Master Thesis

Real-time stereo-matching on embedded hardware for MAVs

Author(s):
Dufour, Pascal

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006422922

Rights / License:
In Copyright - Non-Commercial Use Permitted
Real-Time Stereo-Matching on Embedded Hardware for MAVs

Pascal Dufour

Master Thesis
15. April 2010

Advisor: Dr. Friedrich Fraundorfer
Supervisor Prof. Marc Pollefeys
Abstract

The goal of this thesis was to use stereo-matching onboard a micro air vehicle to compute disparity maps in real-time. A complete stereo-matching system was needed. This includes the hardware side with the camera setup, preparing the images for matching, the actual matching algorithm, post-processing of the disparity map and finally making the result available to the system to be used by other processes. Because of the limitations of processing power, methods other than the computation on the CPU were investigated and evaluated.

Stereo-matching has been around for decades and has been studied in depth. However, it was never feasible to do stereo-matching onboard a MAV. The processing power of embedded computers just wasn’t high enough for real-time stereo-matching and the hardware for mobile platforms was still too big and heavy to be incorporated onto a MAV. During the last few years this has changed and it’s now possible to run a stereo-matching process in the background and receive disparity maps in real-time.

While stereo-matching has been successfully applied in robotics for some time now, the requirements for a MAV are very different. If the disparity map has any influence on the behaviour of the MAV like obstacle avoidance, it has to be fast enough and real-time. The result should also be robust, meaning not too sensitive to noise, brightness or other changing parameters. And because stereo-matching is just one of many processes on the MAV, it should be as efficient as possible.
Acknowledgement

During the last six month I was part of the PIXHAWK student project. It has been a great experience and I would like to take this opportunity to thank everybody who is involved or contributes to the PIXHAWK project.

I would like to thank Prof. Marc Pollefeys to make the PIXHAWK project possible.

I would especially like to thank Friedrich Fraundorfer for continuously providing his support. The discussions with him were always interesting and a great source of ideas.

The whole team was always ready to provide help with difficulties and the working environment was very positive. I would like to thank Lorenz Meier for leading the team and helping me with the more difficult aspects of the PIXHAWK system.
# Contents

1. **Background** 1
   1.1. Introduction 1
   1.2. Stereo-Vision 2
      1.2.1. Correspondence Problem 2
      1.2.2. Epipolar Geometry and Triangulation 2
      1.2.3. Undistortion 3
      1.2.4. Rectification 4
   1.3. State of the Art 5
   1.4. Environment 6
      1.4.1. Hardware 6
      1.4.2. Software Architecture 7
   1.5. Requirements Analysis 9
      1.5.1. Quality and Use of the Disparity Map 9
      1.5.2. Camera and Lenses 10
      1.5.3. Baseline 10
      1.5.4. Performance 14
      1.5.5. Decisions 14

2. **Implementation** 15
   2.1. Image Acquisition 16
   2.2. Undistortion and Rectification 16
      2.2.1. Calibration 17
   2.3. Matching 18
      2.3.1. Sum of Absolute Differences 18
      2.3.2. Sum of Squared Differences 19
Contents

2.3.3. Dynamic Programming ................................................. 19
2.3.4. Performance Improvements ........................................... 19
2.3.5. SAD on the GPU ......................................................... 24
2.4. Post-Processing ............................................................... 31
  2.4.1. Confidence map ....................................................... 31
  2.4.2. Median Filtering ....................................................... 32
2.5. Output ................................................................. 33

3. Evaluation ............................................................... 35
  3.1. Quality Evaluation ...................................................... 35
    3.1.1. Evaluation of Block-Matching Algorithms ....................... 37
    3.1.2. Evaluation of Dynamic Programming Methods .................. 40
    3.1.3. Evaluation of the GPU Implementation ......................... 42
  3.2. Performance Evaluation ............................................... 43
    3.2.1. Undistortion and Rectification ................................ 43
    3.2.2. Matching .......................................................... 44
    3.2.3. Post-Processing .................................................. 45

4. Conclusion and Outlook ............................................. 47

A. Images used for the Evaluation .................................. 49

Bibliography ............................................................... 55
List of Figures

1.1. Correspondence Problem ................................................. 3
1.2. Epipolar Geometry .......................................................... 4
1.3. Camera Alignment ........................................................... 5
1.4. Overall Software Architecture ............................................. 8
1.5. On-board Software Architecture ......................................... 9
1.6. Camera Baseline .............................................................. 11

2.1. Stereo-Matching Framework .............................................. 16
2.2. Horizontal Sliding Window ................................................ 20
2.3. Vertical Sliding Window .................................................... 22
2.4. Subtraction using SSE ....................................................... 23
2.5. SSE minpos() function ...................................................... 23
2.6. Image Sampling .............................................................. 27
2.7. Sampling with Bilinear Interpolation ..................................... 28
2.8. Convolution ................................................................. 28
2.9. Resulting Convolution Distribution ..................................... 29

3.1. Images used for quality evaluation ..................................... 36

A.1. Evaluated Regions .......................................................... 50
A.2. Tsukuba Disparity Maps .................................................. 51
A.3. Venus Disparity Maps ...................................................... 52
A.4. Teddy Disparity Maps ...................................................... 53
A.5. Cones Disparity Maps ...................................................... 54
List of Figures
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Baseline Evaluation</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Specifications of Matching Requirements for Middlebury’s Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Evaluation of SAD and SSD, threshold=1</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation of SAD and SSD, threshold=2</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Evaluation of Dynamic Programming with threshold=1</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Evaluation of Dynamic Programming with threshold=2</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation of the GPU Implementation</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Performance of Undistortion and Rectification</td>
<td>43</td>
</tr>
<tr>
<td>3.8</td>
<td>Performance of Matching Algorithms</td>
<td>44</td>
</tr>
<tr>
<td>3.9</td>
<td>Performance of Confidence Map Computation</td>
<td>45</td>
</tr>
</tbody>
</table>
Background

1.1. Introduction

The the PIXHAWK (http://pixhawk.ethz.ch) project focuses on computer vision onboard a micro air vehicles (MAVs) to allow autonomous action. The goal is to do all the image processing onboard to reach full autonomy. This means many mission-critical processes have to run simultaneously, leading to limited processing power. A stereo-matching process should therefore be very lightweight, while still providing robust disparity maps that can be used for object avoidance.

Chapter 1 starts with the theory behind stereo-matching and also gives an overview of the advances of stereo-matching and what can be done today. The current PIXHAWK system is then described to provide an overview of the hardware and software in place, since this has large implications on the algorithms chosen for the implementation. The last section is the requirements analysis which sets the ground for the implementation and evaluation in the later chapters.

Chapter 2 describes the implementation of the evaluated algorithms as well as the overall framework. This includes all steps necessary to get a good disparity map: image acquisition, rectification, matching, post-processing and output of the disparity map.

Chapter 3 focuses on the evaluation of the different algorithms. This allows the direct comparison of the algorithm. Both quality of the disparity maps and the performance were evaluated.

Finally, the report concludes with an outlook of what could be improved, what is still left to do and what the disparity map could be used for.
1. Background

1.2. Stereo-Vision

Computer stereo-vision is the process of extracting depth information from images recorded from multiple viewpoints. There are different methods to acquire these images:

- **Single, moving camera**: A single camera is moved in space, recording images, thereby providing images from different viewpoints. This implies a more or less static scene, as moving objects would lead to errors in the depth map, or they would have to be filtered out or compensated for. It also means that the relative displacement of the viewpoints is not known and first has to be found using some form of image registration.

- **Multiple camera setup**: If a fixed camera array is used to acquire images from different viewpoints, the exact relative positions of all cameras are known and no image registration is required. All images should however be captured at the same time in order to ensure that no errors result from moving objects or movement of the camera array.

1.2.1. Correspondence Problem

The correspondence problem is the problem of finding correspondences between recorded points in two or more images that were taken of the same 3D scene, but from different viewpoints. It is used for example in image stitching, the process of combining multiple images which show some overlapping regions into a bigger panoramic image. In stereo-matching, the corresponding points can be used to triangulate the position of the point in the real 3D scene which allows the reconstruction of the scene.

Figure 1.1 is an example using two images from the Tsukuba image set from the Middlebury dataset [16]. The two images were taken with a horizontal displacement, which is visible if the objects closer to the camera (e.g. the lamp) are examined. The horizontal shift between the two images for these objects is larger than the shift of the background. By solving the correspondence problem, a disparity map can be built. In figure 1.1, two example pairs of corresponding points are indicated, with their position on the disparity map. The disparity map is a simple depth map, brighter areas are closer to the foreground and darker areas closer to the background. Note that this disparity map is actually the ground truth, not a disparity map computed by an algorithm.

It is important to keep in mind that is not possible to find perfect correspondences. Instead, it’s just a matter of finding the most probable correspondences.

1.2.2. Epipolar Geometry and Triangulation

Given a reference point in one image, finding that point in the other image basically requires a search over the whole image. This is not very efficient and can be optimized by exploiting geometric relations between the images. If the relative translation and rotation of the viewpoints are known, epipolar geometry can be used to compute the position of a point visible in multiple images. To illustrate this, a stereo camera setup with two pinhole cameras is used. Figure 1.2 is an example of such a setup.
1.2. Stereo-Vision

Fig. 1.1: Correspondence Problem. If the correspondence problem is solved, it can be used to triangulate points and determine at which depth they are in the real 3D scene.

If only one image is given, e.g. the one from the left camera, it’s impossible to determine the position of a recorded point $X_L$. It could be anywhere along the line that passes through $O_L$ and $X_L$. It could for example be any of the points $X_1$, $X_2$ and $X_3$. But if a second image is used and the corresponding projected point $X_R$ in that image is known, triangulation can be used to compute the exact position in space.

On the other hand, given the projected point $X_L$ in the left image, the projected point $X_R$ has to be somewhere along the red line (through $e_R$ and $X_R$). This is called the epipolar constraint and is often used in stereo-matching algorithms. Any point in one image has an epipolar line in the other image, on which the projected point has to be found. This constraint makes it much easier when searching for corresponding points. Once that point is found, the position in space can then easily be determined.

Note that the example takes only two images into account. However, if more cameras are used, the epipolar constraint still holds; a given point in one image has to lie on exactly one epipolar line in each of the other images.

1.2.3. Undistortion

Standard epipolar geometry works only on images captures with an ideal pinhole camera. It would be possible to expand the geometric relations of epipolar geometry to other camera models, but the gain from this would not be as big as with standard epipolar geometry. For example,
1. Background

The point $X_L$ is recorded on the left image. If we only have that image, it $X$ could be anywhere on the line that passes through the points $O_L$ and $X_L$. However, with the image of the right camera, we can determine the exact position of $X$.

Figure 1.2.: Epipolar Geometry. The epipolar line in an image from a pinhole camera would actually be a curve in a lens-based camera. Finding a correspondence along such a curve is much more difficult than finding it along a straight line. It is therefore much easier to approximate the camera as a pinhole camera by modifying the recorded image. This process is known as undistortion, as it removes the distortion inherent in an image taken with a lens-based camera.

