methods can be hierarchical (coarse-to-fine) algorithms which op-
erate on an image pyramid where results from coarse levels are used as
constraints for the computation at finer iteration levels. One such algo-
rumth is the phase-based disparity algorithm described in chapter 2.

Segment-based methods can also be considered in between local and
global methods. Here the stereo pair is segmented into regions and then
disparities are estimated individually for each segment. In a later step
the disparities of the segments are combined using certain smoothness con-
straints. Segment-based methods also allow to use surface fitting as a
smoothness constraint within segments in order to better deal with weakly
textured images and large segments. Compared to other methods, segment-
based algorithms are able to deal rather well with textureless images, but
segmentation algorithms can have difficulties when dealing with textured
areas.

The algorithm in this thesis is in fact a hybrid of all of the methods
mentioned here. It combines information from an iterative method, an
algorithm based on segmentation, a local method inside of segments, and
a global energy minimization problem in order to combine all intermediate
results.

1.2 Epipolar geometry

Epipolar geometry describes the projective geometry between two views.
Cameras always only take a projected image of the real world, i.e. 3D real-
world coordinates are projected to 2D camera coordinates. Since cameras
are never perfectly aligned/calibrated and the raw images are also not al-
ways free of distortions the assumptions made in the previous section do not
apply to the raw input images, so it is necessary to know the geometrical
basics of stereo vision.
In stereo vision the (geometrical) goal is to find the projected coordinates of a 3D point $p$ on the image plane (or retinal plane) of the right camera if the point’s projected coordinates on the image plane of the left camera are already known. Figure 1.1 shows such a situation where the point $X$ is projected on the left image plane at $x_L$ and the right image plane at $x_R$.

The left camera’s origin is at $O_L$ and the right camera’s origin is at $O_R$. The baseline connects both camera origins and its intersections $e_L$ and $e_R$ with the left and right image planes are the so-called epipoles. The $O_LX$, $O_RX$, and baseline projections form a so-called epipolar plane. As demonstrated in figure 1.1, a disparity algorithm only needs to search for projections from $O_R$ which intersect with the extended line connecting $O_L$ with $X$. These projections lie in the epipolar plane which intersects with the image plane on a so-called called epipolar line which of course is contained in the right image plane. A disparity algorithm operating on the right image would only have to search for a match on an epipolar line.

Since the calculation of the epipolar line makes the matching process unnecessarily difficult the input images are rectified. The details of rectification algorithms itself are out of the scope of these stereo correspondence methods, so they are not discussed here. The process of rectification changes the input images such that each image scanline (i.e., each horizontal line of pixels) equals an epipolar line. Basically, rectification projects the input images onto a common image plane, so they have a standard coordinate system. Rectification can also eliminate image distortions. Figure 1.2 shows the intermediate steps of a rectification algorithm [19]. In general, stereo correspondence algorithms assume rectified images as their input because they significantly simplify the matching process.
1.2. EPIPOLAR GEOMETRY

Figure 1.2: Rectification of two images [19]: (a) Original image pair overlaid with epipolar lines (b) Images transformed, so epipolar lines are parallel (c) Images transformed, so epipolar lines are horizontal and in vertical correspondence (d) Final result with minimized distortions
Chapter 2

Phase-based disparity

2.1 Overview

One of the components in this thesis is a phase-based disparity method which runs on the GPU and provides real-time processing speed [25, 24]. The phase-based disparity component is explained in this chapter and also serves as an example of how an iterative disparity method works. Moreover, compared to the original work, the phase-based disparity algorithm has been extended a little bit to automatically adjust a parameter based on the input image.

2.2 Algorithm description

There is a wide variety of approaches to phase-based disparity algorithms. The most notable distinction is made between local and global methods. Local methods focus on the intensity values in the close neighborhood around every pixel [20]. Global methods additionally use global constraints like a smoothness constraint over the whole image. While local methods are easy to implement they usually don’t deliver accurate results. In contrast, global methods yield very precise results, but they are unhandy because they require extensive tuning of parameters and iterative optimization.

As a compromise between these two methods, phase-based methods provide a precision level close to global methods while at the same time not requiring parameter tuning and iterative optimization. This property makes filter-based methods more practically useful than global and local methods.
CHAPTER 2. PHASE-BASED DISPARITY

Filter bank

![Gabor filter bank](image)

Figure 2.1: Gabor filter bank used for extracting phase at eight orientations. Even filters are in the top row. Odd filters are in the bottom row. [25]

The algorithms used in the phase-based method critically rely on the Gabor phase to detect features and establish pixel correspondences. This follows the model of cortical simple cells [8, 26]. The spatial phase for a pixel at the location \( x = (x, y)^T \) for a specific orientation \( \theta \) and the peak frequency \( f_\theta = (f_{x, \theta}, f_{y, \theta})^T \) can be extracted using 2D complex Gabor filters:

\[
G(x, f_\theta) = e^{-|x|^2/\sigma^2} e^{i\pi(x f_\theta)} .
\]  

[22] proposes a highly efficient spatial-domain implementation of a Gabor filter bank.

The filters used in that paper are tuned to a very high frequency of \( |f_\theta| = \frac{1}{4} \text{px}^{-1} \). Also, separability [15] and symmetry considerations were exploited to obtain responses at four orientations by utilizing 12 1D convolutions with 11-tap filters. The algorithm introduced in [25, 24] extends that method to obtain responses at eight orientations, using 24 1D convolutions with 11-tap filters. Figure 2.1 shows the Gabor filter bank used in this algorithm.

Gaussian pyramid

Since the phase is periodic, phase-based techniques are only capable of detecting shifts up to half the filter wavelength. In order to extend this range a coarse-to-fine control strategy can be used [11]. An efficient solution involves the use of a Gaussian pyramid [2].

In order to accommodate that the peak frequency \( f_\theta \) is doubled from one scale to the next the filters span an octave bandwidth:

\[
B = \frac{|f_\theta|}{3} .
\]  

(2.2.2)
2.2. ALGORITHM DESCRIPTION

With a cutoff frequency of half the amplitude spectrum the spatial extension is equal to:

\[ \sigma = \sqrt{\frac{2 \ln(2)}{B}} \]

(2.2.3)

At the highest frequency we have

\[ |f_0| = \frac{\pi}{2} \text{px}^{-1} \]  

(2.2.4)

which results in a spatial extension of \( \sigma = 2.25 \text{px} \). The lower frequency responses are obtained by applying the Gabor filters (2.2.1) to an image pyramid which is constructed by repeatedly blurring the images with a Gaussian kernel \( g(x) \) and subsampling. Given the image \( I(x) \) and the subsampling operator \( S \) which reduces the image resolution to half the resolution of the previous level and starting from the original image resolution at pyramid level \( k = 1 \) the next level \( k + 1 \) is calculated via

\[ I^{k+1}(x) = \left( S \left( g * I^k \right) \right)(x) \]  

(2.2.5)

Now, the original filters (2.2.1) have to be applied to each level of the Gaussian pyramid. At the first level \( k = 1 \) the filter response is obtained by convolving the image \( I(x) \) with the oriented filter from (2.2.1). Using the convolution operator \( * \) we obtain the response

\[ R(x) = (I * G)(x) \]  

(2.2.6)

\[ = \rho(x) e^{i\phi(x)} \]  

\[ = C(x) + iS(x) \]  

In this equation

\[ \rho(x) = \sqrt{C(x)^2 + S(x)^2} \]  

(2.2.7)

and

\[ \phi(x) = \arctan \left( \frac{S(x)}{C(x)} \right) \]  

(2.2.8)

denote the amplitude and phase components. \( C(x) \) and \( S(x) \) are the responses of the quadrature filter pair. Similar to (2.2.6), at higher levels the filter response is obtained using

\[ R^k(x) = (I^k * G)(x) \]  

(2.2.9)
By applying the original filters to the lower resolution images the largest detectable shift effectively doubles at each pyramid level. The spatial filter kernels are $11 \times 11$ pixels in size and separable.

While the original paper uses six scales, in this thesis the number of scales $n_s$ is variable and depends upon the maximum allowed disparity $d_{\text{max}}$ in a particular image:

$$n_s = \lceil \ln(d_{\text{max}}) \rceil \quad (2.2.10)$$

This modification allows to use the phase-based disparity method on larger images and improves the quality of the results when using small images or images with smaller maximum disparity because a smaller area is analyzed for correspondence in both images, thus reducing the number of errors.

**Binocular disparity**

While the phase-based disparity algorithm is primarily intended for computing binocular optical flow it also supports a special mode for computing binocular disparity. That mode is used in this thesis in order to calculate the initial disparity map.

Like most disparity algorithms, the phase-based algorithm requires rectified images. Binocular disparity is calculated using the phase difference between the left and right images. A stereo disparity estimate can then be obtained from each oriented filter response (at orientation $\theta$) by projecting the phase difference on the epipolar line (the horizontal). In this way, multiple disparity estimates are obtained at each location.

With the $\frac{\pi}{2 \pi}$operator which depicts reduction to the $[-\pi; \pi]$ interval the estimates are then combined using the median over $\theta$:

$$\delta(x) = \mu_{1/2} \left( \frac{\phi^L_\theta(x) - \phi^R_\theta(x)}{\omega_0 \cos \theta} \right)_{2\pi} \quad (2.2.11)$$

The coarse-to-fine control scheme described in the previous section is used to calculate estimates over all pyramid levels.

Optionally, unreliable estimates can be detected and removed by running the algorithm from left to right and from right to left and looking for mutual consistency. In this thesis unreliable estimates are used, instead. The results of the later processing steps are negatively affected if large parts of the disparity map are missing.

In particular, the Metropolis segmentation algorithm [1] is capable of correcting mistakes in the disparity map during its relaxation phase, but
it has major problems with correcting mistakes due to missing disparity information which practically is the same as having a large error in the disparity map. Thus, unreliable disparity estimates are a compromise leading to better overall results. Moreover this way the number of iterations can be reduced which reduces the overall processing time.
Chapter 3

Segment-based disparity

In this chapter, a segment-based stereo algorithm [10], on which the proposed method is based, is described. This algorithm yields good results for untextured or weakly textured images, but fails with textured images due to the limitations of image segmentation which represents a critical component of this algorithm. In contrast, local and global methods work well in textured areas, but fail in weakly textured areas because the image lacks sufficient information to reliably estimate correspondence between both images. This chapter describes the original stereo algorithm developed by Dellen et al. [10]. The next chapters describe the extensions and modifications required for the texture-supported segment-based disparity algorithm.

The core idea of this segment-based method is that it is possible to establish segment correspondences for weakly textured images. The segments can then be processed individually. The advantage is that the segments conveniently limit the range of possible disparity values for pixels within the segment, thus reducing the potential for matching errors. Also, the disparities of pixels on segment edges, which often coincide with real edges in the image, can be estimated very precisely.

3.1 Algorithm overview

The algorithm consists of several components and processing steps (see figure 3.1). The input data consists of a stereo image pair (step 0). This pair is passed to an initial disparity algorithm (step 1). The original paper uses the algorithms from the Middlebury MRF library\(^1\) which is based on the works of [4, 5, 17, 30].

\(^1\)http://vision.middlebury.edu/MRF/code/
The resulting initial disparity map is then passed to a stereo segmentation algorithm developed by Dellen et al. [9] (step 2). Alternatively, this step can be replaced with other stereo segmentation algorithms such as [27, 31]. Acceptable stereo segmentation algorithms, which are compatible with this disparity algorithm, yield a linked pair of segment maps where each segment in the left image has the same label as its corresponding segment in the right image. More details about this algorithm can be found in the original paper [9].

In the next step (step 3) potentially occluded areas are estimated from the segments. This also takes the estimated depth ordering of the segments into account which is based on the segment’s center-of-mass shift between the left and right image. See section 3.2 for a more detailed description.

The segment edge disparities (step 4) are estimated for each segment by computing the segment silhouette and finding correspondences between both image. The algorithm considers segment edge disparities as the most reliable data. See section 3.3 for more details.

The inner-segment disparities (step 5) are computed using a window-based matching algorithm which strictly operates inside stereo segments which eliminates most errors caused by occlusions. Confidence values are estimated for every disparity value and only disparities with a high confidence value are used, so the result of this step is normally a sparse disparity map. See section 3.4 for a more detailed description.