1.2.4. Rectification

The search of corresponding points becomes even easier when the epipolar lines would match horizontally. Then, corresponding points would have to be on the same horizontal line in both images. This is actually the case if the cameras are aligned to have the same image plane with no relative rotation and only relative translation along the baseline. See figure 1.3 for such a setup.

In reality, such a precise aligned camera setup is almost impossible to build. For example, only a tiny relative rotation between the cameras around the baseline (horizontal axis) would mean a large offset of the epipolar line.

It is much easier to build an imperfect camera mount and correct the images later [8]. Because the images at this step are already undistorted, the correction is a simple linear transformation.
Figure 1.3: Camera Alignment.

If the cameras are aligned to have the same image plane with no relative rotation and only horizontal translation, the epipolar lines become horizontal and match horizontally (move through $X_L$ and $X_R$).

1.3. State of the Art

The correspondence problem can be solved in a variety of ways. One possibility is to compare regions of textures in both images. Well known methods include SAD, SSD, census based matching and using correlation to compare the textures from both images. The advantage of these methods is that they can be easily optimized with sliding windows [18].

Instead of comparing textures, the correspondence problem can also be solved by finding corresponding features [2, 1]. While correspondences might be easier to find, the result is usually a sparse disparity map, meaning the depth information is not computed everywhere in the images.

A lot of the recent research went into global optimization methods. Because the correspondence problem cannot be solved perfectly and there are always ambiguous correspondences, optimizing a cost function over the whole image gives a much better result. Dynamic programming is often used as such an optimization method [9, 13]. Other well known optimization methods are max-flow/min-cut optimization [3, 12], nonlinear diffusion [15] or belief propagation [20].

Often, the algorithms are combined with other techniques like subregioning [19, 13]. This might yield better results or speed up the matching. Very interesting is also current research about parallelization of the algorithms [7]. Especially global methods are computationally complex and mostly just too slow for a real-time implementation. If an algorithm can be executed in par-
1. Background

allel, it could be implemented in hardware, allowing real-time performance [17]. This is very exciting especially for machine vision, and a lot of research in that area also comes from that background.

These hardware implementations usually use field-programmable gate arrays (FPGAs) for testing and development. Another possibility is to use graphics processing units (GPUs) [11]. The option to use programmable shaders makes them very interesting for stereo matching and computer vision in general, as they can outperform a CPU because of their highly parallel structure.

An excellent overview of existing algorithms is [5] and also the Middlebury Stereo Vision website (vision.middlebury.edu/stereo/) and their evaluation and comparison of stereo-matching algorithms [16].

There are many applications for stereo-matching. They range from object avoidance [14] to occupant detection in cars [?]. With increasing processing power, real applications become reality. For example, finished multi-camera stereo systems can be bought, e.g. from PointGrey Research (www.ptgrey.com/). Or the famous Hawk-Eye system (www.hawkeyeinnovations.co.uk) used in tennis games is another example. It is based on an array of multiple high-speed cameras and solving the multiview correspondence problem to accurately keep track of the ball and is already part of the adjudication process.

1.4. Environment

1.4.1. Hardware

There are currently two different MAVs used in the PIXHAWK project. A smaller, coaxial helicopter with a single board computer from Gumstix Inc, and a newer more powerful quadrotor helicopter with a COM Express board and an Intel Core 2 Duo.

Gumstix Overo Fire

The older platform used on the coaxial helicopter is a single board computer from Gumstix Inc, the Gumstix Overo COM Fire.

Specifications:

- System-on-Chip: OMAP 3530
  - 600 MHz ARM Cortex-A8 with NEON SIMD Coprocessor
  - 430 MHz TMS320C64x+ DSP (fixed point, six parallel vector units)
  - 110 MHz SGX GPU
- RAM: 256 MB mDDR
- Size: 17 mm x 58 mm

The processing power is of course limited. The idea is that it is very powerful compared to it’s
size and weight, which are critical factors for an MAV. Probably the most interesting component of this system is the Digital Signal Processor (DSP), which could potentially handle stereo-matching on its own. However, the processing power is simply not enough to handle full autonomy. Stereo-matching in this context does not really provide a big advantage, as object avoidance won’t be necessary.

**Intel Core 2 Duo**

The newer platform is a COM Express board mounted on quadrotor helicopter.

Specifications:

- Intel Core 2 Duo SL9400 CPU: 1.86GHz, 6MB L2 Cache
- RAM: 2GB DDR3, 1066 MHz
- Intel GS45 chipset
  - Intel GMA X4500 integrated graphics
  - 10 Shader units, programmable
- Size: 95 mm x 95 mm

This platform has enough power to support full autonomy. The two interesting components for stereo-matching are the CPU and the GPU with the programmable shaders.

### 1.4.2. Software Architecture

The PIXHAWK software is designed to be modular and extensible. This is important because the software parts for an MAV can become quite complex. The software doesn’t only include the on-board processes, but also the flight board with the IMU (Inertial Measurement Unit), the ground-station and the communication between the software parts. This is why a well structured architecture is so important. Figure 1.4 provides a diagram of the overall software architecture.

The PIXHAWK system is split up into three main parts:

- **The ground-station** serves as interface to the autonomous MAV. It displays data during flight and provides some minimal control of the MAV.

- **The flight board** is an IMU on-board the MAV to handle the sensor data. The most important sensing capabilities are acceleration, angular velocity, the earth magnetic field and barometric pressure. The data from these sensors is used to stabilize the helicopter, provide information of the attitude to other parts of the system and should finally allow autonomous control of the MAV together with the vision data.

- **The on-board software** runs either on the single board computer (Gumstix Overo COM) or the Intel Core 2 Duo COM Express board. The board is mounted on the MAV, so all processing can be done completely without any communication from the ground-station.
1. Background

![Overall Software Architecture](image_url)

**Figure 1.4.** Overall Software Architecture
1.5. Requirements Analysis

The stereo-matching process has to run as a module on-board the MAV and does not depend on any sensor data or the ground-station. Figure 1.5 provides an overview of the design of the on-board software.

The communication is handled with an inter process communication protocol, currently LCM from MIT. It is targeted towards real-time systems where low latency is important. Adding functionality can easily be accomplished by implementing a new process.

1.5. Requirements Analysis

1.5.1. Quality and Use of the Disparity Map

The disparity map will be used mainly for object avoidance. The provided disparity map will have to be queried constantly to detect if the MAV is getting too close to an object, and if this is the case, an evasive maneuver has to be initiated.

Possibly in a later step it could also be used as part of a SLAM (simultaneous localization and mapping) process, though the exact use of stereo-matching in that implementation is not defined yet. It is also possible that the disparity map will be used for visualization on the base-station.
1. Background

The focus is clearly towards object avoidance, as this is the main use of the stereo-matching. This means that while the algorithm should be robust, the quality of the resulting disparity map does not have to be too high. The important thing is that close objects are actually matched correctly and not smoothed over or ignored in any way. If this is taken into account, the algorithm should provide a conservative disparity map, meaning it should assume to matched points too close rather than too far away. It also means that the resolution of the disparity map doesn’t have to be too high.

1.5.2. Camera and Lenses

The first step when building a stereo-matching system is the decision of what hardware to use. The cameras used in the PIXHAWK project are Point Grey Firefly MV cameras. They take images with low noise, which simplifies stereo-matching. The cameras come without lenses, so it’s possible to mount a wide variety of lenses with different focal length. The trade-off is simple, the wider the angle of the lens, the more the image covers of the scene, but the more the depth perception is limited to the foreground. Another negative effect of wide-angle lenses is that they have more distortion, which makes the correction less exact.

See table 1.1 for an example covering multiple lenses.

1.5.3. Baseline

The baseline, the horizontal displacement between the two cameras, has important implications on the runtime and quality of the stereo-matching algorithm.

As the baseline increases and the viewpoints of the recorded images are further apart, the potential depth perception of that system also increases. Points in the foreground of the 3D scene will have a bigger disparity (horizontal shift between the images) if the baseline is bigger. This means that the depth resolution increases since small differences in depth that are not visible with a small baseline become visible with a larger baseline.

On the other hand, an increased depth resolution also means an increased disparity search range. This leads to an increased runtime.

When designing an algorithm for stereo-matching, the requirements are usually not the depth resolution. Rather, the requirements are the distances, at which points can still be successfully matched, given a specified disparity search range.

On one side, the disparity search range defines the distance to the points closest to the cameras for which a successfully correspondence can be found. On the other hand, the baseline defines the points with the largest distance to the cameras that can be distinguished before the background.

Figure 1.6 shows a camera setup seen from above. A point \( Q \) is seen by both cameras and recorded on both images. The projection of \( Q \) onto the images is defined by the position of \( Q \), the focal length \( f \) and the Baseline \( E \).

\( Q \) is projected onto the image of the first camera at position \( d_{p_1} \). In the real scene, \( Q \) has a
Figure 1.6: Camera Baseline.
Two cameras seen from above. A point $Q$ is seen by both cameras. Not only the position of $Q$, but also the Baseline $E$ and the focal length $f$ define the projection of $Q$ onto the sensors.
1. Background

distance from each optical axis, defined here as $d_{x_1}$ and $d_{x_2}$. Applying the intercept theorem to this, the equation relations between $f$ and $Q$ become clear.

\[
\frac{d_{p_1}}{d_{x_1}} = \frac{f}{D} \quad (1.1)
\]

\[
\frac{d_{p_2}}{d_{x_2}} = \frac{f}{D} \quad (1.2)
\]

The baseline $E$ can be described as the difference between $d_{x_1}$ and $d_{x_2}$, as in equation 1.3. Note that in order to simplify the equations, $d_{x_1}$ and $d_{x_2}$ are defined as being positive above their respective optical axis and negative below.

\[
E = d_{x_2} - d_{x_1} \quad (1.3)
\]

**Disparity computation**

The disparity can be computed, when the projected points of $Q$ are not recorded on the same pixel on both images. This is the case if the difference between $d_{p_1}$ and $d_{p_2}$ is more than the size of a single pixel on the image, as shown in equation 1.4, where $s_p$ is the pixel size.

\[
|d_{p_2} - d_{p_1}| \geq s_p \quad (1.4)
\]

Note that $d_{p_2} - d_{p_1}$ is always positive. Now $d_{p_1}$ and $d_{p_2}$ are substituted using equations 1.1 and 1.2.

\[
\frac{dx_2 \cdot f - dx_1 \cdot f}{D} = \frac{(dx_2 - dx_1) \cdot f}{D} = \frac{E \cdot f}{D} \geq s_p \quad (1.5)
\]

Equation 1.5 states the relationship between baseline, focal length and distance to a recordable disparity before infinity. This is extended in equation 1.6 to include all computable disparities $k$.

\[
\frac{E \cdot f}{D} = k \cdot s_p \quad (1.6)
\]

Note that the disparity can be defined as the difference between the projected points in the two images, given a real point in a 3D scene. That difference is discrete, as the projection has to be recorded on a discrete sensor.

**Baseline Requirements**

To do object avoidance on the MAV, it is important that objects relatively close to the helicopter could be detected successfully. On the other hand, the MAV is built for operation in an indoor environment. This means that the distance of the point correspondences distinguishable
before the background does not have to be very large. Using that argumentation, the following
requirements were made:

- **Closest/largest disparity**: The stereo-matching system should be able to successfully
  match correspondences of points that are at least 0.5 meters away from the cameras.

- **Furthest/smallest disparity**: The stereo-matching system should be able to distinguish
  the last disparity before the background no less than 10 meters away from the cameras.

Keep in mind that these requirements were made on assumption of the behavior of the MAV
during autonomous flight. However, during this master thesis, the MAV was not yet capable of
autonomous flight and object avoidance has not been implemented and tested yet. The critical
parameter is the closest disparity described above. Should it prove to be too far away from the
cameras, the disparity search range will have to be improved to bring it closer to the foreground.

Using equation 1.6 with parameters specific to the PtGrey Firefly MV camera and lenses, it is
possible to exactly compute the closest and furthest disparity. The pixel size on the sensor is
0.0141\text{mm}, given a horizontal resolution of 320 pixels and a horizontal sensor size of 4.51\text{mm}.
This is reflected in equations 1.7, which shows the distance to the closest and furthest disparity,
$D_{\text{min}}$ and $D_{\text{max}}$.

\[
D_{\text{min}} = \frac{E \cdot f}{D} = 1 \cdot 0.0141\text{mm}, \quad D_{\text{max}} = \frac{E \cdot f}{D} = k_{\text{max}} \cdot 0.0141\text{mm} \quad (1.7)
\]

Table 1.1 shows the computation for a disparity search range of 30 pixels (e.g. the closest
disparity has a displacement of 30 pixels.

<table>
<thead>
<tr>
<th></th>
<th>f=3 mm</th>
<th>f=3.6 mm</th>
<th>f=4 mm</th>
<th>f=6 mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>close far</td>
<td>close far</td>
<td>close far</td>
<td>close far</td>
</tr>
<tr>
<td>15</td>
<td>1.0643 31.9290</td>
<td>1.2772 38.3149</td>
<td>1.4191 42.5721</td>
<td>2.1286 63.8581</td>
</tr>
<tr>
<td>10</td>
<td>0.7095 21.2860</td>
<td>0.8514 25.5432</td>
<td>0.9460 28.3814</td>
<td>1.4191 42.5721</td>
</tr>
<tr>
<td>9</td>
<td>0.6386 19.1574</td>
<td>0.7663 22.9889</td>
<td>0.8514 25.5432</td>
<td>1.2772 38.3149</td>
</tr>
<tr>
<td>8</td>
<td>0.5676 17.0288</td>
<td>0.6812 20.4346</td>
<td>0.7568 22.7051</td>
<td>1.1353 34.0576</td>
</tr>
<tr>
<td>7</td>
<td>0.4967 14.9002</td>
<td>0.5960 17.8803</td>
<td>0.6622 19.8670</td>
<td>0.9933 29.8004</td>
</tr>
<tr>
<td>6</td>
<td>0.4257 12.7716</td>
<td>0.5109 15.3259</td>
<td>0.5676 17.0288</td>
<td>0.8514 25.5432</td>
</tr>
<tr>
<td>5</td>
<td>0.3548 10.6430</td>
<td>0.4257 12.7716</td>
<td>0.4730 14.1907</td>
<td>0.7095 21.2860</td>
</tr>
<tr>
<td>4</td>
<td>0.2838  8.5144</td>
<td>0.3406 10.2173</td>
<td>0.3784 11.3525</td>
<td>0.5676 17.0288</td>
</tr>
</tbody>
</table>

**Table 1.1:** Baseline Evaluation for a disparity search range of 30 pixels. The values are the
distances to the one-pixel disparity and 30-pixel disparity, assuming an image with 320 horizontal
pixels. The results are in meters.
1. Background

1.5.4. Performance

Stereo-matching has to be real-time. In order to do object avoidance, the frequency has to be high enough to detect objects before the MAV is already too close to them. The helicopter can’t just stop in mid air, so the momentum should always be taken into account.

It was guessed that a frequency of 10 Hz should suffice to successfully evade objects, however this is not based on any data or tests! Once autonomous flight is possible, this will obviously have to be evaluated.

Because stereo-matching is just one of many processes running on-board the MAV, it should be as efficient as possible. In order to reach autonomous flight, at least a visual odometry is necessary. In the future, a SLAM implementation might also be possible, which would require additional computing power. Additional processes might not be mission critical, but still important.

1.5.5. Decisions

The following decisions were made using the arguments and/or guesses above:

- The algorithm should produce at least a **320x240 pixel output disparity map**.
- The disparities should at least include the **range from 0.5 m to 10 m**
- A **5 cm baseline** is used. An additional 8 cm baseline is provided on the camera setup on the MAV.
- The algorithm should run with a **frequency of 10 Hz** or more.
- The algorithm should not require more than **20% of the CPU load**. That’s 20 ms given a frequency of 10 Hz.
- The algorithm should provide **robust** results.
Implementation

Stereo-matching as a functional part of an MAV software requires more than just the matching algorithm. A complete framework that takes care of all necessary steps is needed. To make the framework more useful, it has also been extended to include additional functionality that is not directly required on the MAV, but can be used e.g. for testing or visualization.

These required steps that have to be performed as part of a stereo-matching framework are:

1. **Image Acquisition:** Both left and right image have to be acquired. This could be from the camera directly, from the inter-process-communication protocol or from image files on the hard-disk.

2. **Undistortion and Rectification:** The images from the cameras are always distorted. Because the algorithms work with undistorted images like from a pinhole camera, that distortion has to be removed. An undistorted image has straight epipolar lines. To further simplify the matching step, both images are aligned to have a common image plane. This effectively means that the epipolar lines are horizontally aligned and the search in the matching step is reduced to one dimension.

3. **Matching:** The matching algorithm computes the disparity map from the rectified images.

4. **Post-Processing:** The disparity map is cleaned up by removing areas with low textures in the images.

5. **Output:** The filtered disparity map is now used in additional tasks. This could be simply displaying it on a screen, making it available to the inter-process-communication protocol or querying it to perform object avoidance.

The framework is straight forward. Each step is effectively an abstract class that provides the
2. Implementation

Figure 2.1.: Stereo-Matching Framework. All the steps necessary to run stereo-matching on the MAV with the abstract classes and virtual functions that represent each step.

functionality. The derived classes then have to implement that functionality. Figure 2.1 provides a schematic view of the required steps and the classes that represent them.

The following sections give more in-depth information on each step. Each step except the first (Image Acquisition) provides setter methods to set the images to be processed and each step except the last (Output) provides getter methods to access the processed images. Note that the scheme is always the same: set the images to be processed, call the processing method and finally retrieve the images.

2.1. Image Acquisition

Actually getting the images that are then used in the later steps has to be an own independent step from the rest of the matching framework. The reason behind this is that the source of these images is not fixed. The source could be a video file, simple images on the hard-disk or cameras.

2.2. Undistortion and Rectification

Usually, the algorithms and their geometry used in computer vision assume pinhole cameras, as this makes the computation much easier. However, the cameras used in reality always use lenses and therefore always provide distorted images. The cameras used in stereo-matching on the MAV have 4 mm lenses with a diagonal field of view of about 81° and are therefore subject to quite a bit of barrel distortion. Furthermore, tangential distortion might also occur due to an
The goal of the undistortion is to warp the image so it looks like an image taken with a pinhole camera. This means that the image is a perfect rectilinear projection: straight lines in the scene appear as straight lines on the recorded image. This is important because it allows the exploitation of the epipolar constraint: Given a projected point in one image and the rotation and translation between the cameras, that projected point in the other image has to lie along a specific straight line.

In the most simplest terms, stereo-matching algorithms require solving the correspondence problem: given a point viewed by one camera, find that point in the image of the other camera. This usually requires a search in two dimensions. However, if the image is not distorted, the epipolar constraint can be used to narrow the search to the epipolar line. To further narrow the search, rectification of the images is used. It means aligning the images from both cameras as if they had the same image plane, therefore making the epipolar lines match horizontally. Now the search for the corresponding point is limited to the same horizontal line in the other image.

It is important to note that both the undistortion as well as the rectification simply is a warp operation on the image. This means that both warp operations can be combined into one, thus saving computation time. This is usually done with a look up table: it defines for each pixel in the warped image which pixel of the unwarped image should be used. That table is filled once and can then be applied to every recorded frame. It is a two-dimensional array of the same size as the image (640x480 in the case of the PtGrey Firefly MV cameras). Each entry corresponds to a position on the original, distorted and unrectified image. To generate the new undistorted and rectified image, the look up table is used in the following way:

1. Loop through all pixels in the new (still empty) undistorted and rectified image.
2. For each pixel (at position $x_i, y_i$), consult the look up table at $[x_i, y_i]$.
3. The entry found in the look up table at that position correspond to coordinates in the original image. Copy the pixel at those coordinates to the pixel $(x_i, y_i)$ in the new image.

Note that the entries in the look-up-table are usually float and not integer values. They usually point to coordinates between pixels. Therefore, various interpolation methods could be applied, though bilinear interpolation provides already very good results. Nearest neighbor interpolation does not provide such a smooth image, but for stereo-matching it is enough.

### 2.2.1. Calibration

The undistortion and rectification has to be very exact. Otherwise the epipolar lines would not align horizontally and the matching algorithm would fail to find correspondences. The look-up-table is built from the calibration of the camera setup. It depends on the following parameters [4, 6]:

- **Intrinsic camera parameters**: (for each camera)
  - Radial distortion coefficients
  - Tangential distortion coefficient


---

imperfect lens.
2. Implementation

- Focal point
- Principal point

- **Extrinsic camera parameters:** (for the camera setup)
  - Translation between the camera’s coordinate system
  - Rotation between the camera’s coordinate system

These parameters have to be found experimentally using a calibration tool. Bouguets “Camera Calibration Toolbox for Matlab” \(\text{http://www.vision.caltech.edu/bouguetj/calib_doc/}\) proved to be a very efficient and flexible tool to retrieve the calibration data. A set of images of a checkerboard pattern are recorded with the stereo-camera setup and saved. The toolbox then analyzes the images (with the help of the user) and computes the intrinsic parameters for each camera (so that the lines of the checkerboard become straight). Finally the extrinsic parameters can be computed from the intrinsic parameters and the images.

OpenCV is used to first compute the look-up-table from the calibration data and then warp the camera images.

2.3. Matching

For evaluation (see Part III), different local algorithm were implemented as prototypes. Out of those, the simple SAD (Sum of Absolute Difference) algorithm proved to be fast while still giving robust results. Furthermore, SAD looked the most promising for optimization with SSE (Streaming SIMD Extension).