Finally, the sparse disparity data from the edge disparity and inner-segment disparity estimates is interpolated (step 6) using a spring-mass

Figure 3.1: Block diagram of the algorithm. [10]
model incorporating region constraints. Every pixel of the image is represented by a mass point. Mass points belonging to the same segment are connected to their nearest neighbors via elastic springs. Mass points belonging to different segments are not connected. The amplitude of the mass point denotes its disparity value. The edge and inner-segment disparity values are used to define forces which attract the mass points into their respective precomputed disparity values from step 4 and 5. However, segment-edge mass points do not get a force if they are occluded according to the occlusion map calculated in step 3. A damping force is used to drive the system into a local minimum. A more detailed description of the interpolation algorithm can be found in section 3.5.

3.2 Potentially occluded areas

In step 3 of figure 3.1 the potentially occluded areas are computed based on the estimated ordering of the segments. The estimated disparity is based on the center-of-mass of the segment. Given a pixel $i$ and the set of pixels $S_{i,\text{left}}$ and $S_{i,\text{right}}$ in the left and right image that share the same label as $i$, the estimated center-of-mass disparity for said pixel is:

$$d_{cm,i} = \frac{\sum_{u \in S_{i,\text{left}}} x_u}{\sum_{u \in S_{i,\text{left}}} 1} - \frac{\sum_{u \in S_{i,\text{right}}} x_u}{\sum_{u \in S_{i,\text{right}}} 1}$$  \hspace{1cm} (3.2.1)

From this the occlusion map is determined using:

$$O_i = \theta \left( \sum_{j \in N_i} \theta \left( d_{cm,j} - d_{cm,i} - \tau_3 \right) \right)$$  \hspace{1cm} (3.2.2)

Here, $N_i$ is the rectangular $5 \times 5$ pixels neighborhood around pixel $i$ and $\tau_3 = 5$ is a threshold for the minimum disparity difference between two pixels. Also, $\theta$ is a step function with the following properties:

$$\theta \left( n \right) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{if } n \leq 0 \end{cases}$$  \hspace{1cm} (3.2.3)

Note that in contrast to the Heaviside step function $\Theta \left( n \right)$ this function yields 1 only when $n$ is greater than zero, but not when $n$ equals zero.
3.3 Segment-edge disparities

In step 4 of figure 3.1 the segments’ edge (boundary) points are determined and the disparities of these edge points are estimated by comparing each segment in the left image with its corresponding segment in the right image. A segment is only taken into account if all its edge points are visible in both the left and right image, i.e., if the segment does not ”move” out of the image borders on the left or right. A pixel \(i\) with segment label \(s_i\) and position \(x_i\) is considered to be a left segment edge pixel \(i_l\) if there is no pixel \(u\) with
\[
x_u < x_i \text{ and } s_u = s_i.
\]
(3.3.1)
A pixel \(i\) with segment label \(s_i\) is considered to be a right segment edge pixel \(i_r\) if there is no pixel \(u\) with
\[
x_u > x_i \text{ and } s_u = s_i.
\]
(3.3.2)

Given the edge pixels \(i_l\) and \(i_r\) for the left image and the edge pixels \(j_l\) and \(j_r\) for the right image along the x-axis line, the edge disparity is then computed as
\[
d^e_i = (i_l - j_l) \delta_{i_l,i} + (i_r - j_r) \delta_{i_r,i},
\]
(3.3.3)
with the amplitude
\[
a^e_i = \delta_{i_l,i} + \delta_{i_r,i}.
\]
(3.3.4)
The amplitude denotes the confidence of the respective disparity value.

3.4 Inner-segment disparities

In step 5 of figure 3.1 the disparities of weakly textured areas within the segments are computed using a region-constrained window-based matching algorithm based on correlation coefficients. A rectangular window \(R_i\), normally of \(11 \times 11\) pixels in size, around each pixel \(i\) of the left image \(I_L\) is constructed and the correlation coefficient with each rectangular window \(R_j\), of equal size as \(R_i\), around each pixel \(j\) of the right image \(I_R\) along the x-axis is computed via
\[
c_{i,j} = \frac{a_{i,j}}{b_i b_j},
\]
(3.4.1)
where
3.5. INTERPOLATED DISPARITIES

\[ a_{i,j} = \sum_{u \in R_j} \sum_{v \in R_i} (I_R(x_u, y_u) - m_j) (I_L(x_v, y_v) - m_i) \delta_{s_u,s_i} \delta_{s_v,s_i} \]

\[ b_i = \sum_{v \in R_i} (I_L(x_v, y_v) - m_i) \delta_{s_v,s_i} \]

\[ b_j = \sum_{u \in R_j} (I_R(x_u, y_u) - m_j) \delta_{s_u,s_i} \]

Here, \(s_i, s_u,\) and \(s_v\) are the segment labels of the respective pixels \(i, u,\) and \(v.\) Also,

\[ m_i = \sum_{v \in R_i} I(x_v, y_v) \delta_{s_v,s_i} / \sum_{v \in R_i} \delta_{s_v,s_i} \]

\[ m_j = \sum_{u \in R_j} I(x_u, y_u) \delta_{s_u,s_i} / \sum_{u \in R_j} \delta_{s_u,s_i} \]

For each pixel \(i\) the pixel \(j_{\text{max}}\) is found for which the correlation coefficient \(c_{i,j}\) is maximal:

\[ j_{\text{max}} = \arg \left[ \max_j (c_{i,j}) \right] \quad (3.4.2) \]

With this the disparity of the pixel \(i\) is defined as:

\[ d^C_i = x_i - x_{j_{\text{max}}} \quad (3.4.3) \]

The respective amplitude \(a^C_i\) is given by \(c_{i,j_{\text{max}}}:\)

\[ a^C_i = \theta \left( \left( \sum_{u \in R_{j_{\text{max}}}} \sum_{v \in R_i} \delta_{s_v,s_i} \delta_{s_u,s_i} \right) - \tau_1 \right) \theta (c_{i,j_{\text{max}}} - \tau_2) \quad (3.4.4) \]

where \(\tau_1\) and \(\tau_2\) are thresholds and \(\theta\) is the step function defined in (3.2.3). Normally, \(\tau_1 = 40\) and \(\tau_2 = 0.92.\)

3.5 Interpolated disparities

In the last phase (step 6 of figure 3.1) the intermediate results obtained in the previous steps are combined and these disparities get interpolated which results in a dense disparity map. Each pixel \(i\) in the left image is assigned a mass \(m_i\) and all masses within a segment are connected by
springs with the spring constant $k$ (see also: figure 3.1). Each mass has a "position" $x_i$ and "velocity" $v_i = dx_i/dt$, where the position corresponds to the instantaneous disparity value of respective mass’s pixel, scaled by a factor $f_s = 5/d_{\max}$ which depends on $d = 1.5 \max (d^e_i)$. The resulting system is described by the following equations:

$$\frac{dx_i}{dt} = v_i$$
$$\frac{dv_i}{dt} = -\gamma v_i + F_{\text{edges}} + F_{\text{inner}} + F_{\text{int}}$$

The forces $F_{\text{edges}}$ and $F_{\text{inner}}$ in that equation drive the system towards the edge disparities and inner disparities, respectively. The force $F_{\text{int}}$ represents the interaction between neighboring pixels within a segment. These forces are described as follows:

$$F_{\text{edges}} = k^e a_i^e (f_s d_i^e - x_i)$$
$$F_{\text{inner}} = g^c a_i^c (f_s d_i^c - x_i)$$
$$F_{\text{int}} = k \sum_{l \in N_i} (x_l - x_i) \delta_{s_i,s_l}$$

The sum in $F_{\text{int}}$ goes over all four nearest neighbors $N_i$ of mass $i$ which belong to the same segment. The constant $\gamma = 0.5$ determines the amount of damping in the spring-mass system. The other constants determine the
"weight" that a certain force gets during interpolation. They are set to $k^c = 0.25$, $k^e = 5$, and $k = 10$. The system is initialized with random values. In the next step it gets iterated using a 4th-order Runge-Kutta technique with 1200 iteration steps and a step size of $1/6$ time units. In order to remove outliers from the inner disparity data $g^c$ is defined as:

$$g^c = \begin{cases} k^c (1 - |x_i - d^c_i|) & \text{if } g^c > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$= \max (k^c (1 - |x_i - d^c_i|), 0)$$
Chapter 4

Algorithm components

4.1 Overview

The result of this diploma thesis is a segment-based disparity method incorporating texture information, from now on referred to as texture-supported segment-based disparity algorithm. In this chapter, the algorithm is introduced. An in-depth description of the various processing steps is given in the following chapters.

The texture-supported segment-based disparity algorithm is based on the segment-based disparity algorithm explained in chapter 3 and [10]. Conventional disparity algorithms work well on textured images, but fail on untextured images because textured images provide a lot of hints for stereo matching. In contrast, untextured images (or image areas) provide an insufficient amount of such hints which leads to bad or unusable results. The segment-based disparity algorithm has the opposite properties. It works well with untextured images, but fails to find correspondences in textured images. This thesis combines the segment-based approach with a more traditional approach in order to get the best of both worlds.

The general idea behind segment-based disparity is that a stereo segmentation algorithm can produce sufficiently reliable results for segment edges in untextured images. Disparity values for these edges can be calculated very easily since each segment’s pixels can be identified by their segment label which is the same in both images of the stereo pair. Additionally, segments can be used as effective constraints when finding correspondences for non-edge pixels. While local stereo correspondence algorithms have difficulties matching large areas the segment-based algorithm limits the matching process to pixels that belong to the same stereo segment pair. This way the analyzed image area is kept small enough such
CHAPTER 4. ALGORITHM COMPONENTS

that correct correspondences can be found even in untextured areas. The primary reason why the segment-based disparity algorithm performs poorly on textured images is that the utilized co-segmentation algorithm does not produce useful results in textured image areas.

In order to overcome this limitation a new texture-based segmentation algorithm was developed as part of this thesis. However, while this segmentation algorithm alone already improves the disparity results, this thesis goes further by also incorporating disparity values from a traditional stereo correspondence method. The motivation behind this is that by merging two stereo correspondence algorithms with very distinct properties it should be possible to achieve good results in both textured and untextured images at the same time.

Another issue with the original disparity algorithm is its slow performance. The original algorithm has processing times of almost 30 minutes for a 400 × 400 px image. The goal of this thesis is to achieve processing times of less than a minute in order to facilitate faster experimentation with different ideas. The components of the original algorithm are shown in figure 3.1. The components of the texture-supported algorithm are shown in figure 4.1. Compared to the original segment-based disparity algorithm this algorithm adds several components for improving results on textured images. The added components are:

- Texture detection
- Fine smoothing
- Coarse Smoothing
- Blurring
- Average edge disparity

In addition to the added components, all of the original components got replaced with improved versions. The affected components and their improvements are:

- Pre-segmentation disparity: performance-optimized
- (Co-)segmentation: texture support, performance-optimized
- Occlusion map: general improvements
- Edge disparity: general improvements
- Inner disparity: general improvements, performance-optimized
4.2 PROCESSING STEPS

• Interpolation: texture support, performance-optimized

With the exception of "Blurring" and "Average edge disparity", all newly added components are also performance-optimized. All of the performance-optimized processing steps use parallelized algorithms which get executed on the GPU. The optimized components are also marked with a little "GPU" symbol in figure 4.1. With the exception of "Pre-segmentation disparity" and "Co-segmentation" all of the "GPU" components are implemented using a special Python-to-GPU compiler which was also developed as part of this thesis. The compiler allows for writing high-performance code at a high abstraction level, again facilitating fast experimentation. The compiler is described in more detail in section 7.3. In general, the implementation aspects of the algorithm as a whole are discussed in chapter 7.

4.2 Processing steps

A visualization of each processing step is shown in figure 4.1. The input data of this algorithm is a rectified stereo image pair (see step 1 in figure 4.1).

The goal of this thesis is to extend the original segment-based disparity algorithm (see chapter 3 and [10]) such that it not only works well on untextured images, but also on textured images. Thus, the "Texture detection" step (see step 2 in figure 4.1) plays a central role in this thesis. In particular, it is the foundation of the texture-based segmentation algorithm and it has an important influence on the "Interpolation" step which combines the intermediate disparity results of the components in this algorithm. The texture detection method is based on an algorithm designed by HP Labs [3] which, unlike many other algorithms, is capable of detecting actual texture without false-positives on edges/discontinuities, i.e. edges are not treated as texture. This step will be explained in more detail in section 6.2.

After the texture detection the image is smoothed in order to remove texture from the image while retaining the color values. This step is necessary for the segmentation to yield proper segments in textured areas. At first, fine smoothing (see step 3 "Fine smoothing") is applied to the whole image in order to reduce overall noise in the image which helps the segmentation algorithm with finding pixels of similar color value. Next, stronger smoothing (see step 4 "Coarse smoothing") is applied to textured areas, as determined by the previous texture detection method. This is the actual step that removes strong texture from the image. The "Fine smooth-
ing” and “Coarse smoothing” steps are based on the mean-shift algorithm which is a discontinuity-preserving smoothing method. The “discontinuity-preserving” property of that algorithm is the key to being able to process the image in the later segmentation step without causing segments to get merged due to smoothed/removed edges. Both smoothing steps are discussed in chapter 6.