An algorithm similar to SAD was also developed for the GPU. Possible future revisions of the hardware could include a graphics card with dedicated memory, which could completely do the matching. If such a card will ever be used is questionable though, as these kind of graphics chips require much more power than GPUs with shared memory.

2.3.1. Sum of Absolute Differences

SAD gives good matches when enough texture is available. The use of stereo-matching is primarily object avoidance. The assumption has been made that objects that are close to the camera also display texture or visible edges towards the background. Textureless regions are almost always under- or overexposed areas in the image, walls or objects without clear texture. However, at least the silhouette of an object in the foreground should be always visible.

SAD is a simple block matching algorithm. The correspondence problem is approached by comparing blocks of pixels in each image. The dissimilarity is the sum of differences in pixel intensities, therefore the name. The match with the highest similarity wins.

Computing that sum of differences in pixel identities for the block around \((x, y)\) is simple. Let \(f_{i,j}\) be the intensity of the pixel at coordinates \((i,j)\) in the reference image. For the second image, \(g(i,j)\) is used. \(w\) defines the extend of the block in either direction around the center...
2.3. Matching

pixel (the block has therefore width and height $2w + 1$. The variable $d$ is the disparity, defined here as the offset of the block along the epipolar line in the second image.

$$\text{sad}(x, y, d) = \sum_{i=x-w}^{x+w} \sum_{j=y-w}^{y+w} |f_{i,j} - g_{i+d,j}| \quad (2.1)$$

Equation 2.1 gives the SAD as a dissimilarity value of block at $(x, y)$ in the reference and $(x + d, y)$ in the second image. $d$ therefore represents the disparity. Without changing $x$ and $y$, the SAD values for all possible $d$’s are computed and compared. Because the strategy is winner-take-all, the winner is simply the disparity associated with the smallest SAD value. Equation 2.2 describes that strategy for the block around $(x, y)$. Let $D$ be the disparity search range.

$$\text{disparity}(x, y) = \min_{d \in D} (\text{sad}(x + d, y)) \quad (2.2)$$

This is continued for each pixel until the disparity map is completely filled.

2.3.2. Sum of Squared Differences

The dissimilarity value can also be computed as the sum of squared differences (SSD). Instead of taking the absolute of the differences of pixels intensities in the left and right image, the difference can also be squared. Otherwise, SSD is just another block matching algorithm and works exactly like SAD. Equation 2.3 describes the dissimilarity value of a block at $(x, y)$ and disparity $d$.

$$\text{sad}(x, y, d) = \sum_{i=x-w}^{x+w} \sum_{j=y-w}^{y+w} (f_{i,j} - g_{i+d,j})^2 \quad (2.3)$$

2.3.3. Dynamic Programming

The problem with this approach is that the intensity differences for all pixels in the block are computed every time, instead of computed once and then reused. For example, $\text{sad}(x + 1, y, 0)$ could be much easier computed if $\text{sad}(x, y, 0)$ is known! By applying a simple sliding window scheme, data can be reused and the algorithm sped up. This works in the horizontal as well as the vertical direction.

2.3.4. Performance Improvements

The block matching methods here (SAD and SSD) have the flaw that they compute the dissimilarity value every time from scratch. Instead, previously computed dissimilarity values could be reused in future computations. For example, $\text{sad}(x + 1, y, 0)$ could be much easier computed
2. Implementation

Figure 2.2: Horizontal Sliding Window.
A window slides across a horizontal scan line. Only the change has to be computed, here it's the columns indicated by the ‘-’ and ‘+’ symbols.

if \( \text{sad}(x, y, 0) \) is known. By applying a simple sliding window scheme, data can be reused and the algorithm sped up. This works in the horizontal as well as the vertical direction.

Great performance boosts can also be achieved by implementing the core parts of the algorithms with SSE (Streaming SIMD Extension), allowing parallelization of the algorithm using SIMD (Single Instruction, Multiple Data).

**Horizontal sliding window**

By sliding a window across a horizontal scan line, only the change has to be computed, the rest can be reused. Figure 2.2 shows a window that is moved one pixel to the right. The value for the new window is the same as the value for the old window plus the difference. Here the part to the left of the new window that was still part of the old window has to be subtracted, while the new part to the right has to be added. The rest overlaps between the old and new window and can be reused.

For the SAD algorithm, this means that for a fixed disparity \( d \), the sliding window scheme can be used to compute the SAD of the window one pixel to the right:

\[
\text{sad}(x + 1, y, d) = \text{sad}(x, y, d) - \sum_{j=y-w}^{y+w} |f(x - w, j) - g(x + d - w, j)| + \sum_{j=y-w}^{y+w} |f(x + 1 + w, j) - g(x + d + 1 + w, j)|
\]  

(2.4)

Both equation 2.4 and figure 2.2 verify that always only columns have to be added or subtracted. What’s also interesting is that if a column has to be subtracted, it was added some cycles before, so the value for that column was already computed. This means that the values for the columns can be computed before the sliding window is executed, allowing further reuse.

SSE provides a fast way to compute the columns. Using the 128bit registers and SIMD (Single Instruction, Multiple Data) instructions, the values for eight columns can be computed simultaneously. The image is stored as a two-dimensional array of 8bit values in memory. Horizontally
neighboring pixels are therefore also next to each other in memory. Knowing this, it is possible to load 16 pixel values with one instruction into the register. The absolute difference can be build for eight pixels simultaneously:

1. From the reference image, load the pixels starting at position \((x, y)\) into the register.
2. From the second image, load the pixels starting at position \((x + d, y)\) into another register.
3. Convert the data from both registers into 16bit values, leaving only the first eight pixels in each register.
4. Subtract one register from the other, interpreting the data as eight 16bit integers.
5. Take the absolute from the values computed above.

The steps above can be repeated a number of times (at the same horizontal position, but moving one pixel down) until all absolute differences are computed. They can then be summed up to produce the values for the eight columns.

Note that the sliding window requires a fixed offset \(d\) in the second image to work. This simply means that the sliding window scheme has to be completed before the offset is changed. Because the algorithm needs the SAD values for all disparities, the sliding window scheme is repeated for every possible offset, requiring \(O(ImageWidth \cdot DisparityRange)\) space in memory. This is of course a drawback, but the gained computation time makes more than up for it.

**Vertical sliding window**

The algorithm could be run by repeating the above procedure for every horizontal line in the image, basically starting over at each new line. However, the sliding window scheme can also be applied when moving to the next horizontal line.

The columns used during the pass on the next line don’t have to be completely recomputed, they are still almost the same as before. The only difference for each column is the top pixel that is removed and a new pixel at the bottom that is added. Figure 2.3 is a graphical representation of this. Equation 2.5 describes the transition from one line to the next for an arbitrary column with center \((x, y)\).

\[
col(x, y + 1, d) = \col(x, y, d) - |f(x, y - w) - g(x + d, y - w)| + |f(x, y + 1 + w) - g(x + d, y + 1 + w)|
\]  

(2.5)

Using SSE, the subtractions and additions can again be done for eight columns in parallel.

The nice thing is that when both horizontal and vertical window sliding schemes are applied, the runtime of the SAD algorithm is independent of the block/window size. For every disparity, each pixel in the original images is read exactly twice. Once when it is required for the computation of the absolute intensity difference that is to be added to the column, and once when that value is required to subtract it from the column. Theoretically it could be reduced to only
2. Implementation

![Vertical Sliding Window](image)

**Figure 2.3:** Vertical Sliding Window.

After completing a pass on one horizontal line, the algorithm moves to the line below. The computed columns are still mostly the same, except for the pixel above and below.

one read instruction. In practice however, the additional data structure required to temporarily store those values reduced the performance rather than improving it. The re-computation of the values with SSE was simply faster than the cache issues that come with additional data structures.

**Optimizations with SSE**

SSE allows the parallelization of the algorithm by executing a computation on multiple variables with one single instruction.

For example, SSE provides a fast way to implement the horizontal sliding window. The columns described with equation 2.4 and Figure 2.2 can be computed in parallel. Using the 128bit registers and SIMD instructions, the values for eight columns can be computed simultaneously. The image is stored as a two-dimensional array of 8bit values in memory. Horizontally neighboring pixels are therefore also next to each other in memory. Knowing this, it is possible to load 16 pixel values with one instruction into the register. The absolute difference can then be build for eight pixels simultaneously:

1. From the reference image, load the pixels starting at position \((x, y)\) into the register.
2. From the second image, load the pixels starting at position \((x + d, y)\) into another register.
3. Convert the data from both registers into 16bit values, leaving only the first eight pixels in each register.
4. Subtract one register from the other, interpreting the data as eight 16bit integers.
5. Take the absolute from the values computed above.

Figure 2.4 is a graphical representation of the subtraction step. Note that the register is actually just filled with one 128bit value. But depending on the SSE instruction used, it is interpreted as a number of other data types. E.g. in this case as eight 16bit integers.

After all costs (SAD values) for all disparities for a pixel are computed, the winner has to be selected. This is usually done by iterating over all SAD values and keeping track of the smallest. Unfortunately, this requires a lot of conditional tests (if-then-else statements). Often
2.3. Matching

**Figure 2.4.:** Subtraction using SSE.

The 128bit SSE register1 and register2 are filled with eight 16bit integer values. The SSE subtract instruction allows the computation of all eight values in one step.

**Figure 2.5.:** SSE \texttt{minpos()} function.

The function takes a register filled with eight 16bit integers as input and returns the value of the smallest element and its position. Applying this function multiple times in a tree-like fashion, the overall smallest element of a set can be found. The example here with two levels can find the smallest element in a set of up to 64 elements, which is usually enough for an SAD stereo-matching implementation.

these branches break the instruction pipeline of the CPU, leading to performance loss. The new instructions in SSE4 include a \texttt{minpos()} function, which can be used to speed up the search for the smallest element in an array. The function takes eight 16bit integer values and returns the smallest element along with the position in the register. The SAD values are therefore split up into groups of eight. The \texttt{minpos()} function is executed and returns the smallest element and the position for each group. The set to search for the smallest value is reduced by a factor of eight. The \texttt{minpos()} function can now simply be applied again until the set contains only one element. Note that while the winning criteria is the smallest SAD value, the disparity is the wanted information. This means the association between cost and disparity has to be kept.

Figure 2.5 is an example of how the smallest element in a set can be found. The winning SAD value would be 3, and the disparity associated with it would be 10. Note that the position in the top level determines the disparity, as the SAD values were stored in an array according to ascending disparity. Like that, loading the data into the register is simply a matter of passing the address of that array (and the offset within the array) to the SSE load function.
2. Implementation

2.3.5. SAD on the GPU

Todays Graphics Processing Units (GPUs) are quite powerful. They provide programmable vertex and fragment shader which allow special implementations of algorithms to run on the GPU. This technique is known as general-purpose computing on graphics processing unit (GPGPU). Because stereo matching is a form of image processing and works on images, GPUs are ideal to handle stereo matching algorithms.