Another prerequisite of the stereo segmentation algorithm is an initial disparity map which is used to connect segments in the left image with segments in the right image. This initial disparity map is calculated in the ”Pre-segmentation disparity” step (see step 5) which is based on the phase-based disparity algorithm discussed in chapter 2. The algorithm was chosen because of its good results at a very high performance suitable for real-time video applications.

Before the smoothed image pair is passed to the segmentation algorithm it is blurred slightly (see step 6 ”Blurring”) via simple Gaussian smoothing in order to widen the edges in the image and thus simplify the process of resolving the image into individual segments. This step is only required for the particular segmentation method used in this thesis and should normally not be used in combination with other segmentation methods.

Another central component of this algorithm is the ”Co-segmentation” (see step 7), also referred to as the ”Segmentation” component. As was demonstrated by the original segment-based disparity algorithm, segments work fairly well for estimating disparities of untextured image areas. Unfortunately, segmentation does not yield good results in textured image areas. One of the contributions of this thesis is an extended segmentation method which can form segments not only based on color similarity, but also based on the existence (i.e., textured or untextured) and color of texture. In order to achieve this, the segmentation method combines the components discussed above (pre-segmentation disparity, fine and coarse smoothing, blurring, texture detection) into a new kind of segmentation algorithm. The segmentation itself is based on a modified Metropolis algorithm which was chosen due to its real-time processing performance. The original Metropolis algorithm is described in [1]. The modifications to that algorithm are explained in chapter 6 with the most important extension being a ”texture” image component which is explained in section 6.5.

In the next step (step 8) the ”Edge disparity” is determined for each segment by calculating the segment boundary points in the left and right image and finding correspondences between the edge pixels in both images. As with the original method, segment edge disparities are considered to be the most reliable data in this algorithm. As an improvement over the
original method, segment edge disparities are also estimated for segments that partially reach out of the image boundaries. This step is explained in section 5.1.

The edge disparity is then used to calculate the "Average edge disparity" (see step 9) for each segment by simply taking all edge disparity values of the segment, calculating the mean value, and assigning that value to the whole segment. This component is discussed in section 5.2.

The average edge disparity is required for the following "Occlusion map" stage (see step 10) in which potentially occluded areas are estimated. The calculation takes the estimated depth ordering into account which is based on the average edge disparity. The original segment-based disparity algorithm used the center-of-mass, but that proved to be not sufficiently reliable. This part of the algorithm is described in section 5.3.

In the "Inner disparity" stage (see step 11) the inner-segment disparities are computed. For this purpose a window-based matching technique is used which strictly operates inside stereo segments. This helps eliminate most errors caused by occlusions. It also imposes useful restrictions on the area which gets compared and thus helps reduce matching errors. The inner disparity algorithm used in this thesis has been parallelized and slightly improved, so it does not yield too large disparity values - in particular ones which would be outside of the visible image area. The "Inner disparity" component is discussed in section 5.4.

The final "Interpolation" stage (see step 12) combines the intermediate disparity results of the previous steps using a spring-mass model incorporating region constraints. Here, the edge disparity, inner disparity, and pre-segmentation disparity are represented as forces which act on mass points representing the pixels in the image. Mass points belonging to the same segment are connected to their nearest neighbors via elastic springs. In contrast, mass points which belong to different segments are not connected via springs. The amplitude of the mass point denotes its disparity value. As with the original algorithm, forces are used to drive the mass points into the determined edge and inner-segment disparities in the previous steps. These forces are only added for unoccluded pixels, as determined in the "Occlusion map" step. Additionally, the new algorithm has been improved for textured scenes by incorporating a third external force which drives the mass points to the pre-segmentation disparity value at the respective pixel's location. That force is only added in textured regions. This modification requires a pre-segmentation disparity algorithm that works well in textured areas, but not necessarily in untextured areas. Finally, a damping force is used to drive the system into a local minimum. The interpolation
algorithm has also been parallelized for execution on the GPU, resulting in a massive speed-up. A more detailed explanation of the algorithm is given in section 5.5.
Figure 4.1: Block diagram of the proposed algorithm. New components are in green. Modified components are in blue. The corresponding intermediate results are shown in figure 4.1.
Figure 4.1: Results of each processing step; right image data is not shown: (a) Left input image (b) Texture detection (c) Smoothed and blurred (d) Pre-segmentation disparity (e) Segmentation of left image (f) Segmentation of right image. Continued in figure 4.2.
4.2. PROCESSING STEPS

Figure 4.2: Continued results of each processing step: (a) Edge disparities (b) Edge disparity average (c) Occlusion map (d) Inner disparities (e) Final disparity / interpolation
Chapter 5

Post-segmentation components

In this chapter all components which come after the segmentation (steps 8-12 in figure 4.1) are explained in detail. The pre-segmentation and segmentation components are explained separately in the next chapters except for the "pre-segmentation disparity" component (step 5 in figure 4.1) which is already described in chapter 2.

5.1 Edge disparity

In step 8 of figure 4.1 the disparities of each segment’s edge pixels are estimated. This is done by comparing each segment in the left image with its corresponding segment in the right image.

A pixel $i$ with segment label $s_i$ and position $x_i$ is considered to be a left segment edge pixel $i_l$ if there is no pixel $u$ with $x_u < x_i \land s_u = s_i$. \hspace{1cm} (5.1.1)

A pixel $i$ with segment label $s_i$ is considered to be a right segment edge pixel $i_r$ if there is no pixel $u$ with $x_u > x_i \land s_u = s_i$. \hspace{1cm} (5.1.2)

If the segment reaches out of the left image boundaries the left segment edge pixel’s ($i_l$) position is $x_i = 0$. Similarly, if the segment reaches out of the right image boundaries the right segment edge pixel’s ($i_r$) position is $x_i = W - 1$ where $W$ is the image width. A left boundary pixel with position $x_i$ in the left image and the corresponding left boundary pixel with position $x_j$ in the right image are only taken into account if

$$1 < x_i < W - 2 \land 1 < x_j < W - 2.$$ \hspace{1cm} (5.1.3)

33
The right boundary pixels \( r_i \) and \( r_j \) are handled with a slightly different rule which takes potentially occluded background segments into account by ignoring segments for which the left edge is not visible:

\[
1 < x_i < W - 2 \land 1 < r_i < W - 2 \land 1 < r_j < W - 2
\]  
(5.1.4)

By applying both rules independently on the left edge pixel pair and the right edge pixel pair, partially visible segments are still taken into account. The rules (5.1.3) and (5.1.4) also ignore left edge pixels that touch the right image boundary and the right edge pixels that touch the left image boundary. This limitation is critical because it is common that one pixel wide segments appear in the edge areas of the image pairs and their (incorrect) disparities would "destroy" the disparity map around the image boundaries.

In a final processing step, pixels with an edge disparity that occurs less than \( \tau_e = 6 \) times in the whole edge disparity map, are removed from the edge disparity map. The purpose of this processing step is to remove outliers and allow computing a realistic minimum edge disparity value for the "Inner disparity" processing step in section 5.4.

Given the edge pixels \( i_l \) and \( i_r \) (left and right edge, respectively) for the left image and the edge pixels \( j_l \) and \( j_r \) for the right image along the x-axis line, the edge disparity is then computed as

\[
d^e_i = (i_l - j_l) \delta_{i_l,i} + (i_r - j_r) \delta_{i_r,i}
\]  
(5.1.5)

Here, \( i \) must be set to \( i_l \) and \( i_r \) for the left and right edge disparity, respectively, but it is only set to \( i_l \) if (5.1.3) is true for the left edge pixel pair and it is set to \( i_r \) only if (5.1.3) is true for the right edge pixel pair. The corresponding amplitude is defined as

\[
a^e_i = \delta_{i_l,i} + \delta_{i_r,i}
\]  
(5.1.6)

The edge disparities play a very important role in this algorithm because they belong to the most reliable data available. The quality of the segmentation algorithm obviously has the greatest influence on the edge disparities. Here, "quality" is not measured in terms of how well the segments match the way humans perceive a scene, but rather how reliably segment pixels are connected in the left and right image. A higher number of small segments leads to more disparity hints in the image. This basically corresponds to having a more textured image with traditional disparity algorithms.

Unlike the original implementation in section 3.3 segments that partially reach out of the image are taken into account, too. In other words, if
some edge pixels are not visible in one of the images the disparities of the other (visible) pixels belonging to the respective segment are still calculated. This modification adds important disparity information around the image borders.

### 5.2 Average edge disparity

In step 9 of figure 4.1 the average edge disparity is calculated for each segment. All pixels belonging to a segment are assigned the mean value of the corresponding segment’s edge disparities. Given the set of edge disparities \( D_s \) of a segment \( s \), all pixels of that segment are assigned the following value

\[
d_s = \frac{1}{N} \sum_{d \in D_s} d,
\]

where \( N \) is the number of pixels in \( D_s \). The average edge disparity is needed for the calculation of the occlusion map as explained in the next section.

### 5.3 Occlusion map

The occlusion map is calculated in step 10 of figure 4.1. The purpose of the occlusion map is to find pixels that should not be marked as connected in the final interpolation step. Occluded pixels are, by definition, only visible in one image, thus it is impossible to find a stereo correlation for these pixels. The occlusion map marks such pixels and excludes them from the final disparity map. If the calculated occlusions are larger than the actual occlusions disparity information gets lost unnecessarily. In contrast, if the calculated occlusions are smaller than the actual occlusions the final disparity map will contain incorrect disparity values which will negatively affect a limited area around the occlusion. Obviously, occlusions should be as correct as possible.

The segment occlusion \( O_{s,i} \) of each pixel \( i \) caused by the segment with label \( s \) is determined using:

\[
O_{s,i} = (1 - \delta_{s_i,s}) \cdot \theta \left( \sum_{j \in N_i} \delta_{s_j,s} \cdot \theta \left( d_{s_j} - d_{s_i} - \tau_3 \right) \right)
\]

(5.3.1)

Here, \( N_i \) is the 5 pixels neighborhood along the x-axis to the right of pixel \( i \), \( d_s \) is defined in (5.2.1), \( s_i \) is the segment label of pixel \( i \), \( \tau_3 = 5 \) is a
CHAPTER 5. POST-SEGMENTATION COMPONENTS

Figure 5.1: Center of mass shifting negatively when objects occlude each other

threshold for the minimum disparity difference between two pixels, and \( \theta \) is the same step function as (3.2.3).

Finally, the occlusion \( O_i \) of each pixel \( i \) is defined as the combination of all occlusions

\[
O_i = \theta \left( \sum_{s \in S} O_{s,i} \right),
\]

(5.3.2)

with \( S \) being the set of all segment labels in the left image.

The occlusion detection algorithm has changed significantly in this proposed method when compared to the original algorithm in section 3.2 which is based on the center of mass. The modifications are based on the observation that the center of mass is incorrect and sometimes even shifts in the opposite direction when segments are occluded. This effect is illustrated in figure 5.1. In the worst case this can lead to negative disparities which of course cannot exist in reality. The center of mass gets influenced by overlapping segments far too easily and thus is too error-prone. A solution that works acceptably well is to replace center of mass with the average edge disparity values described in the previous section.

Another improvement over the original algorithm is that the size of the detected occlusions is significantly reduced. The reduction is illustrated in figure 5.2 where the gray area represents the segment, the black area is analyzed area of new algorithm, and the red and black areas together denote the analyzed area of the original algorithm which marks everything within a box around the segment as potentially occluded. It should be emphasized here that not the whole ”analyzed area” is actually considered occluded. Only those parts of that area which belong to segments with a lower average
5.4. INNER DISPARITIES

In step 11 of figure 4.1 the disparities of weakly textured areas within the segments are computed using a region-constrained window-based matching algorithm based on correlation coefficients.