On the Pixhawk’s MAV an integrated GPU with shared memory is used. It is far less powerful than a dedicated GPU with own memory, but the power consumption is much lower. During the evaluation, it became clear that it was nowhere near fast enough to handle the stereo matching. However, the hope is that a future revision of the MAV’s hardware might include a dedicated GPU that could be used to run the algorithm, thus freeing the CPU to do other tasks.

GPU Pipeline and Programmable Shader

The GPU typically handles only computation for computer graphics. Recently they became so powerful that they can handle computation usually performed on the CPU. This can be achieved with programmable shaders.

Vertex Shader

The vertex shader handles computation for the vertices. When rendering an image, the task of the vertex shader is to transform each vertex’s 3D position to the screen’s 2D position. The vertex shader has access to the following information [10]:

- The 3D position of the vertices
- The texture coordinates of the vertices
- Additional components of the vertices, like normals, colors, lightning, etc.
- Predefined constant values

It is possible to manipulate that information or leave it unchanged and simply pass it onto the next stage in the pipeline, a geometry shader if present, or the rasterizer. Because the vertex shader has no access to texture information, it is not well suited to stereo-matching.

However, the undistortion and rectification could be done using the vertex shader. An image to be undistorted and rectified is applied as a texture to a wire-frame. The vertices of that wire-frame correspond to exactly one pixel of the image. The vertex shader can now move the vertices in 3D space and with that warp the image. Alternatively, the texture position could be changed, so that it reflects the position of the undistorted rectified image, leading to the same effect.
2.3. Matching

Fragment Shader

For the task of stereo-matching, the fragment shader is much better suited. It is also known as pixel shader.

The fragment shader is also known as pixel shader and is dedicated to computing the color and other attributes of each pixel. The rasterizer takes the shapes described as 3D models (vertices, texture coordinates etc) and computes a raster image (pixels) that can be displayed e.g. on a screen. The fragment shader takes the pixels as input and computes it’s color and other attributes. Fragment shader work completely local, e.g. they only compute the values for one pixel and do not have access to other pixel’s attributes. The information available to the fragment shader is [10]:

- The color of the pixel
- Interpolated texel colors
- Texture coordinates of the pixel
- Predefined constant values

The output of the fragment shader is what to show on the screen:

- The color of the pixel
- The depth of the pixel

The depth value of a pixel determines if it is actually shown. The rasterizer generates pixels for each 3D object, so if two objects are behind each other, two pixels are generated. These two pixels would actually be shown on the same pixel on screen, so a test has to be performed to select one and discard the others. Each pixel holds a z-value or depth value. It is usually the depth or z-component of the pixel’s coordinates in 3D space. This would allow the usual depth perception: e.g. a closer object hides the one behind it.

Computation of Pixel Intensity Differences

For standard SAD computation, the pixel intensity differences are usually computed when needed. On the GPU however, it’s easier to compute the intensity differences for the whole image at a given disparity. This can be accomplished in the following way:

1. Load both left and right image as textures in the GPU.
2. Place both textures on the screen, behind each other
3. Use a fragment shader to compute the difference for each pixel. See source code 2.1
4. Render the resulting image and save it as a new texture.
5. Increase the disparity by one by moving the texture of the right image one pixel to the right. Go back to step 3. If enough disparities have been computed, stop.
2. Implementation

Source code 2.1 shows the complete code executed by the shader. It is written in the OpenGL shading language, also known as GLSL or glSlang. It is very similar to C and provides a high-level abstraction to the assembly language or hardware-specific language.

```glsl
uniform sampler2D texL; // Texture from left image
uniform sampler2D texR; // Texture from right image

void main()
{
    // Compute the color value of the left texture at this pixel's location
    vec4 texval0 = texture2D( texL, vec2(gl_TexCoord[0]));

    // And the color value from the right texture
    vec4 texval1 = texture2D( texR, vec2(gl_TexCoord[1]));

    // Set the output color of this pixel to the absolute difference between the textures
    gl_FragColor = abs(texval0 - texval1);
}
```

**Source Code 2.1: Absolute difference shader code**

The source code 2.1 is very simple. The variables declared in lines 1 and 2 allow access to the textures. These variables were actually set in OpenGL so the shader has access to them. The `main()` function contains the executed code. First, the interpolated texel color values are read using the method `texture2D()` (line 7 and 10). Then at line 13 the color of the pixel is set to the absolute difference between the texel color values.

Not that this code is executed for every pixel individually. The execution is parallelized to speed up the execution, since today’s GPUs can have hundreds of shader processors.

One texture for each disparity is computed in this way. The pixels in such a texture hold the pixel intensity differences for that disparity. For example, an SAD value \(sad(x, y, d)\) (see equation 2.1) could simply be computed by just summing up all pixels of the computed texture associated with disparity \(d\) of the block with center \((x, y)\):

\[
sad(x, y, d) = \sum_{i=x-w}^{x+w} \sum_{j=y-w}^{y+w} tex_d(i, j)
\]

The algorithm could be implemented in the way the SAD value is computed in 2.6. However, this would not be efficient. While the pixel intensity differences would only be computed once for each pixel, the sum of those pixels would be recomputed from scratch every time the location of the block is updated. No reuse of the computation is possible, as the fragment shader are executed in parallel. The algorithm has to be completely parallel and each pixel has to be treated completely local. Therefore, also no sliding window implementation is possible.

Another solution to this problem comes with the use of texture sampling.
2.3. Matching

Texture Sampling

Because the fragment shader doesn’t allow access to neighboring pixels, it is not efficiently possible to implement the sliding window protocol. However, texture sampling can be used to sum up pixels. Figure 2.6 shows the result of a sampled texture. The original texture (upper left) is sampled to create a texture with half the width and height of the original. Using bilinear interpolation, each pixel of the sampled image contains the average of four pixels of the original image. This can be continued: sampling the already sampled texture yields a new texture with dimension of 1/4 the width and height of the original.

![Image Sampling](image.png)

**Figure 2.6:** Image Sampling.

Using image sampling with bilinear interpolation on the GPU, sums of pixels can be computed efficiently. From upper left to lower right: original image then the sampled image with size 1/2, 1/4, 1/8, 1/16 and 1/32 of the original size.

Using image sampling with bilinear interpolation, as described in figure 2.7, it is possible to create a convolution-like SAD implementation. In standard SAD, the sum of the pixel intensity differences are uniformly distributed on a block; each pixel is counted exactly once. However, with texture sampling, it is much easier to use a convolution-like approach. This is achieved by exploiting the fact that pixels in the sampled texture are the average of a region of pixels. E.g. a pixel in the texture that was created by sampling an image four times contains the average of a 16x16 pixel block. If pixels from differently sampled textures are used together, it creates a convolution-like window, as described in figure 2.8.

It is important to note that the convolution is not centered as perfectly as shown in figure 2.8. E.g. the texture created by sampling the image three times contains the average of 8x8 pixels, and can be interpreted as a block of size 8x8. The problem is that the whole texture is split up into these blocks, so the location is fixed. It is therefore not possible to get the block at any desired position. This could lead to a convolution as shown in figure ??.

This means that the center of the convolution is not at the pixel that is queried! However, it is
2. Implementation

\[
\frac{a_0 + a_1 + b_0 + b_1}{4} \quad \frac{a_2 + a_3 + b_2 + b_3}{4} \\
\frac{c_0 + c_1 + d_0 + d_1}{4} \quad \frac{c_2 + c_3 + d_2 + d_3}{4}
\]

\[
\frac{(a_0 + a_1 + b_0 + b_1) + (a_2 + a_3 + b_2 + b_3) + (c_0 + c_1 + d_0 + d_1) + (c_2 + c_3 + d_2 + d_3)}{16}
\]

**Figure 2.7:** Sampling with Bilinear Interpolation.
The example shows pixels of an image that is sampled with bilinear interpolation. Each sampling step computes new pixels by taking the average of four original pixels.

**Figure 2.8:** Convolution.
Using differently sampled textures, it is possible to create this kind of convolution. The numbers indicate the size of the region and correspond to the times the texture was sampled, e.g. ‘1’ is the original image and corresponds to a one pixel area, ‘8’ is the texture that was sampled 3 times and corresponds to an area of 8x8 pixels of the original image.

pretty close. And on average, the center is at the center of the convolution.
Using such a convolution has the nice effect that the center is weighted more than the outer regions. In the standard block-matching SAD implementation, every pixel is weighted the same. This has the problem that precision is lost as the window-size is increased, but has the advantage of being perfectly centered.

The implementation on the shader is quite simple. Because the sampled textures are already available, the code in the fragment shader just has to look at all texel’s color values and sum them up. Source code 2.2 describes how this is done.

```cpp
uniform sampler2D tex1; // original absolute difference texture
uniform sampler2D tex2; // texture above sampled to half the size
uniform sampler2D tex4; // sampled to 1/4 the size
uniform sampler2D tex8; // sampled to 1/8 the size
uniform sampler2D tex16; // sampled to 1/16 the size
uniform sampler2D tex32; // sampled to 1/32 the size
uniform float disp; // disparity of this pixel

void main(void)
{

```
Figure 2.9: Resulting Convolution Distribution.
This is an example distribution resulting from summing up the sampled textures around an arbitrary point indicated by the arrow. The numbers indicate the bilinear interpolation.

Source Code 2.2: SAD cost computation

Lines 1 through 6 set the variables to access the textures. Line 7 contains a variable that was set in OpenGL and which reflects the disparity of this pixel. After reading the texel’s color information in lines 12 to 17, these values are summed up to produce the SAD value. Remember that the texel’s color is just a grey-scale value that is interpreted as the dissimilarity value. The scale factors define how the convolution is built up, e.g. the distribution of the convolution in 2.9 would have been created with values 1, 4, 16, 64, 256 and 1024 to get the sum of all the pixels in the sampled images out of the average (e.g. a 8x8 pixel block would have to be multiplied by 64). The values here were found using trial and error, while examining the quality of the resulting disparity map.

The color of the pixel to be shown on screen is set to the disparity associated with it (line 24). The computed SAD value represents the cost, or dissimilarity of the disparity associated with this pixel. The pixel with the lowest SAD cost should eventually be shown on screen, meaning it should have the lowest depth. This is achieved at line 27, where the SAD cost is mapped to
2. Implementation

class the interval from 0 to 1. This makes it possible to just let the standard depth-test handle the
visibility problem. Note that the colors are represented by 4-dimensional vectors, but the sad
variable is a grey-value anyway, so it’s fine to take just the red component, as it was done here.

Algorithm

The previous chapters gave an overview of the possibilities of the GPU to implement a local
stereo-matching algorithm. An algorithm has to combine these steps to provide the resulting
disparity map. This was done with the following steps:

1. Load the textures: The undistorted and rectified images are loaded as textures in the
    GPU.

2. Compute textures for the dissimilarity: For each disparity, do:
   a) Place the texture of the left image so it just fills the screen.
   b) Place the texture of the right image at the same location, but offset it \(d\) pixels to the
      right, where \(d\) is the disparity.
   c) Use the fragment shader to render the absolute difference between the textures. See
      source code 2.1 as a reference
   d) Save the rendered result as a texture.