A rectangular window $R_i$ of $(2\beta + 1) \times (2\beta + 1)$ pixels (in this thesis: $\beta = 5$) around each pixel $i$ of the left image $I_L$ is constructed and the correlation coefficient with each rectangular window $R_j$, of the same size as $R_i$, around each pixel $j$ of the right image $I_R$ along the x-axis is computed as

$$c_{i,j} = \frac{a_{i,j}}{b_i b_j},$$

(5.4.1)

with

Figure 5.2: Occluded areas of a segment (gray): black: new algorithm, red+black: previous algorithm

edge disparity than the currently selected segment are actually marked as occluded. Still, due to the significantly decreased size of the analyzed area the errors in the occlusion map are much smaller with the new algorithm. As can be seen in figure 5.2, occlusions are now only analyzed around the left part of the segment silhouette. Pixels to the right of segment silhouette are never considered occluded because this case is impossible, anyway. Also, when looking at the bottom of figure 5.2 it becomes apparent that the new algorithm now detects inner edges of the segment, so "holes" in the segment are treated optimally, too.
\begin{align*}
a_{i,j} &= \sum_{u \in R_j} \sum_{v \in R_i} (I_R(x_u, y_u) - m_j) (I_L(x_v, y_v) - m_i) \delta_{s_u,s_i} \delta_{s_v,s_i} \\
b_i &= \sum_{v \in R_i} (I_L(x_v, y_v) - m_i) \delta_{s_v,s_i} \\
b_j &= \sum_{u \in R_j} (I_R(x_u, y_u) - m_j) \delta_{s_u,s_i},
\end{align*}

where \(s_i, s_u,\) and \(s_v\) denote the segment labels of the respective pixels \(i, u,\) and \(v.\) Also:

\begin{align*}
m_i &= \sum_{v \in R_i} I(x_v, y_v) \delta_{s_v,s_i} / \sum_{v \in R_i} \delta_{s_v,s_i} \\
m_j &= \sum_{u \in R_j} I(x_u, y_u) \delta_{s_u,s_i} / \sum_{u \in R_j} \delta_{s_u,s_i},
\end{align*}

For each pixel \(i\) the pixel \(j_{i,\max}\) is found for which the correlation coefficient \(c_{i,j}\) is maximal:

\[j_{i,\max} = \arg \max_{j \in J_i} (c_{i,j})\]  \hfill (5.4.2)

The set of pixels \(j\) for which \(j_{i,\max}\) is search is defined as

\[J_i = \{j\mid \forall j \in I_R : y_i = y_j \land 0 < x_i - x_j < d_{i,\max} \land \beta < x_j\}\]  \hfill (5.4.3)

where

\[d_{i,\max} = \begin{cases} 
d^e_{\min} & \text{if } T_{p,i} \geq \tau_{d,\min} \land (x_{l,i} \leq 1 \lor x_{r,i} \geq W - 2) \\
\max & \text{otherwise} 
\end{cases}\]  \hfill (5.4.4)

with \(W\) being the image width and \(x_{l,i}\) and \(x_{r,i}\) being the x-coordinates of the left- and right-most pixels of the segment belonging to pixel \(i.\) Also

\[d_{\max} = \min \{1.4d^e_{\max}, 0.4W\}\]  \hfill (5.4.5)

with \(d^e_{\max}\) being the maximum edge disparity

\[d^e_{\max} = \max \{d_i^e | i \in I_L\}\]  \hfill (5.4.6)
and $T_{p,i}$ being the texture value of pixel $i$ and $\tau_{d,\text{min}}$ being a texture threshold (here: $\tau_{d,\text{min}} = 0.17$) and $d^e_{\text{min}}$ being the minimum edge disparity

$$d^e_{\text{min}} = \min \{ d^e_i | i \in I_L \} \quad (5.4.7)$$

Now, the disparity of the pixel $i$ is defined as

$$d^e_i = x_i - x_{j_i,\text{max}} \quad , \quad (5.4.8)$$

and the corresponding amplitude $a^e_i$ is given by $c_{i,j_i,\text{max}}$

$$a^e_i = \theta \left( \left( \sum_{u \in R_{j_i,\text{max}}} \sum_{v \in R_i} \delta_{s_u,s_i} \delta_{s_v,s_i} \right) - \tau_1 \right) \theta \left( c_{i,j_i,\text{max}} - \tau_2 \right) \quad , \quad (5.4.9)$$

where $\tau_1$ and $\tau_2$ are thresholds and $\theta$ is the step function defined in (3.2.3). Experimentally, $\tau_1 = 40$ and $\tau_2 = 0.92$.

The original algorithm in section 3.4 has been improved in this thesis by limiting the disparity range in which correlation coefficients are compared. This eliminates negative disparities and pixels that might have a higher correlation coefficient, but which are too far apart (i.e. which have a too large disparity value).

### 5.5 Interpolation

In step 12 of figure 3.1, which is also the final calculation, the intermediate results obtained in the previous steps are combined and interpolated. The result of this step is a dense disparity map. Each pixel $i$ in the left image is assigned a mass $m_i$ and all masses within a segment are connected by springs with the spring constant $k$ (see figure 5.1). Each mass has a "position" $x_i$ and "velocity" $v_i = dx_i/dt$, where the position corresponds to the instantaneous disparity value of respective mass’s pixel. The resulting system is described by the following equations:

$$\frac{dx_i}{dt} = v_i$$

$$\frac{dv_i}{dt} = -\gamma v_i + F_{\text{edges}} + F_{\text{inner}} + F_{\text{int}} + F_{\text{pre}}$$

The forces $F_{\text{edges}}$ and $F_{\text{inner}}$ in that equation drive the system towards the edge disparities and inner disparities, respectively. These forces are described as follows:
Chapter 5. Post-segmentation Components

Figure 5.1: Spring-mass model of new algorithm incorporating pre-segmentation disparity for textured areas.

\[ F_{\text{edges}} = k^e a_i^e (d_i^e - x_i) \]
\[ F_{\text{inner}} = g^e a_i^e (d_i^e - x_i) \]

Here, \( k^e \) is a system parameter. In order to remove outliers from the inner disparity data \( g^e \) is defined as

\[ g^e = \max \left\{ k^e \left( \frac{d_{\text{max}}}{15} - |x_i - d_i^e| \right), 0 \right\} \tag{5.5.1} \]

where \( d_{\text{max}} \) is defined as in (5.4.5) and \( k^e \) is a system parameter.

The force \( F_{\text{pre}} \) with the system parameter \( k^s \) additionally drives the system towards the pre-segmentation disparity calculated in step 5 of figure 4.1:

\[ F_{\text{pre}} = k^s T_i (d_i^s - x_i) \delta_{d_i^e, 0} \tag{5.5.2} \]

This force does not exist in the original algorithm described in section 3.5. It is only applied to pixels that are neither edge pixels nor pixels belonging to an untextured area. In the definition above, \( T_i \) is the texture indicator of pixel \( i \). Its value is 0 for pixels belonging to untextured areas and 1 for pixels belonging to textured areas. The reasoning behind adding this force is that the pre-segmentation disparity algorithm is considered to provide...
5.5. **INTERPOLATION**

high-quality results in textured areas whereas the segment-based disparity algorithm has huge problems with texture. Thus, the pre-segmentation disparity is incorporated into the interpolation and given a higher weight than the inner disparities: $k^e \gg k^s \gg g^c$.

The force $F_{\text{int}}$ represents the interaction between neighboring pixels within a segment:

$$F_{\text{int}} = k \sum_{l \in N_i} (x_l - x_i) \delta_{s_i,s_l}$$

(5.5.3)

The sum in $F_{\text{int}}$ goes over all four nearest neighbors $N_i$ of mass $i$ which belong to the same segment. The constant $\gamma = 0.5$ determines the amount of damping in the spring-mass system. The other constants determine the "weight" that a certain force gets during interpolation. They are set to $k^c = 0.25$, $k^e = 8$, $k^s = 1$, and $k = 10$. The system is initialized with random values. In the next step the system is iterated using a parallelized 4th-order Runge-Kutta algorithm with 1200 iteration steps and a step size of 1/6 time units. The result of this step is the final dense disparity map of the texture-supported segment-based disparity algorithm.
Chapter 6

Texture-supported segmentation

6.1 Overview

When talking about texture it is important to recognize that texture depends on scale. At different zoom levels, the same image area can appear textured or untextured. Moreover, depending on the application, texture can mean different things. Sometimes only a repeating pattern is considered texture. The pattern does not even need to be fine-grained. Other times any strong variance of color intensities is considered texture. This is why texture is hard to define. It highly depends on the application. In this case, texture is considered to be an extended two-dimensional area of strong, fine-grained color variance. This means that single edges in an otherwise low-variance area are not considered texture. It is difficult to provide an exact and universal mathematical definition, but section 6.2 provides a definition in terms of a specific texture detection algorithm that matches texture according to the given requirements.

The co-segmentation algorithm used in the segment-based disparity algorithm and the GPU-optimized co-segmentation algorithm used in the proposed method only work best in untextured areas. In contrast, textured areas in the image either receive no segment labels or they are not linked correctly in the stereo pair. This is because textured areas have a higher color variance which to the segmentation algorithm looks like an area full of small discontinuities and thus full of small segment boundaries. In case of segmentation algorithms based on paramagnetic clustering this color variance translates to a region with strong forces which in turn prevent the formation of regions with shared spin. Consequently, important
information is lost which could otherwise provide useful clues for the disparity algorithm. Since this algorithm’s weakest link is the segmentation it is crucial to improve the quality of the segmentation.

In order to achieve this goal it is necessary to modify the segmentation algorithm such that it can form segments even in textured image areas. However, an important aspect of this goal is that color should still be taken into account, so that adjacent textured areas of different color can be resolved into separate segments.

The modification of the segmentation algorithm consists of three components:

1. The first component is a texture detection algorithm which ignores individual discontinuities within untextured areas (see section 6.2).

2. The second component is an algorithm that can smooth texture without destroying discontinuities, so the segmentation algorithm only needs to process an untextured image (see section 6.4). Note that if the smoothing algorithm were not discontinuity preserving the segmentation algorithm would not be able to form clean segment boundaries.

3. The final component is an extended segmentation algorithm (see section 6.5). This segmentation method’s input data is enriched with an additional component which represents the existence of texture for each pixel in the image (based on the results from the first component). Thus, the resulting input is a four-layer 3D matrix with HSV\textsuperscript{1}+texture (or RGB\textsuperscript{2}+texture) components.

The biggest problem is to find texture detection and smoothing algorithms that do not treat discontinuities as texture which must be detected/smoothed. Algorithms with adequate properties for the segmentation method are described in the next two sections and the following section explains the segmentation algorithm’s modifications in more detail.

### 6.2 Texture detection

The texture detection method is needed for the smoothing, segmentation, and interpolation steps of the texture-supported segment-based disparity algorithm. Thus, it is of great importance that the texture detection
6.2. TEXTURE DETECTION

Algorithm 6.1 Component count descriptor

1. Calculate average gray level in the current block.

2. Separate the pixels within the block into two groups: those with gray level above average and those with gray level below or equal to average.

3. The background group is defined as the larger one of the two groups. The smaller group is the foreground group.

4. Convert the block into a binary image where background pixels are set to 0 and the foreground pixels are set to 1.

5. Segment the image by assigning unique labels to 4-connected components in the binary image.

6. Count the number of segments.

7. Invert the binary image and repeat steps 5 and 6.

8. Return the sum of both segment counts.

method is reliable and produces good results. Many texture detection methods have problems with filtering out discontinuities. They mistakenly recognize edges or single lines in an otherwise untextured area as strong texture. The problem with this is that the image is smoothed and texture-segmented in a later step, but in this case discontinuities in untextured areas should not be smoothed because the segmentation algorithm needs strong discontinuities in order to form clean segments. Another reason why discontinuities should not be treated as texture is that the pre-segmentation disparity algorithm’s results (see chapter 2) are integrated in textured areas during the interpolation step (see section 5.5), but this is only desired in image areas that are in fact textured and not mistakenly recognized to be so. Also, the segmentation algorithm would treat mistakenly recognized discontinuities as segments, but due to their small size they would not be stable and thus constantly interfere with the actual segment boundary.

In 2007 HP Labs published a paper about a texture detection method which uses a “disorganization indicator” based on “component counts” [3].

Since texture is dependent on scale, the algorithm analyzes the image using different block sizes. The authors use blocks of 5 × 5, 10 × 10, and 25 × 25 pixels and the same configuration is used in this thesis. The first
part of the algorithm is the component count descriptor which is calculated as described in algorithm 6.1.

A few examples of the component count are shown in figure 6.1 which lists samples from the original paper. As can be seen, the textured areas in the images respond with a high component count whereas the untextured areas have a low component count.

Obviously, the component count alone is not sufficient to describe texture because low-contrast blocks can still have a lot of components. Thus, in addition to the component count the contrast of the block is calculated by taking the absolute difference between the average gray level of the fore-
6.2. TEXTURE DETECTION

ground and background groups. When combined, the component count and the contrast are able to describe the texture value of an image. According to the authors, this method also provides some robustness in the face of noise.