3. Sample the dissimiliarity-textures: Use bilinear interpolation to sample all textures for
   the dissimilarity multiple times. This yields textures at different sizes or different levels
   of details.

4. Compute the convolution-like SAD:
   a) For each disparity, place a rectangle that fills the screen. Put the original dissimilar-
      ity texture and all sampled textures associated with that disparity onto the rectangle.
   b) For every screen-pixel, the rasterizer now generates one pixel for every dispari-
      ty/rectangle. E.g. \(d\) pixels for every screen-pixel given the \(d\) disparities.
   c) Use the fragment shader to compute the convolution-like sum of the textures asso-
      ciated with a certain disparity. Source code 2.2 is used here:
      i. Sum up the interpolated texel colors of all textures on the rectangle.
      ii. Set the depth of the pixel to the computed sum
      iii. Set the color of the pixel to the disparity.

5. Render the scene: The scene is rendered to produce the resulting disparity map, which
    then can be saved as an image.
2.4. Post-Processing

Filtering provides an efficient solution to increase the quality of the image. The most common filter to use during the post-processing step is a confidence map. The problem with local algorithms like block-matching algorithms is, that they do bad in textureless regions. A local algorithm cannot smooth over areas with no texture, as there is simply no information available. The best thing to do in this scenario is to remove that area from the disparity map altogether.

2.4.1. Confidence map

In order to remove areas with low texture from the disparity map, a confidence map is needed. This map, as the name says, is an estimation of the confidence that the disparity map is correct.

The confidence map in our case should reflect the quality of the texture, so that areas with no texture have bad confidence. This is done by computing the variance in the image. The variance is easy to define for an area or block within the image: it is simply the mean value of the square of the deviation from the mean pixel intensity value.

First, the mean of all pixel intensities inside the block is computed. Note that the block is usually of the same size as the window used for the stereo-matching, since the confidence should somehow reflect the matching-algorithm. Equation 2.7 computes the mean for a square block centered at \((x, y)\) and width and height \(2w + 1\).

\[
mean(x, y) = \frac{1}{(2w + 1)^2} \sum_{i=x-w}^{x+w} \sum_{j=y-w}^{y+w} f(i, j) \tag{2.7}
\]

From this, the variance can be computed as follows:

\[
var(x, y) = \frac{1}{(2w + 1)^2} \sum_{i=x-w}^{x+w} \sum_{j=y-w}^{y+w} (f(i, j) - mean(x, y))^2 \tag{2.8}
\]

Using this kind of confidence value to filter out areas with low texture works well. But the problem is that if there is areas that only display vertical variance are not filtered out. Imagine for example a horizontal line. It would produce good confidence values, but the algorithm would have problem finding a good match, a known issue of block matching algorithms. The solution is to not compute the variance of the whole block, but just the variance a horizontal line going through the center of the block. If this line displays good variance, the algorithm should also be able to find a clear match.

The confidence computed using this variance of a horizontal line does well in practice, although there are two issues with this approach:

- If there is good texture somewhere else than the horizontal line going through the center, the pixel will still be filtered out from the disparity map.
2. Implementation

- If there is only repeating vertical texture, this approach does not filter out the pixel, although the algorithm cannot find a clear match.

Improvements with SSE4

The approach of computing the variance does manage to filter out most problematic regions of the disparity map. Because the computation is really simple, it is also efficient. Using SSE4, the performance can be improved further. While the variance is a sound mathematical concept, the exact computation is not actually required to test if an area contains bad texture. It is enough to simply find the darkest and brightest pixels and compare them. If the difference is large, there should be good texture. On the other hand, if the brightest pixel has almost the same intensity as the darkest pixel, this means that all pixels in the tested region have more or less the same intensity, thus indicating low texture variance.

Using the SSE4 `minpos()` function, it is very easy to find the values for the brightest and darkest pixels. To compute the confidence value for an 8x1 horizontal line of pixels starting at position \((x, y)\) in the image in this way, the following can be executed:

1. Load the eight pixel values starting at \((x, y)\) into the 128bit `register1`.
2. Find the pixel with the lowest intensity by executing `minpos(register1)` and save the result.
3. Compute the complement of each pixel intensity to the largest number possible (e.g. 16bit values: \(0xFFFF - intensity\)) and store them in `register2`.
4. Find the lowest value in `register2` by executing `minpos(register2)`. Now find the original pixel intensity associated with that value (subtract the result from \(0xFFFF\)) to get the pixel intensity of the brightest pixel.
5. Compute the difference between the two found values (brightest - darkest).

The performance improvement over the standard variance computation is great, so that this post-processing step becomes very cheap to perform.

2.4.2. Median Filtering

The confidence map filters out areas with bad texture. This is great, as the remaining parts of the disparity map have a high chance of being correct. This is especially important if that data is then used for object avoidance. However, the confidence map only removes part of the disparity map, it does not actually improve it.

One way of improving the confidence map in the post-processing step is to use a median filter. The idea is that if there are disparity values that are completely different from the disparity values of their neighbors, they are wrong with a high probability. Applying a median filter results in a smoothing over the whole disparity map. This result looks cleaner and is less noisy. A drawback of median filtering is that it actually changes the computed disparity values. If a process later queries the disparity map, it has no guarantee that the value encountered was
2.5. Output

actually the best disparity found. This is probably not a problem for most uses, but in some cases it could be. Providing the raw, unaltered disparity map might be better in some scenarios. If an application requires further filtering, this would still be possible, but not the other way around.

Median filtering was applied by using the functionality provided by OpenCV. The performance of median filtering is sadly not so great. The problem is that the median value has to be found, which implies checking all pixel values inside a block. Already a 5x5 pixel median filter required a few milliseconds to execute. This was not acceptable for the benefit it could bring.

2.5. Output

The last step of the stereo-matching pipeline allows the application of the computed and filtered disparity map. One use could be to display it on a screen. It might also be possible to include object avoidance at this step if it is really simple. For more complex uses, it might be better to separate the process clearly from the stereo-matching. This can be done by making the disparity map available over LCM/IPC.
2. Implementation
Evaluation

The last chapter gave a detailed description of the implementation of the algorithms. Various performance improvements were also described. While the optimizations were described, there were no references to any measurable quality criteria or to runtime performance.

This chapter evaluates these algorithm and allows a side by side comparison. One criterion is of course the quality of the matching. The algorithm should provide a good disparity map that deviates not too much from the ground truth. On the other side, the low computation power requirement is also a key factor that is crucial when a stereo-matching algorithm should run on an MAV. The requirement was made that the stereo-matching should not take longer than 20 ms to compute a disparity map.

3.1. Quality Evaluation

The quality of the algorithm is evaluated with the evaluation framework provided by Middlebury’s Stereo-Vision Website found at vision.middlebury.edu/stereo/. They provide rectified left and right images that can be used for stereo-matching. The algorithm is run on these input images and the disparity map is computed. The computed disparity map can then be uploaded using a web form and is compared to the ground truth. The evaluation then reports the percent of pixels that are not at the correct disparity.

The current version of this evaluation framework is version 2. The older version 1 of the evaluation might have been a better choice to provide information for local stereo-matching methods, as version 2 is geared towards global methods. However, to guarantee the same evaluation in the future, it was important to use the current version, which will hopefully still be supported for some time.
3. Evaluation

The provided image pairs are Tsukuba 3.1(a), Venus 3.1(b), Teddy 3.1(c) and Cones 3.1(d). They are shown in figure 3.1. Both Tsukuba and Venus originated in the old version 1 of the evaluation framework, while Teddy and Cones were added in the current version 2 to provide a real challenge to the top-performing global methods.

![Images](a) Tsukuba.

![Images](b) Venus.

![Images](c) Teddy.

![Images](d) Cones.

**Figure 3.1.: Images used for quality evaluation.**

Matching of the images Tsukuba (a), Venus (b), Teddy (c) and Cones (d) were evaluated. These serve as input for the evaluation framework provided by the Middlebury College that was also used in their paper [16] to compare stereo-matching methods.

The sizes and required disparity search ranges are indicated in table 3.1.

The evaluation should reflect the stereo-matching process on-board the MAV as closely as possible. Therefore, the input images were converted to 8 bit gray-scale images, as this is the type of image that is provided by the cameras.

The algorithm was then run on the four converted input image pairs with the disparity search range indicated in table 3.1. Because the algorithms provide the left disparity map, but the right disparity map is actually required for the evaluation, the input images were first rotated 180°.
3.1. Quality Evaluation

The matching was then executed with these rotated images, left and right images exchanged. The result would be the same if the cameras used to record the images were upside down. The computed disparity map was then rotated back and scaled by the factor indicated. Finally, the resulting scaled disparity maps were uploaded to be processed for evaluation.

The information provided by Middlebury’s Stereo-Evaluation is in the same format for all values: percent of bad pixels. It describes how many percent of pixels of the disparity map are not within a certain threshold of the disparity of the ground truth. The computed values that allow the comparison of algorithms are:

- **Nonocc:** Non-occluded pixels that are visible from both left and right input image. The occluded regions that are visible from only one image do not count toward this value, neither do unknown regions. See figure A.1(a) for a reference.

- **All:** All pixels of the disparity map, except unknown regions. In the two old image pairs Tsukuba and Venus, a border region of 18 (horizontally) and 10 (vertically) are completely excluded from the evaluation. The newer image pairs Teddy and Cones evaluate include these regions. A.1(b) serves as a reference.

- **disc:** The pixels near depth discontinuities. The disparity in these regions is especially difficult to find. See A.1(c) for a reference.

- **Average:** The average percent of bad pixels. This value is a general indicator for the quality of the algorithm. It is the average of the above values of all images.

### 3.1.1. Evaluation of Block-Matching Algorithms

The block matching algorithm SAD is the main candidate to run on the PIXHAWK MAV. As another possibility, SSD (Sum of Squared Differences) was also implemented and evaluated. SAD was evaluated with different block sizes 3x3, 5x5, 5x9 and 9x9 pixels. SSD was evaluated as a reference with block size 9x9 pixels. Table 3.2 shows the detailed results from the evaluation.

The results in table 3.2 are the percent of pixels that are not within an error threshold of 1. To get further information about the quality of the matching, the evaluation was also run with a threshold of 2. This allows the comparison of the exactness of the matching. Table 3.3 shows the same evaluation with an error threshold of 2.