In order to combine the component count and contrast descriptors their values have to be converted to probabilities in the range \([0, 1]\). The original paper suggests different methods to do this. In the implementation used in this thesis the conversion of the component count \(n\) and the contrast \(c\) is calculated via:

\[
\begin{align*}
  n_{\text{conv}} &= \max \left( \min \left( \sqrt[2]{\frac{\log (n)}{2}}, 1 \right), 0 \right) \\
  c_{\text{conv}} &= \max \left( \min \left( \sqrt[2]{\frac{\log (c)}{2}}, 1 \right), 0 \right)
\end{align*}
\] (6.2.1)

(6.2.2)

Let \(n_5, n_{10}, n_{25}\) and \(c_5, c_{10}, c_{25}\) be the converted component counts and contrasts for block sizes of \(5 \times 5\), \(10 \times 10\), and \(25 \times 25\), respectively. The texture likelihood is calculated with the weights \(\omega_5, \omega_{10}, \omega_{25}\) where \(\omega_5 + \omega_{10} + \omega_{25} = 1\) for every pixel:

\[
p = \omega_5 n_5 c_5 + \omega_{10} n_{10} c_{10} + \omega_{25} n_{25} c_{25}
\] (6.2.3)

Here, \(\omega_5 = 0.2\), \(\omega_{10} = 0.55\), and \(\omega_{25} = 0.25\). Finally, the texture indicator for every pixel is defined as

\[
T = \begin{cases} 
1 & \text{if } p > \tau \\
0 & \text{otherwise}
\end{cases}
\] (6.2.4)

where \(\tau\) is the texture threshold which depends on the conversion method for the component count and contrast. In this thesis a threshold of \(\tau = 0.62\) is used.

Note that the implementation of the texture detection algorithm used in this thesis is a modification of the original algorithm described in the original paper [3]. When looking at a sample result from the original paper (figure 6.2a) it becomes apparent that the component count is very coarse-grained which is problematic during segmentation because it results in a large position error of \(\sigma = \pm 12.5\)px (with a maximum window size of \(25 \times 25\)px) which affects the segment boundaries. This translates into very imprecise disparity values. The algorithm has been modified in this thesis to provide a higher positional resolution as can be seen in figure 6.2b. The difference between the original algorithm and the modified variant in this
thesis is that in this thesis the component count and contrast descriptors are applied using overlapping blocks. The original algorithm would place the first $5 \times 5$ block at $(0, 0)$ and the second block at $(0, 5)$ etc. The modified algorithm places the first block at $(0, 0)$ and the second block at $(0, 1)$ etc. Thus, each pixel of the image takes part in 25 calculations in the $5 \times 5$ block scenario. The resulting texture component of each pixel is the mean value of all blocks containing that pixel. Note that the border area (4px for $5 \times 5$ blocks)) would be touched fewer times, but in order to guarantee a high positional resolution for the whole image a border of 24px (the maximum block size is $25 \times 25$) around the whole image is simply removed from all calculation steps. This means that the $x$ and $y$ dimensions of the final disparity results in this thesis are 48px smaller when compared to most other disparity algorithms. This is a relatively minor disadvantage, but it must be taken into account when using scaled-down images.

### 6.3 Binary segment counting

Steps 5 and 6 in the texture detection algorithm’s sub-algorithm 6.1 depend on a binary segmentation algorithm. As the texture detection algorithm runs parallelized on the GPU there is a very important restriction which has to be taken into account by the segmentation algorithm: Memory may
not be allocated at runtime.

This practically prohibits the reuse of existing open-source binary segmentation libraries because they depend on complex data structures which are allocated on-demand. Additionally, in this special case the texture detection algorithm only needs to count the number of segments. It is not necessary to generate a complete segment map which means the algorithm can take shortcuts which reduce the number of memory operations and thus improve the runtime performance. Such a binary segmentation algorithm which runs on the GPU and only counts the number of segments has been designed and implemented in this thesis.

This binary segmentation algorithm uses a segment matrix $S$ of the same width $w$ and height $h$ as the binary input image $B$ to store temporary labels. Additionally, a variable-length vector $M$ with a maximum length of $\lceil hw/2 \rceil$ is used to map the temporary labels in $S$ to their actual labels. The additional map $M$ increases the memory requirements of the algorithm by 50% and it would not be necessary if only the segment map were of interest. However, since the texture detection algorithm only depends on the number of segments it is possible to trade memory for runtime efficiency.

The binary image $B$ is scanned from left to right, top to bottom. In each step the current pixel $i$ is compared against the top neighbor pixel $t_i$ and the left neighbor pixel $l_i$. Let $S_p$ be the temporary label of pixel $p$ and let $M_{S_p}$ be its remapped label. Let $B_p \in \{0, 1\}$ be the binary value of a pixel $p$. If a pixel $u$ is undefined, i.e. outside of the image boundaries, its value is defined as $B_u = 0$. Thus, if there is no top neighbor (this only happens in the first row) $B_{ti} = 0$ and if there is no left neighbor (only happens in the first column) $B_{li} = 0$.

With the simplification $R_p = M_{S_p}$ and $\dim M$ being the length of $M$, i.e. the number of labels in $M$, the temporary labels are defined as

$$S_i = \begin{cases} 
0 & \text{if } B_i = 0 \\
R_{li} & \text{if } B_i = 1 \land B_{li} = 1 \land (B_{ti} = 0 \lor R_{li} < R_{ti}) \\
R_{ti} & \text{if } B_i = 1 \land B_{ti} = 1 \land (B_{li} = 0 \lor R_{li} \geq R_{ti}) \\
1 + \dim M & \text{if } B_i = 1 \land B_{li} = 0 \land B_{ti} = 0
\end{cases} \quad (6.3.1)$$

where after each iteration step the label map $M$ is updated as follows:

\begin{align*}
\forall k \in [1, \dim M] : & M_k := R_{li} \text{ if } M_k = R_{li} \land B_i = B_{li} = B_{ti} = 1 \land R_{li} < R_{ti} \\
\forall k \in [1, \dim M] : & M_k := R_{ti} \text{ if } M_k = R_{ti} \land B_i = B_{ti} = B_{li} = 1 \land R_{ti} > R_{ti} \\
M_{S_i} := & 1 + \dim M \text{ if } B_i = 1 \land B_{li} = 0 \land B_{ti} = 0
\end{align*}
The first two rules replace a label with another label. According to the last term in both rules the new label is always smaller than the original label. In other words, when two regions are joined the smaller label wins.

The third rule causes the vector $M$ to grow by one element such that at the end of the iteration $M_{\dim M} = \dim M$. In general, if $M_i = i$ is true it means that the label $i$ is not remapped. The number of segments equals the number of non-remapped segment labels:

$$n_M = \sum_{i=1}^{\dim M} \delta_{i,M_i} \quad (6.3.2)$$

### 6.4 Smoothing

The mean-shift algorithm [6, 34, 7, 13] allows for discontinuity-preserving image smoothing. This makes it a perfect candidate as a pre-segmentation smoothing method.

The algorithm tries to iteratively find a representative color for the every pixel position $p = (y, x)$. In each iteration the mean color $\langle c \rangle_p$ and mean position $\langle p \rangle_p$ is calculated for a color sphere of radius $r_c$ around the current color $c$ and a spatial sphere of radius $r_p$ around the current position $p$. Only pixels whose positions and colors lie within those spheres are taken into account. Let $n_p$ be the number of pixels within the respective spheres around the position $p$ and let $T$ be the set of position-color tuples $(p_i, c_i)$ in the image. Then the mean values for each iteration can be obtained from

$$\langle c \rangle_p = \frac{1}{n_p} \sum_{(p_i, c_i) \in T} f_c(p_i), \quad (6.4.1)$$

and

$$\langle p \rangle_p = \frac{1}{n_p} \sum_{(p_i, c_i) \in T} f_p(p_i), \quad (6.4.2)$$

where

$$f_c(p_i) = \begin{cases} c_i & \|p - p_i\| \leq r_p \\ 0 & \text{otherwise} \end{cases} \quad (6.4.3)$$

and

$$f_p(p_i) = \begin{cases} p_i & \|p - p_i\| \leq r_p \\ 0 & \text{otherwise} \end{cases} \quad (6.4.4)$$
The algorithm is initialized with the current pixel’s color value $c$. After each iteration the current color value $c$ is updated to be $\langle c \rangle_p$. In this particular implementation of the algorithm the iteration stops either after having been repeated 100 times or when the difference between the current position and color reaches a limit $L$:

$$\|p - \langle p \rangle_p\| + \|c - \langle c \rangle_p\| \leq L \quad (6.4.5)$$

After the last iteration the resulting smoothed color value is set to the current color value $c$.

At this point it should be mentioned that in other implementations of this algorithm the position and color space are not necessarily viewed as different features with distinct spheres around the current position and color. Also, the current position is not necessarily fixed, but can move towards the mean position.

As mentioned at the beginning of this subsection the major advantage of the mean-shift algorithm is its edge-preserving nature. This also leads to a second advantage: This method can be used to safely remove very fine-grained texture from the image. While image segmentation is normally able to deal with fine-grained texture it is possible to improve the segmentation results by reducing the noise in the image because the segmentation method used in this disparity algorithm computes a pixel similarity factor which is used to determine whether two pixels (and consequently two spins) should be connected. More noise in the image leads to a lower similarity factor which in turn leads to weaker connections. Without the fine-grained texture noise the segmentation method becomes less sensitive to small color changes and is able to detect real color boundaries more reliably.

### 6.5 Texture component for segmentation

#### Problem description

The last part of the texture-supported segmentation algorithm consists of an extension of the original segmentation method. The modified segmentation algorithm processes not only the colors red, green, and blue, but also an additional texture component. In the original algorithm [1] the color distance is calculated via

$$\Delta c_{ij} = \|g_i - g_j\| \quad , \quad (6.5.1)$$
where $g_i$ and $g_j$ are the color vectors for pixels $i$ and $j$ respectively. Given the color distance $\Delta c_{ij}$ and the texture values $t_i$ and $t_j$ the overall value distance between two pixels is now calculated by

$$\Delta v_{ij} = \sqrt{\Delta c_{ij}^2 + (t_i - t_j)^2}.$$  \hspace{1cm} (6.5.2)

Instead of the color distance the modified segmentation algorithm uses the value distance to determine the similarity between two pixels. Now the question is which texture representation for $t_i$ and $t_j$ and which color representation provide the best results for the disparity algorithm. This is the topic of the next section.

**Texture and color components**

At this point it is important to note that the segmentation results of this extended algorithm might not be perceived as better by humans because the segments look arbitrary and do not map directly to perceived objects in the original image. However, in the case of segment-based disparity algorithms the segmentation quality is determined by the stability of the segments in the stereo pair and the number of segments. An image with a large number of seemingly arbitrary and small segments can return better results than an image which mostly consists of big segments. As long as the segment edges can be reliably connected between both images of the stereo pair the disparity algorithm can provide reliable results. Due to the complex interactions involved between the various sub-algorithms, it is practically impossible to find a quality measure for segmentation methods in the context of segment-based disparity algorithms. Thus, the quality has to be measured at a more abstract level by comparing the final disparity results. Unfortunately, this makes it difficult to optimize the segmentation algorithm in a systematic way. In other words, it is necessary to experiment with unintuitive texture representations for the segmentation algorithm.

The initial approach was to simply segment the smoothed color components in combination with the raw texture level. This solution did not lead to acceptable results because as such texture and color represent conflicting information, so the color image needs to be smoothed in order to eliminate these conflicts.

The final solution, which is now used in this thesis, uses two smoothing steps and a constant texture component. At first the whole image is smoothed using the mean-shift algorithm described above. In this step only very light smoothing is used with $r_c = 25$, $r_p = 3$, and $L = 3$. As mentioned in section 6.4, this increases the average pixel similarity in the image...
6.5. TEXTURE COMPONENT FOR SEGMENTATION

Figure 6.1: Smoothing results: (a) Original image (b) Fine smoothing (c) Coarse smoothing

and in turn helps to keep more segments separated. Next, an additional coarse (strong) smoothing step is introduced in which only textured areas ($T = 1$) are smoothed using mean-shift with $r_c = 65$, $r_p = 6$, and $L = 15$. The coarse smoothing step eliminates texture from the image. An example image and its corresponding fine and coarse smoothing results are shown in figure 6.1.

The resulting smoothed image is passed as the color component to the segmentation algorithm. Through experimentation it was determined that the texture component works well when set to $\tau_c = 40$ where $T = 1$ and $\tau_c = 0$ where $T = 0$. This solves the noise problem in the texture indicator as explained above and provides reliable results. Nevertheless, possible future improvements of this segmentation method are discussed in chapter 10.
Chapter 7
Implementation

7.1 Overview

A significant part of this work consists of the actual implementation of the various algorithmic components. A proof of this algorithm’s practical usefulness and an analysis of its problems can only be provided by actually building and testing a running system. The algorithm itself is written in Python using the numpy/scipy libraries, except for the phase-based disparity (chapter 2) and segmentation [1] algorithms which have not been developed as part of this thesis.

Python is a very expressive dynamically typed language. Its productivity advantage allows for rapid implementation and iteration of ideas. The numpy/scipy Python libraries\(^1\) provide a set of scientific functions for working with vectors, matrices, and images. Internally, they are implemented in C, so they have good overall performance characteristics. These libraries behave like normal Python packages, so dynamic typing and other Python advantages are preserved.

However, due to the number of computationally intensive methods used, the numpy/scipy libraries hit their performance limits in some components which became major bottlenecks. In order to keep the processing time at an acceptable level these components had to be adapted to run parallelized on the GPU using NVIDIA’s CUDA architecture\(^2\). For this purpose a Python-to-CUDA compiler (\texttt{py2gpu}) and a compiler generator framework (\texttt{PyMeta2}) have been implemented as part of this thesis. This chapter focuses on the py2gpu compiler because of its important role in the implementation.

\(^1\)http://www.scipy.org/
\(^2\)http://www.nvidia.com/object/what_is_cuda_new.html
7.2 Parallelization

Due to py2gpu’s approach to abstracting parallelism, the implementation of the individual algorithm components is very straightforward, so there is nothing to gain from a detailed explanation of each component’s implementation. The general principle behind parallelizing the components in this thesis is to split the data matrices into $n \times m$ sized windows. Each window is assigned one parallel thread which handles the calculation for its own window independently of all other threads. A schematic showing pixel-sized windows can be seen in figure 7.1.

The only exception is the interpolation component (see section 5.5) because it depends on an iterative and thus inherently serial 4th-order Runge-Kutta (from now on abbreviated as RK4) algorithm. Moreover,
each iteration step of the spring-mass model depends on its four-neighbors’ values at the same iteration step. Hence, it is impossible to calculate each $n \times m$ window independently of all other threads. All threads must be synchronized before each iteration step. In other words, before the next iteration starts all threads must have finished the previous iteration step. Even with a parallelized linear equation solver this synchronization problem would still exist, so instead of parallelizing the RK4 algorithm the approach in this thesis is to parallelize each iteration step of the spring-mass model (also see figure 7.2): The next iteration value of all the mass points is calculated in parallel on the GPU (with one thread per mass point) and the result is written into a separate output matrix, so the input data is not overwritten while some threads still depend on their four-neighbors’ values. Then, all threads are synchronized and the output matrix of the previous step becomes the input matrix of the next step. This solution is rather simple, but it provides a decent performance improvement, not only compared to the Python implementation, but also compared to a hand-optimized version of the interpolation component written in C for the CPU.\footnote{Several components of this disparity algorithm were temporarily rewritten in C/Cython, but the py2gpu compiler showed more promising results, so the final implementation is based on py2gpu.}

7.3 Python->GPU compiler (py2gpu)

Introduction

The py2gpu compiler is a simple translator which converts Python code to CUDA C++ code. By transforming Python to C++ instead of low-level assembly it is possible to reuse a lot of the complex code optimizations already implemented in the C++ compiler and focus on high-level implementation aspects.

Since Python is a dynamically typed language and C++ is statically typed the Python code has to be annotated by hand with type information. In theory it would be possible to emulate dynamic typing using C++ templates, but such an implementation requires a type inferencer which is a complex piece of software that is by far out of the scope of this thesis.

The idea behind the compiler is that a lot of image processing algorithms are window-based (with single-pixel operations being just a special case). The py2gpu compiler is specialized on such window-based algorithms and it automatically manages the window positioning based on the currently
executing thread’s identifier and provides a multi-dimensional coordinate system relative to the current window, as shown in figure 7.1 where an image is split in four windows with four threads running concurrently on those windows and each thread’s window having a coordinate system that starts at (0,0). Additionally, the py2gpu compiler has automatic memory management. In comparison, hand-written CUDA code is very complicated because it requires manual management of window positions and coordinates and it only supports flat one-dimensional arrays, so multi-dimensional coordinates have to be calculated by hand and transformed in terms of the current window position. Moreover, hand-written CUDA code requires the developer to manually allocate and free memory which leads to more bugs which are hard to debug.

Due to its properties the py2gpu compiler allows rapidly implementing and iterating ideas and running them with very high performance. At its current stage the compiler does not produce optimal code, but it already runs two orders of magnitude faster than the original code and with additional optimizations it should be possible to achieve another major speed-up. The big advantage of this approach is that all existing code will automatically benefit from improvements to the compiler and in theory it would be possible to get very close to the performance of hand-optimized code while at the same time achieving very high developer productivity through Python.

The following subsections explain the most important parts of the compiler implementation and introduce py2gpu’s end-user API.
7.3. PYTHON->GPU COMPILER (PY2GPU)

The whole processing pipeline which converts the Python source to a .cu source file and finally compiles it for the GPU is shown in figure 7.2.

As the very first step of this compiler, a parser has to convert the Python input code into an abstract syntax tree (AST). For this purpose Python already provides a library called "ast", so instead of reinventing the wheel the py2gpu compiler simply uses that library to parse Python code and retrieve the corresponding AST.

The next step is to analyze the AST and emit C++ code. This transformation phase can be enhanced with additional high-level optimizations before generating C++ code. In the case of the py2gpu compiler the main optimization consists of automatically detecting an optimal number of parallel threads for executing a GPU function.

In order to traverse and analyze the AST the PyMeta2 [18] pattern-matching language was created as part of this thesis. PyMeta2 is a Python implementation of the OMeta language version 2 [32, 33], with a few parts reused from the original PyMeta implementation which is based on the OMeta language version 1. OMeta and thus PyMeta2 is based on a variant of Parsing Expression Grammars (PEGs) [12]. It can be used to express parsing and transformation rules and thus provides a very flexible framework for building programming languages. Its main flexibility stems from its ability to pattern-match arbitrary objects with a single pattern language which makes it far superior than parser generators like Flex/Bison, Lex/Yacc, and ANTLR which can only operate on linear character streams and which require separate languages for implementing lexers, tokenizers, and parse tree visitors.

Table 7.1 lists all pattern matching expressions supported by PyMeta2. Algorithm 7.1 shows an example of how to parse simple arithmetic expressions and emit an AST with PyMeta2.

Since PyMeta2 can directly operate on arbitrary object streams it is well suited to handle Python’s AST. For example, the following code snippet extracted from the compiler handles parsing of binary arithmetic operators:

---

4CUDA also has a 1D-3D texture memory, but this is not suitable for general-purpose applications because it is read-only

5https://launchpad.net/pymeta
### Table 7.1: Definition of the language of parsing expressions (*e*, *e*₁, and *e*₂ are parsing expressions, and *r* is a non-terminal)

<table>
<thead>
<tr>
<th>expression</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>e₁ e₂</em></td>
<td>sequencing</td>
</tr>
<tr>
<td>*e₁</td>
<td>e₂*</td>
</tr>
<tr>
<td><em>e</em>⁺</td>
<td>zero or more repetitions</td>
</tr>
<tr>
<td><em>e</em>⁻</td>
<td>one or more repetitions</td>
</tr>
<tr>
<td>(<em>e</em>)</td>
<td>grouping</td>
</tr>
<tr>
<td>~<em>e</em></td>
<td>negation</td>
</tr>
<tr>
<td>~~~<em>e</em></td>
<td>look-ahead</td>
</tr>
<tr>
<td><em>r</em></td>
<td>rule application</td>
</tr>
<tr>
<td><em>e</em>:x</td>
<td>stores the result of <em>e</em> in <em>x</em></td>
</tr>
<tr>
<td><em>(code)</em></td>
<td>matches if <em>code</em> evaluates to <em>True</em></td>
</tr>
<tr>
<td><em>(code)</em></td>
<td>executes a piece of <em>code</em></td>
</tr>
<tr>
<td>'x'</td>
<td>matches character &quot;*x&quot;</td>
</tr>
<tr>
<td>&quot;hello&quot;</td>
<td>matches character sequence &quot;hello&quot;</td>
</tr>
<tr>
<td>[<em>e₁ e₂</em>]</td>
<td>matches a list with two elements</td>
</tr>
</tbody>
</table>

Algorithm 7.1 Parser for simple arithmetic expressions. Based on [32]

```python
dig = '0' | ... | '9'
num = dig+:ds -> ['num', int(''.join(ds))]
fac = fac:x '*' num:y -> ['mul', x, y]
    | fac:x '/' num:y -> ['div', x, y]
    | num
exp = exp:x '+' fac:y -> ['add', x, y]
    | exp:x '-' fac:y -> ['sub', x, y]
    | fac
grammar = exp
node :name = :n ?(n.__class__.__name__ == name) -> n
binop = node('BinOp'):n -> '(%s %s %s)' % (self.parse(n.left, 'op'), self.parse(n.op, 'anybinaryop'), self.parse(n.right, 'op'))
anybinaryop = add | div | mult | sub
add = node('Add') -> '+'
div = node('Div') -> '/'
```
mult = node('Mul') -> '*'
sub = node('Sub') -> '-'

The node rule is used to match a certain node type. The parse() method is used to walk down the tree and parse a sub-node. The AST nodes for addition, division, multiplication, and subtraction are converted directly to their respective C operators. As can be seen, PyMeta2 allows to specify parsing rules in a very elegant and compact way.

Usage

Due to their technical nature, the usage examples can be found in appendix A. The appendix lists several features not mentioned here and it shows the level of abstraction achieved by py2gpu more effectively than the overview in this section.

Conclusion (py2gpu)

This purpose of this section is to provide a short introduction to the principles behind the py2gpu compiler in order to give an impression of the abstraction level achieved by the compiler. Instead of managing threads, window positions, coordinate system, and memory allocation by hand, the end-user can focus on the actual algorithm and write bug-free code much more easily. The abstraction works very well in practice, at least in the context of the many diverse components implemented in this thesis. Other algorithms might need support for additional features like map/reduce operations which can be used to efficiently reduce a very large matrix to a single resulting number (e.g., the sum of all matrix elements). As shown by the pycuda library\(^6\), such a feature is certainly possible to implement with a nice abstraction API and it would be useful extension for the future. Additionally, this project could be combined with NVIDIA’s Copperhead compiler\(^7\) which is also a Python-to-GPU compiler. The difference between py2gpu and Copperhead is that the former is specialized on image processing whereas the latter is specialized on element-wise mapping and map/reduce operations and, unlike py2gpu, Copperhead contains a type inferencer which completely eliminates the need to annotate GPU functions with static type information\(^8\).

\(^6\)http://documen.tician.de/pxcuda/
\(^7\)http://code.google.com/p/copperhead/
\(^8\)this works by specializing the untyped code on-the-fly when a function is called, utilizing the type information in the passed arguments at runtime
Chapter 8

Results

8.1 Quality metrics

In this chapter three main metrics are used. All three metrics require a ground truth disparity map.

The root-mean-square (RMS) error between the computed disparity map $d_C(x,y)$ and the ground truth map $d_T(x,y)$ is obtained via

$$ R = \sqrt{\frac{1}{N} \sum_{(x,y)} \left| d_C(x,y) - d_T(x,y) \right|^2}, \quad (8.1.1) $$

where $N$ is the number of pixels in the image.

The percentage of bad pixels, or the cumulative error, can be computed using:

$$ B(\Delta) = \frac{1}{N} \sum_{(x,y)} \theta \left( |d_C(x,y) - d_T(x,y)| - \Delta \right) \quad (8.1.2) $$

where $\theta$ is defined in (3.2.3) and $\Delta$ is a disparity error threshold. In this thesis the cumulative error is plotted graphically as a function of $\Delta$.

The third metric is an error histogram constructed using

$$ H(\Delta) = \frac{1}{N} \sum_{(x,y)} \delta_{\Delta, |d_C(x,y) - d_T(x,y)| + 0.5} \quad (8.1.3) $$

and plotted as a function of $\Delta$.