<table>
<thead>
<tr>
<th></th>
<th>Tsukuba</th>
<th>Venus</th>
<th>Teddy</th>
<th>Cones</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Disparity search range</strong></td>
<td>0..15</td>
<td>0..19</td>
<td>0..59</td>
<td>0..59</td>
</tr>
<tr>
<td><strong>Input width (pixels)</strong></td>
<td>384</td>
<td>434</td>
<td>450</td>
<td>450</td>
</tr>
<tr>
<td><strong>Input height (pixels)</strong></td>
<td>288</td>
<td>383</td>
<td>375</td>
<td>375</td>
</tr>
<tr>
<td><strong>Scaling of disparity map</strong></td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

*Table 3.1.: Specifications of Matching Requirements for Middlebury’s Evaluation*
3. Evaluation

<table>
<thead>
<tr>
<th>Evaluation with threshold=2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Tsukuba</strong></td>
</tr>
<tr>
<td>nonocc</td>
</tr>
<tr>
<td>all</td>
</tr>
<tr>
<td>disc</td>
</tr>
<tr>
<td><strong>Venus</strong></td>
</tr>
<tr>
<td>nonocc</td>
</tr>
<tr>
<td>all</td>
</tr>
<tr>
<td>disc</td>
</tr>
<tr>
<td><strong>Teddy</strong></td>
</tr>
<tr>
<td>nonocc</td>
</tr>
<tr>
<td>all</td>
</tr>
<tr>
<td>disc</td>
</tr>
<tr>
<td><strong>Cones</strong></td>
</tr>
<tr>
<td>nonocc</td>
</tr>
<tr>
<td>all</td>
</tr>
<tr>
<td>disc</td>
</tr>
<tr>
<td><strong>Average</strong></td>
</tr>
</tbody>
</table>

Table 3.2.: Evaluation of SAD and SSD with threshold=1: Percent of bad pixels.

For an explanation of what was evaluated for nonocc, all and disc, see the maps in figure A.1.

There are a few interesting observations to make. As expected, the quality of the matching increases with block size. The problem is that if the block size increases, the disparity map loses details. For example, the regions near discontinuities tend to be matched better with smaller block sizes. This makes sense, as larger blocks tend to smooth the image, while smaller blocks keep the discontinuities sharper. On the other side, the areas with low texture present less of a problem with larger blocks. This can be verified by comparing the values for non-occluded regions for different window sizes. It is obvious that a block size of 3x3 or 5x5 pixels is too small.

The rows disc describe only the regions near discontinuities, which are especially hard to match. Because SAD or SSD do not handle discontinuities separately, the quality in those regions is not really a very useful value. The matching might still be correct because only parts of the block is actually occluded and the other is visible from both images. In general however, errors are to be expected.

What is astonishing is that there is quite a difference between the old Tsukuba and Venus images and the new Teddy and Cones images. While the algorithms do pretty well for Tsukuba and Venus, the quality of the matching in the other images is around three times worse. The new
3.1. Quality Evaluation

Table 3.3.: Evaluation of SAD and SSD with threshold=2: Percent of bad pixels.

<table>
<thead>
<tr>
<th></th>
<th>SAD 3x3</th>
<th>SAD 5x5</th>
<th>SAD 5x9</th>
<th>SAD 9x9</th>
<th>SSD 9x9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tsukuba</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>16.1</td>
<td>11.1</td>
<td>9.34</td>
<td>7.60</td>
<td>8.32</td>
</tr>
<tr>
<td>all</td>
<td>17.3</td>
<td>12.9</td>
<td>11.2</td>
<td>9.38</td>
<td>10.1</td>
</tr>
<tr>
<td>disc</td>
<td>16.2</td>
<td>16.4</td>
<td>19.0</td>
<td>20.9</td>
<td>25.7</td>
</tr>
<tr>
<td><strong>Venus</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>23.3</td>
<td>13.7</td>
<td>9.93</td>
<td>7.00</td>
<td>6.93</td>
</tr>
<tr>
<td>all</td>
<td>24.3</td>
<td>15.0</td>
<td>11.3</td>
<td>8.38</td>
<td>8.37</td>
</tr>
<tr>
<td>disc</td>
<td>25.4</td>
<td>23.6</td>
<td>31.3</td>
<td>33.6</td>
<td>39.3</td>
</tr>
<tr>
<td><strong>Teddy</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>32.6</td>
<td>22.4</td>
<td>20.3</td>
<td>20.1</td>
<td>20.5</td>
</tr>
<tr>
<td>all</td>
<td>39.1</td>
<td>30.1</td>
<td>28.3</td>
<td>27.9</td>
<td>28.5</td>
</tr>
<tr>
<td>disc</td>
<td>35.1</td>
<td>26.9</td>
<td>27.8</td>
<td>30.0</td>
<td>33.9</td>
</tr>
<tr>
<td><strong>Cones</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>33.9</td>
<td>20.3</td>
<td>17.3</td>
<td>14.9</td>
<td>14.3</td>
</tr>
<tr>
<td>all</td>
<td>40.4</td>
<td>28.6</td>
<td>26.1</td>
<td>23.7</td>
<td>23.7</td>
</tr>
<tr>
<td>disc</td>
<td>36.3</td>
<td>25.2</td>
<td>28.4</td>
<td>26.3</td>
<td>30.7</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>28.3</td>
<td>20.5</td>
<td>20.0</td>
<td>19.1</td>
<td>20.9</td>
</tr>
</tbody>
</table>

Images were added to provide a challenge to the top-performing algorithms, as the image pairs from the old evaluation were virtually solved. The fact that they are much harder to match is also reflected on Middlebury’s Stereo-Evaluation, where also global methods are doing much worse on the Teddy and Cones images. There might be multiple factors that lead to the worse results:

- **Disparity Search Range:** The old Tsukuba and Venus images have a maximum disparity of 15 and 19 (see table 3.1). The other images however have a maximum disparity of 59. This makes it harder to find the correct disparity, as there is a higher resolution in depth.

- **Evaluated Regions:** The new image pairs do not exclude the border regions from evaluation. The algorithm is expected to somehow extrapolate the disparity information right up to the border of the image. This extrapolation is not performed with this method, so a certain border region is always excluded from matching.

- **Sub-pixel Accuracy:** The old images have integer value ground truth. The newer Teddy and Cones ground truth however have sub-pixel accuracy. The implementations of SAD and SSD only compute integer values for the disparities. This means that the algorithm has to be more exact for the Teddy and Cones images to reach the same score. E.g. if the error threshold of 1 disparity is used: the ground truth shows the disparity for a pixel
3. Evaluation

at say 10.6, then accepted correct disparities range from 9.6 to 11.6. When only using integer values, the only two accepted disparities are 10 and 11. But if the ground truth is also in integer, three disparities would be accepted. E.g. if the ground truth shows the disparity for a pixel at 11, then the accepted disparities (using an error threshold of 1) would be 10, 11 and 12. This favors algorithms that compute only integer values for the Tsukuba and Venus images.

3.1.2. Evaluation of Dynamic Programming Methods

Dynamic Programming was considered as a better strategy than simple winner-take-all. The disparity maps were evaluated in the same way as SAD and SSD. Two implementations were tested. The first used simply the absolute intensity difference of a single pixel as dissimilarity value. The second used a 3 by 3 pixel SAD block. The algorithms are indicated here as DP1x1 and DP3x3. Table 3.4 shows the evaluation results for an error threshold of 1.

<table>
<thead>
<tr>
<th>Evaluation Dynamic Programming, threshold = 1</th>
<th>DP1x1</th>
<th>DP3x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tsukuba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>27.4</td>
<td>20.2</td>
</tr>
<tr>
<td>all</td>
<td>29.1</td>
<td>21.9</td>
</tr>
<tr>
<td>disc</td>
<td>33.9</td>
<td>30.2</td>
</tr>
<tr>
<td>Venus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>58.8</td>
<td>26.7</td>
</tr>
<tr>
<td>all</td>
<td>59.5</td>
<td>27.9</td>
</tr>
<tr>
<td>disc</td>
<td>67.2</td>
<td>48.8</td>
</tr>
<tr>
<td>Teddy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>49.5</td>
<td>43.9</td>
</tr>
<tr>
<td>all</td>
<td>54.8</td>
<td>49.6</td>
</tr>
<tr>
<td>disc</td>
<td>56.5</td>
<td>41.2</td>
</tr>
<tr>
<td>Cones</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>45.6</td>
<td>24.9</td>
</tr>
<tr>
<td>all</td>
<td>51.8</td>
<td>33.3</td>
</tr>
<tr>
<td>disc</td>
<td>55.2</td>
<td>38.6</td>
</tr>
<tr>
<td>Average</td>
<td>49.1</td>
<td>33.9</td>
</tr>
</tbody>
</table>

Table 3.4: Evaluation of Dynamic Programming with threshold=1: Percent of bad pixels. See figures A.1 for an explanation of the evaluated regions.

The results are quite bad. The difficulty with dynamic programming is always finding the right costs. In the case of stereo-matching, one cost is simply the dissimilarity value. The other is a cost associated with a jump in disparity, because the disparity map should be smooth. The best
value for this cost depends on the images used. The cost for the disparity-jump was set so that the matching was best for the scenes recorded with the cameras. By finding the right value for the four image pairs, the evaluation results could definitely be improved. However that is not the goal of the evaluation.

Using dynamic programming tends to smooth the resulting disparity map. This is nice because the textureless regions are filled out. Disparity discontinuities are preserved most of the time. On the other hand, the smoothing might be too much, so that small jumps in disparity are ignored and just smoothed over. This is for example the reason why the Tsukuba disparity map computed with dynamic programming has the erroneous horizontal lines.

The evaluation requires constant parameters for all four image pairs. While the disparity map computed from the Cones image pair seems to have less errors, the Tsukuba disparity map has a lot.

But even for the Cones images, the result from the evaluation is not really good. To evaluate the algorithm further, the threshold was increased to 2. Table 3.5 shows the results from that evaluation.

<table>
<thead>
<tr>
<th>Evaluation Dynamic Programming, threshold = 2</th>
<th>DP1x1</th>
<th>DP3x3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tsukuba</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>17.0</td>
<td>13.5</td>
</tr>
<tr>
<td>all</td>
<td>18.8</td>
<td>17.0</td>
</tr>
<tr>
<td>disc</td>
<td>25.4</td>
<td>25.0</td>
</tr>
<tr>
<td><strong>Venus</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>21.7</td>
<td>11.6</td>
</tr>
<tr>
<td>all</td>
<td>23.0</td>
<td>13.0</td>
</tr>
<tr>
<td>disc</td>
<td>43.0</td>
<td>36.2</td>
</tr>
<tr>
<td><strong>Teddy</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>31.2</td>
<td>27.0</td>
</tr>
<tr>
<td>all</td>
<td>38.3</td>
<td>34.4</td>
</tr>
<tr>
<td>disc</td>
<td>37.3</td>
<td>28.3</td>
</tr>
<tr>
<td><strong>Cones</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>22.8</td>
<td>16.9</td>
</tr>
<tr>
<td>all</td>
<td>31.5</td>
<td>26.0</td>
</tr>
<tr>
<td>disc</td>
<td>37.7</td>
<td>30.2</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>29.0</td>
<td>23.4</td>
</tr>
</tbody>
</table>

*Table 3.5:* Evaluation of Dynamic Programming with threshold=2: Percent of bad pixels.

The evaluation turns out much better. It seems that dynamic programming smoothes too much, which might actually look nicer, but decreases the exactness of the matching too much.
What is also interesting is that the results for DP1x1 and DP 3x3 are not very different. An SAD implementation with a block size of only one pixel is completely useless, but the dynamic programming optimization over the scan-line is enough to provide a usable disparity map.

### 3.1.3. Evaluation of the GPU Implementation

The algorithm implemented for the GPU was evaluated in the same way as the block-matching algorithms. But the disparity maps provided by the GPU implementation are actually the right disparity maps. Therefore, no conversion from left to right disparity map was required. This probably improved the results a little bit.

Another difference is that the algorithm was implemented only for image sizes of 640 by 480 pixels. The images were therefore first converted to those resolutions, then matched and converted back to the original sizes. Because the width of images used for the evaluation were all smaller than 640 pixels, this provided a sub-pixel accuracy.

<table>
<thead>
<tr>
<th></th>
<th>GPU Implementation</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>threshold=1</td>
<td>threshold=2</td>
<td></td>
</tr>
<tr>
<td><strong>Tsukuba</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>18.5</td>
<td>5.60</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>20.2</td>
<td>7.42</td>
<td></td>
</tr>
<tr>
<td>disc</td>
<td>30.0</td>
<td>18.4</td>
<td></td>
</tr>
<tr>
<td><strong>Venus</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>16.1</td>
<td>5.57</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>17.5</td>
<td>8.79</td>
<td></td>
</tr>
<tr>
<td>disc</td>
<td>42.9</td>
<td>27.4</td>
<td></td>
</tr>
<tr>
<td><strong>Teddy</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>35.5</td>
<td>19.8</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>41.9</td>
<td>27.4</td>
<td></td>
</tr>
<tr>
<td>disc</td>
<td>48.4</td>
<td>29.1</td>
<td></td>
</tr>
<tr>
<td><strong>Cones</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonocc</td>
<td>30.9</td>
<td>13.4</td>
<td></td>
</tr>
<tr>
<td>all</td>
<td>38.2</td>
<td>22.0</td>
<td></td>
</tr>
<tr>
<td>disc</td>
<td>42.3</td>
<td>22.9</td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>31.9</td>
<td>17.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: Evaluation of the GPU Implementation: Percent of bad pixels. See figures A.1 for an explanation of the evaluated regions.

The interesting thing to note is that the error threshold has a very big influence on the quality. For the Tsukuba and Venus images, more than 10% of the pixels are more than one, but less than two disparities away from the ground truth. Using an error threshold of 2 gives quite good
results, while an error threshold of 1 gives rather bad results. A factor might be the exact procedure that was used to compute the disparity maps. Because the width of the input images was less than what was actually used for matching, sub-pixel accuracy was achieved. The Tsukuba and Venus ground truth favor disparity maps with integer accuracy. Another factor might be that the implementation on the GPU tends to smooth over large areas. This is nice because there are less regions with completely wrong disparity values. But the smoothing reduces the exactness of the matching.

3.2. Performance Evaluation

During the requirements analysis, it became clear that stereo-matching should be efficient and not eat up too much processing power. 20 ms to completely compute a disparity map was needed. This includes the undistortion, rectification and post-processing required in a complete stereo-matching system. The run-times were all recorded on the COM Express board with the Intel Core 2 Duo processor. One exception is the GPU implementation. The run-times on a dedicated and integrated GPU needed to be compared.

3.2.1. Undistortion and Rectification

Undistortion is a simple warping of the image obtained from the camera, as is the rectification. Warping an image is usually done with a Look-Up-Table (LUT). It makes sense to combine both LUTs from undistortion and rectification into one, yielding the same result as if they were applied separately. Because the image size used in the matching is actually half the width and half the height of the original images, they have to be resized. Instead of first executing the undistortion and rectification and then resizing the result, everything can be accomplished with a single LUT. This improves the speed quite a bit. However, it has the disadvantage that no undistorted, rectified image with full resolution is computed along the way. If other processes use the images from the same cameras, a full-resolution undistorted (and probably rectified) image has to be computed anyway.

The runtimes are indicated in table 3.7. All timing was recorded in milliseconds.

<table>
<thead>
<tr>
<th>Undistortion and Rectification (ms)</th>
<th>interpolation: nearest neighbor</th>
<th>linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>at full resolution (640x480)</td>
<td>5.88</td>
<td>15.93</td>
</tr>
<tr>
<td>including resize to 320x240</td>
<td>1.64</td>
<td>3.85</td>
</tr>
</tbody>
</table>

*Table 3.7.: Performance of Undistortion and Rectification. Note that this is the value for both images.*
3. Evaluation

3.2.2. Matching

The actual stereo-matching is of course the most important part of the system and requires also the most time. The largest effort was invested in this step and to improve the performance to the point where it could really be used on the MAV.

The table 3.8 shows the runtimes for the matching. SAD with window-size 9x9 was optimized with SSE4 to a SAD 8x8 version.

The performance of the GPU implementation was measured on the COM Express board as well. But because the integrated GPU was extremely slow, a reference using the dedicated GPU of a desktop computer with similar specifications was needed. On all algorithms, the disparity search range was 30. The implementation on the GPU took the original 640 by 480 images as input, while the SAD algorithm accepted only half the resolution (320x240).

<table>
<thead>
<tr>
<th>method:</th>
<th>SAD</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>9x9 block, C++</td>
<td>26.12</td>
<td>12.10</td>
</tr>
<tr>
<td>8x8 block, SSE4</td>
<td>11.04</td>
<td>1100 + CPU load</td>
</tr>
</tbody>
</table>

Table 3.8: Performance of Matching Algorithms.
All times are in milliseconds. The CPU load indicated was an increase on both CPU cores from a load of about 8% to around 45%.

The optimizations with SSE4 decreased the runtime of the SAD matching to less than half of the original time. This shows the great improvements in performance that can be achieved by optimizing key code fragments with SSE.

Using the integrated graphics card, the GPU implementation requires roughly the same time to compute the disparity map. But the provided disparity map has double the resolution, that’s four times the amount of pixels to process. This really shows the raw processing power of the GPU.

Unfortunately, the same algorithm on the integrated GPU on the COM Express board wasn’t really usable. With a runtime of over a second, the performance of the integrated GPU was about a hundred times worse than the performance of the dedicated GPU! The main problem is the shared memory, which is of course much slower than any dedicated memory. Furthermore the CPU load increased drastically. From the idle state with a load of about 8%, both cores jumped to about 45% when the algorithm was run. It could be that some OpenGL commands are actually executed on the CPU. The only part requiring a lot of computations that is executed in OpenGL is the texture sampling. The rest is done with the shaders. It seems that texture sampling should be implemented in hardware on a GPU, but if it isn’t, it would explain the bad performance and the increase in CPU load. Another possible problem might be that the memory I/O is actually managed by the CPU. If the shared memory is managed badly, it could explain some rise in CPU load.

No matter what the exact cause for the rise in the CPU load is, it makes the algorithm completely unusable on the MAV. Even if the runtime of the algorithm could somehow be brought down to a manageable range, the CPU would still be under too much load. The whole idea of performing the stereo-matching on the GPU was actually that the CPU would be freed for other processes.
3.2.3. Post-Processing

The filtering of the disparity map with a confidence map increases robustness, while only requiring minimal computation time. Experimentation with different methods have shown that taking the variance of only a horizontal line yielded the best results. The length corresponds to the width of the window that was used for matching, in the case of the SSE4 SAD version, this was eight pixels wide. The table 3.9 shows the time required to compute the confidence map. One version computes the variance using standard C++, and the optimized version computes the absolute difference between darkest and brightest pixel using also SSE4 instructions.

After the confidence map is created, the areas with low confidence should be removed from the disparity map. This is indicated in the filter column.

<table>
<thead>
<tr>
<th>Confidence Map (ms)</th>
<th>variance</th>
<th>difference with SSE4</th>
<th>filter (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.40</td>
<td>0.74</td>
<td>0.29</td>
</tr>
</tbody>
</table>

Table 3.9: Performance of Confidence Map Computation.

Run-time to compute the confidence map using the variance of a horizontal line, or the difference between darkest and brightest pixel on that line. The filtering step of actually removing the bad textures is indicated separately and independent of the confidence map.

The optimization with SSE is again really worth it, as the results are almost identical, but the performance of the SSE version is over 4.5 times better.
3. Evaluation
Conclusion and Outlook

This report has shown how a stereo-matching system can be implemented for an MAV. Different algorithms were tested, both on the GPU and on the CPU. The goal was to build a complete stereo-matching system with low performance requirements. Because other mission critical processes have to run in parallel, the required runtime to produce a disparity map should be below 20% load of one CPU core. Or not more than 20 ms given the estimated frequency of 10 Hz.

An implementation on the GPU was developed and tested. Sadly, the integrated GPU used on the hardware of the MAV was not fast enough to handle the matching. Especially because the matching on the GPU also increased the CPU load, it makes no sense to use that implementation for the current hardware. Although a future hardware revision with a better GPU might change this.

However, using a fast local algorithm and optimizations with Intel’s SSE, it is possible to compute dense disparity maps in under 20 ms. The evaluation has shown that the SAD block-matching algorithm can provide reasonably robust disparity maps. The performance evaluation proved that it is possible to build a stereo-matching system which requires very little processing power. If linear interpolation is used for the undistortion, rectification and resizing step, the whole matching process requires less than 16 ms to compute a filtered disparity map. If no interpolation is used and simply the nearest neighboring pixels are taken, that time is even reduced to below 14 ms.

A good framework to synchronize the cameras is required, since it is crucial that both images are taken at exactly the same time. Otherwise the matching gives wrong results. Because also other processes use the images as input for their algorithms, that framework should make the images available to any part of the system that is interested. This can easily be accomplished with inter process communication.
4. Conclusion and Outlook

Unfortunately, during this master thesis, the new MAV was in development and not yet ready to fly autonomously. Therefore, no evaluation of an object avoidance implementation was possible. It is still not clear whether 10 Hz is enough for object avoidance, or if the frequency has to be higher. This will have to be evaluated with real tests with a flying helicopter.
Images used for the Evaluation

All computed disparity maps used for the evaluation are provided in this appendix. It allows the comparison of the different algorithms. As a reference, the ground truth used for the evaluation is provided as well. For each image pair, all computed disparity maps are given. Additionally, table A.1 shows what regions were actually evaluated, as an example for the Cones image pair.
A. Images used for the Evaluation

(a) Non-occluded: Only white regions are evaluated, black regions are unknown or half-occluded.

(b) All: Only white regions are evaluated, black regions are unknown.

(c) Near Discontinuities: Only white regions are evaluated, black and grey regions are not evaluated

(d) Ground Truth

Figure A.1.: Evaluated Regions.
Black and grey indicates regions that are not evaluated and therefore do not count towards the error statistics. Only white regions are evaluated.
Figure A.2: Tsukuba disparity maps used for evaluation.
A. Images used for the Evaluation

Figure A.3.: Venus disparity maps used for evaluation.
Figure A.4: Teddy disparity maps used for evaluation.
A. Images used for the Evaluation

Figure A.5.: Cones disparity maps used for evaluation.
Bibliography


Bibliography