A fourth metric, which primarily makes sense in the context of the proposed and the original segment-based algorithm, is the percentage $M$ of undefined/missing pixels. Both algorithms provide information about the
Table 8.1: Comparison of root-mean-square error, measured in units of disparity value

<table>
<thead>
<tr>
<th></th>
<th>Texture-supported</th>
<th>Original</th>
<th>Phase-based</th>
<th>Graph-cuts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baby</td>
<td>2.6 ± 0.2</td>
<td>6.4 ± 5.2</td>
<td>7.2 ± 0.0</td>
<td>7.5 ± 0.0</td>
</tr>
<tr>
<td>Cones</td>
<td>2.3 ± 0.3</td>
<td>2.6 ± 0.4</td>
<td>4.4 ± 0.0</td>
<td>5.5 ± 0.0</td>
</tr>
<tr>
<td>Lampshade</td>
<td>5.2 ± 0.2</td>
<td>14.0 ± 0.3</td>
<td>18.0 ± 0.0</td>
<td>5.0 ± 0.0</td>
</tr>
<tr>
<td>Midd</td>
<td>4.5 ± 0.3</td>
<td>20.3 ± 4.2</td>
<td>18.3 ± 0.0</td>
<td>9.7 ± 0.0</td>
</tr>
<tr>
<td>Monopoly</td>
<td>6.7 ± 3.5</td>
<td>27.5 ± 4.4</td>
<td>17.6 ± 0.0</td>
<td>28.4 ± 0.0</td>
</tr>
<tr>
<td>Plastic</td>
<td>7.4 ± 3.7</td>
<td>4.7 ± 0.9</td>
<td>12.5 ± 0.0</td>
<td>11.7 ± 0.0</td>
</tr>
<tr>
<td>Rocks</td>
<td>2.6 ± 0.3</td>
<td>4.2 ± 0.9</td>
<td>5.3 ± 0.0</td>
<td>17.0 ± 0.0</td>
</tr>
<tr>
<td>Teddy</td>
<td>2.4 ± 0.1</td>
<td>6.0 ± 4.3</td>
<td>4.9 ± 0.0</td>
<td>3.7 ± 0.0</td>
</tr>
<tr>
<td>Tsukuba</td>
<td>1.8 ± 1.1</td>
<td>2.5 ± 2.1</td>
<td>2.4 ± 0.0</td>
<td>6.7 ± 0.0</td>
</tr>
<tr>
<td>Venus</td>
<td>1.1 ± 0.1</td>
<td>2.3 ± 2.5</td>
<td>2.7 ± 0.0</td>
<td>3.9 ± 0.0</td>
</tr>
<tr>
<td>Wood</td>
<td>8.3 ± 1.9</td>
<td>4.4 ± 0.6</td>
<td>11.3 ± 0.0</td>
<td>7.5 ± 0.0</td>
</tr>
<tr>
<td>Average</td>
<td>4.1 ± 2.5</td>
<td>8.6 ± 8.4</td>
<td>9.5 ± 6.3</td>
<td>9.7 ± 7.3</td>
</tr>
</tbody>
</table>

8.2 Quantitative comparison

The eleven images used for the quantitative quality comparison are listed in figure 8.1 and figure 8.2. Currently, the Middlebury stereo evaluation website\(^1\) compares only "Cones", "Teddy", "Tsukuba", and "Venus". All of these four images have a high amount of texture. The set of images analyzed here also contains very weakly textured images because the proposed method should be applicable to a very wide variety of images.

A comparison of the root-mean-square error for all algorithms and images is given in table 8.1. Since the phase-based and Graph-cuts algorithms
both are deterministic they have a standard deviation of 0. Both segment-
based algorithms highly depend on the segmentation results, so their stan-
dard deviations can become pretty large when the segmentation method
has problems resolving the image into segments. These cases are good
starting points for further improvements of the segmentation method. In
some cases, problems in the segmentation phase are caused by a too large
errors in the pre-segmentation disparity map. This is shown below in the
more detailed analysis of the individual image results.

It should be noted that the results also have to be seen in terms of
the performance goal of reducing the processing times from 30 minutes
for original segment-based method, to less than 30 seconds for the more
complex texture-supported method. A performance comparison is provided
further below.

The proposed method has a significantly smaller mean RMS value and
standard deviation of $4.1 \pm 2.5$ compared to the other algorithms (original
segment-based: $8.6 \pm 8.4$, phase-based: $9.5 \pm 6.3$, Graph-cuts: $9.7 \pm 7.3$).
The largest outlier is “Wood” with an RMS of $8.3 \pm 1.9$ which is still below
the mean RMS of the other algorithms. These values show that the goal of
creating a generically applicable stereo disparity method has been achieved.

Moreover, the algorithm performs very well in this metric even when
looking at individual images. For eight of the eleven images the proposed
method performs better than the other three algorithms. For two of the
images it is the second-best method and for the last image the proposed
method has the third-best result. This demonstrates that the proposed
method can compete with and even outperform state-of-the-art algorithms.
In the following, a more detailed analysis is shown for each for each of the
test images.

**Baby**

The comparison of the Baby image results is shown in figure 8.3. Both the
cumulative and histogram error graphs show that the proposed method
performs better than the other three methods. As shown in the right part
of the histogram, there is a very small amount of large errors and almost
80% of all pixels have been matched with a disparity error of less than 0.5.

The $\delta$-map shows very well that, compared to the original method, the
proposed texture-supported method deals a lot better with the textured im-
age background. Also, the untextured areas in this image are resolved very
well with both methods. However, the proposed method has problems with
the baby’s left leg. This is caused by errors in the pre-segmentation (phase-
based) disparity map in exactly that region. These errors lead to segmentation problems. A better performing pre-segmentation method could solve this problem. The Graph-cuts algorithm's results are also shown in order to demonstrate that Graph-cuts has problems with large untextured areas like the wide box the baby is sitting on.

**Cones**

The "Cones" image is one of the four Middlebury stereo evaluation images. The Middlebury evaluation compares the cumulative error metric with $\delta = 1$. The Graph cuts algorithm is highly optimized for this image and metric, as shown in figure 8.4. The second-best algorithm is the proposed texture-supported method. The histogram shows that the errors are noticeably smaller than with the original segment-based method, with almost 20% of the errors being in the range of merely 1 disparity value. The RMS results in table 8.1 show a very different picture with Graph cuts having the worst results. This means that the Graph cuts algorithm has fewer, but therefore very large errors.

**Lampshade**

The result comparison is shown in figure 8.5. The proposed method is almost on par with the Graph cuts algorithm in the cumulative error. The histogram shows that the difference is mostly caused by large errors which can be easily seen in the left region of the $\delta$-map. In the $\delta$-maps it can also be seen that structures in the background are handled much better than with the original method and the large box at the top of the image does not interfere with the smaller box in front of it, anymore. On closer inspection it can be seen that while Graph cuts performs better in the background area it has more problems with the untextured foreground object. In the RMS benchmark, the proposed method’s results are very close behind Graph cuts. Both algorithms have a significantly better RMS value than the phase-based and the original disparity algorithms.

**Midd**

The "Midd" results comparison is shown in figure 8.6. Obviously, the difficulty with this image primarily lies in its untextured background. This image is only shown here in order to demonstrate the background disparity handling extension included in the proposed method and explained in section 5.4. When comparing the foreground objects in the $\delta$-map, the
proposed method performs very well and it has fewer problems with the untextured objects than Graph cuts and small details are resolved better than with the original method. Also, the proposed method has the best RMS value of the algorithms.

Monopoly
Figure 8.7 shows a comparison for the "Monopoly" image which has a mixture of textured and untextured areas. Foreground objects have significantly fewer errors in the proposed method. Especially Graph cuts has major problems in the weakly textured center area of the image. In contrast, the original method has difficulties resolving the textured areas. Only the phase-based algorithm has comparably good results as the proposed method, but it has difficulties detecting the boundaries of the wood piece at the center of the image. As with the previous image, the (less important) untextured background area causes major problems for all algorithms, but the background handling improvements in the proposed method allow it to perform slightly better in this area. Since the proposed method also has by far the smallest RMS error it can be considered to have the best performance for this image.

Plastic
The "Plastic" image, shown in figure 8.8 is largely untextured which causes major problems for all algorithms. The cumulative error and the histogram both show that the proposed method has significantly smaller issues than the other algorithms, though. Some of the errors can be traced back to the segmentation which cannot deal with the very large errors from the phase-based method. The original segmentation algorithm has the best RMS value for this image, followed by the proposed method which has a few very large errors that negatively affect the RMS value.

Rocks
In figure 8.9, all of the algorithms perform comparably well in the cumulative error and histogram comparisons. Graph-cuts and the phase-based method have slightly larger errors than the texture-supported and the original disparity methods, which is also visible in the RMS results in table 8.1. The proposed method has significantly better RMS results than the other algorithms, so it can be considered the best-performing algorithm for this image.
Teddy
The results for the "Teddy" image (see figure 8.10) show nicely that the proposed method has improved a lot in the textured area (top right) compared to the original method and the overall results are much better. Still, Graph cuts is better optimized in this particular benchmark. On the other hand, the proposed method has a better RMS value, indicating that Graph cuts has much larger outliers.

Tsukuba
The "Tsukuba" image results are can be seen in figure 8.11. This image contains quite a bit of texture and again the proposed method improves over the original method. Also, it has the best RMS of the four algorithms, but Graph cuts performs better in the cumulative error.

Venus
The "Venus" image results (figure 8.12) again show that the proposed method has significantly improved in textured areas when compared to the original method. Most errors are very small (up to two disparity values), so the overall quality is pretty good. Graph cuts can still achieve a higher accuracy in in the small $\delta$ range, but it has the worst results in the RMS benchmark, so it has large outliers.

Wood
The results for the "Wood" image are shown in figure 8.13. This image is almost textureless and the proposed method performs quite well in the cumulative and histogram benchmarks. The RMS value is not very good, though. The biggest problem is the bottom region in the left half of the image. In that area the phase-based algorithm has too large errors which disturb the segmentation results and thus cause problems for the texture-supported method. Improvements in the pre-segmentation disparity algorithm could solve most of these problems and would be a good starting point for future work.

8.3 Runtime comparison
The proposed method is 79±19 times faster than the original segment-based algorithm. Computation times are well below 30s. A speed comparison for
8.3. **RUNTIME COMPARISON**

<table>
<thead>
<tr>
<th>Image</th>
<th>Texture-supported</th>
<th>Original</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baby</td>
<td>(18.5 ± 0.1) s</td>
<td>(1359 ± 67) s</td>
<td>73x</td>
</tr>
<tr>
<td>Cones</td>
<td>(21.0 ± 0.2) s</td>
<td>(1499 ± 57) s</td>
<td>71x</td>
</tr>
<tr>
<td>Lampshade</td>
<td>(19.5 ± 0.4) s</td>
<td>(1496 ± 19) s</td>
<td>77x</td>
</tr>
<tr>
<td>Midd</td>
<td>(19.7 ± 0.2) s</td>
<td>(1930 ± 162) s</td>
<td>77x</td>
</tr>
<tr>
<td>Monopoly</td>
<td>(20.0 ± 0.5) s</td>
<td>(1559 ± 77) s</td>
<td>78x</td>
</tr>
<tr>
<td>Plastic</td>
<td>(18.6 ± 0.1) s</td>
<td>(1431 ± 33) s</td>
<td>77x</td>
</tr>
<tr>
<td>Rocks</td>
<td>(19.5 ± 0.1) s</td>
<td>(2250 ± 69) s</td>
<td>115x</td>
</tr>
<tr>
<td>Teddy</td>
<td>(20.5 ± 0.3) s</td>
<td>(1498 ± 86) s</td>
<td>73x</td>
</tr>
<tr>
<td>Tsukuba</td>
<td>(13.8 ± 0.2) s</td>
<td>(692 ± 84) s</td>
<td>50x</td>
</tr>
<tr>
<td>Venus</td>
<td>(19.6 ± 0.2) s</td>
<td>(1047 ± 62) s</td>
<td>54x</td>
</tr>
<tr>
<td>Wood</td>
<td>(20.7 ± 0.1) s</td>
<td>(2121 ± 180) s</td>
<td>103x</td>
</tr>
</tbody>
</table>

Table 8.2: Runtime performance comparison

Each image is shown in Table 8.2. The performance results are very promising and at the same time the quality of the results has also significantly improved.
Figure 8.1: Images used for comparison: (a) Baby (b) Cones (c) Lampshade (d) Midd (e) Monopoly (f) Plastic
Figure 8.2: Images used for comparison (continued): (a) Rocks (b) Teddy (c) Tsukuba (d) Venus (e) Wood
CHAPTER 8. RESULTS

Figure 8.3: Baby: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method
8.3. RUNTIME COMPARISON

Figure 8.4: Cones: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method
CHAPTER 8. RESULTS

Figure 8.5: Lampshade: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method
8.3. RUNTIME COMPARISON

Figure 8.6: Midd: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method
Figure 8.7: Monopoly: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method
8.3. RUNTIME COMPARISON

Figure 8.8: Plastic: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method
CHAPTER 8. RESULTS

Figure 8.9: Rocks: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method
8.3. RUNTIME COMPARISON

Figure 8.10: Teddy: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method
CHAPTER 8. RESULTS

Figure 8.11: Tsukuba: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method
8.3. **Runtime Comparison**

Figure 8.12: Venus: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method
Figure 8.13: Wood: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method
Chapter 9

Conclusion

The goal of this thesis was to create a general-purpose stereo disparity algorithm that deals well with textured and untextured images at the same time and thus is widely applicable. The secondary goal was to achieve runtimes below 30s despite the complexity of the implementation. A segment-based disparity algorithm was chosen as the starting point because it provided good performance in untextured scenes. Since that algorithm had problems in textured scenes, most of the work consisted of extending that algorithm to better deal with textured areas. The segmentation method was identified as the most critical component causing failures in textured areas. Thus, an important goal was the design of a texture-supported segmentation method that can yield valid results in textured areas. This goal was approached with edge-ignoring texture detection in order to find textured areas and discontinuity-preserving texture smoothing in order to remove texture from the stereo image’s color components. Several other improvements were added to the original segment-based disparity algorithm, some of which also allow it to perform better in weakly textured areas by reducing the number of outliers and making assumptions about the possible range of disparity values in certain areas of the image.

A texture-supported segment-based disparity algorithm was successfully implemented. As the results show, the goal of creating an algorithm that is widely applicable to very distinct images has been successfully achieved in this work. The proposed method performs particularly well in the RMS (root-mean-squared) benchmark, meaning there are only few and rather small outliers in the disparity map. In other words, most disparities are very close to the actual value. In contrast, Graph cuts sometimes has a lower percentage of erroneous pixels, but the errors are very large. This in turn leads to a large RMS. It can be argued that the error percentage alone is not very meaningful without taking the size of the error into account.
From this point of view, the RMS is a more meaningful metric. Still, the results chapter takes several metrics into account in order to provide a more differentiated analysis which can guide future improvements to the algorithm.

The results show that textured regions pose no real problem to the proposed method, anymore, so this extension of the original segment-based method can be seen as a very nice success. At the same time, the runtime performance has been improved over the original segment-based method by almost two orders of magnitude using a custom Python->GPU compiler. Overall, the results are very pleasing.
Chapter 10

Future work

10.1 Texture classification

Currently the texture segmentation algorithm can only distinguish between average texture color values. It would be very interesting to see if a more complex classification of texture improves the results. For example, texture could be categorized into "dots", "lines", and "chaos". Also, "lines" could be further categorized by the direction of the lines in steps of 15°, for example.

10.2 Interpolation

The spring-mass model interpolation is very time-consuming. It would be desirable to find a more efficient alternative that is also more robust to errors. Currently, the edge disparity has an overwhelming influence on the final results and the inner-segment disparity can only make small corrections. The interpolation step could also be extended with shape fitting to allow detecting planes and round surfaces and improve their respective disparity results.

10.3 Speed optimizations

The next step for this algorithm is obviously to achieve real-time video performance. This can be partially achieved with faster hardware. At the time of this writing NVIDIA already offers significantly faster GPUs and in case of video processing the individual algorithm components could be distributed on across multiple GPUs. Also, the inner-segment disparity could
be optimized using a coarse-to-fine strategy in which the segment is first analyzed at a coarse zoom level in order to find a small acceptable disparity range and in the next step the exact value could be determined within that limited disparity range. Additionally, improvements to the py2gpu compiler can probably yield significant speed-ups for some components. However, in order to achieve real-time video performance several other optimizations would need to be exploited.

10.4 Edge disparity smoothness constraints

The edge disparity can sometimes have large outliers. In particular, edges that are almost parallel to the x-axis often result in bad edge disparities. This problem could be solved with a smoothness constraint that takes several neighboring edge pixels into account.

10.5 Integration into segmentation

It might be possible to improve the overall segmentation and disparity results by tightly coupling the segmentation and disparity algorithms in a feedback-loop. The segmentation algorithm depends on a disparity map to connect segments in the left and right images. The disparity algorithm depends on a segment map to yield good disparity values. It might be beneficial to feed the intermediate results of every iteration step of the segmentation algorithm into the disparity algorithm and then feed the disparity values back to the segmentation algorithm’s next iteration step. This would allow the segmentation algorithm to dynamically adjust the connected segments based on the dynamically improving disparity map and possibly fix errors that currently appear in the segment map.


Appendix A

Usage of py2gpu

GPU functions are specified via the `@blockwise` decorator which takes two required arguments both of which are dicts. The following code snippet which simply multiplies a matrix with an integer demonstrates how to work with the decorator:

```python
from py2gpu.api import (blockwise, Int32Array, FloatArray)
from numpy import array

@blockwise({('output', 'input'): (1, 1)},
           {('output', 'input'): Int32Array,'factor': int})
def multiply(output, input, factor):
    output[0, 0] = factor * input[0, 0]

input = array([[[...], ...]])
output = input.copy()
multiply(output, input, 5)
# On the CPU this would be the same as:
# output = input * 5
```

The first argument to `@blockwise` maps the function’s argument names to their respective window size in pixels. Here, the function’s `output` and `input` arguments are specified as $1 \times 1$ px windows. Instead of using hard-coded integers to specify window dimensions it is possible to use strings referring to argument names: `input`: `('factor', 'factor')` would use $\text{factor} \times \text{factor}$ sized windows.

The second argument denotes the types of the variables. In this case `output` and `input` both are 32-bit integer matrices and `factor` is an integer.
GPU functions cannot return anything because they are executed in parallel on multiple threads, so the output array always has to be provided as an argument (here: `output`). As can be seen in the code snippet it is possible to use multi-dimensional coordinates to access the matrices and the coordinates are relative to the current window position, i.e. in each thread `input[0,0]` points to a different position in the matrix and the compiler starts enough threads to process the whole matrix.

A more interesting example is the possibility to combine different window sizes and even use matrices that are not split into windows:

```python
@blockwise({ 'x ': (1 , 1) , 'y ': (2 , 2)}
            {( 'x ', 'y ', 'z '): FloatArray})
def reducer(x, y, z):
x[0 ,0] = z[0 ,0] * y[0 ,0] + z[0 ,1] * y[0 ,1] +
        z[1 ,0] * y[1 ,0] + z[1 ,1] * y[1 ,1]
```

Here, `z` is treated a matrix with an absolute coordinate system. All threads can access the whole `z` matrix and the `z`’s origin (0,0) is the same for all threads. In contrast, `x` and `y` are split into windows, but unlike the first code snippet the window size of `y` is twice the size of `x`. For this to work, the `y` matrix must be twice the size of the `x` matrix. The compiler automatically verifies whether the proportions of the provided arguments are correct.

In the example above the 2×2px windows are non-overlapping. However, several components of this algorithm, e.g. the interpolation step (section 5.5) and the texture detection step (section 6.2), have to work with overlapping windows. Such an operation mode can be enabled by passing `overlapping=True` to the decorator:

```python
@blockwise({ 'x ': (1 , 1) , 'y ': (2 , 2)})
            {( 'x ', 'y '): FloatArray},
    overlapping=True)
```
```python
def overlap(x, y):
    x[0,0] = y[0,0] + y[0,1] + y[1,0] + y[1,1]

An example of overlapping windows is shown in figure A.1. Since the windows are overlapping the y[0,1] coordinate of the first thread T1 points to the same location as the y[0,0] coordinate of the second thread T2. With overlapping=True all arguments are treated as overlapping. Alternatively, it is possible to limit this feature to a specific set of parameters by using, for instance, overlapping=(‘x’, ’y’) to treat only x and y as overlapping and all other parameters as non-overlapping.

Finally, the last important feature is called threadmemory. This allows to allocate a certain amount of private memory to each thread. Currently, this does not utilize the GPU’s shared memory, but instead tries to fit everything in register memory and if that is impossible it falls back to global memory. The following shows a more complicated example which combines several of the previously explained features and uses a temp threadmemory variable:

```python
@blockwise({ 'x' : ( 'n' , 1)} ,
            {('x' , 'temp'): Int32Array ,
             ('n' , 'temp_sum'): int },
            threadmemory={'temp' : ( 'n' , 1)})

def reversed_elems(x, n):
    # Invert elements
    for i in range(n):
        temp[i, 0] = x[n-i-1, 0]
    # Calculate the sum of temp’s elements
    temp_sum = temp.sum()
    # Write new result back to x
    for i in range(n):
        x[i, 0] = temp_sum * temp[i, 0]
```

In this example the 2D matrix x is split one-dimensional windows of size n×1 and the elements of each window are reversed and multiplied with the sum of the elements. Here, the sample code also makes use of the high-level array API by calling the sum() method on temp. Other available methods are max(), min(), mean() and the binary inversion (0=>1, 1=>0) via invert().
## List of Figures

1.1 Stereo cameras taking an image of an object (top) and the corresponding depth map (bottom) .......................... 1
1.2 Preferred camera alignment ................................................. 2
1.1 Point correspondence geometry [23] ................................. 6
1.2 Rectification of two images [19]: (a) Original image pair overlaid with epipolar lines (b) Images transformed, so epipolar lines are parallel (c) Images transformed, so epipolar lines are horizontal and in vertical correspondence (d) Final result with minimized distortions ................................. 7
2.1 Gabor filter bank used for extracting phase at eight orientations. Even filters are in the top row. Odd filters are in the bottom row. [25] ................................................. 10
3.1 Block diagram of the algorithm. [10] ............................... 16
3.1 Spring-mass model: The amplitude of the mass represents its disparity. Masses belonging to the same segment are connected via $F_{int}$. Additionally, $F_{data}$ contains forces for the edge disparity and inner-segment disparity. [10] ................................................. 20
4.1 Block diagram of the proposed algorithm. New components are in green. Modified components are in blue. The corresponding intermediate results are shown in figure 4.1. ............................... 29
4.1 Results of each processing step; right image data is not shown: (a) Left input image (b) Texture detection (c) Smoothed and blurred (d) Pre-segmentation disparity (e) Segmentation of left image (f) Segmentation of right image. Continued in figure 4.2. ............................... 30
4.2 Continued results of each processing step: (a) Edge disparities (b) Edge disparity average (c) Occlusion map (d) Inner disparities (e) Final disparity / interpolation ............................... 31
5.1 Center of mass shifting negatively when objects occlude each other ........................................ 36
5.2 Occluded areas of a segment (gray): black: new algorithm, red+black: previous algorithm ............ 37
5.1 Spring-mass model of new algorithm incorporating pre-segmentation disparity for textured areas. .................................................. 40
6.1 Sample images and their component count. [3] .......................................................... 46
6.2 Comparison of texture detection results (original paper vs modified algorithm) ............................ 48
6.1 Smoothing results: (a) Original image (b) Fine smoothing (c) Coarse smoothing .......................... 53
7.1 Parallelization with py2gpu using pixel-sized windows .................................................... 56
7.2 Parallelization of interpolation component ........................................................................ 56
7.1 Four threads assigned to four windows ............................................................................ 58
7.2 Processing pipeline in the py2gpu compiler ...................................................................... 59
8.1 Images used for comparison: (a) Baby (b) Cones (c) Lampshade (d) Midd (e) Monopoly (f) Plastic .................................................................................. 70
8.2 Images used for comparison (continued): (a) Rocks (b) Teddy (c) Tsukuba (d) Venus (e) Wood ...................................................................................... 71
8.3 Baby: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method ............................................................................ 72
8.4 Cones: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method ...................................................................................... 73
8.5 Lampshade: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method ................................................................. 74
8.6 Midd: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method ...................................................................................... 75
8.7 Monopoly: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method ................................................................. 76
8.8 Plastic: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method ...................................................................................... 77
8.9 Rocks: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method ...................................................................................... 78
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.10</td>
<td>Teddy: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method</td>
</tr>
<tr>
<td>8.11</td>
<td>Tsukuba: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method</td>
</tr>
<tr>
<td>8.12</td>
<td>Venus: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method</td>
</tr>
<tr>
<td>8.13</td>
<td>Wood: (a) Cumulative error (b) Error histogram (c) Proposed method (d) Original method (e) Graph-cuts (f) Phase-based method</td>
</tr>
<tr>
<td>A.1</td>
<td>Overlapping windows</td>
</tr>
</tbody>
</table>
Bibliography


BIBLIOGRAPHY


