GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Fakultät für Physik

Master's Thesis

Ein in Echtzeit arbeitender, kantenerhaltender Glättungsfilter für Bildverarbeitung auf paralleler Hardware

A real-time, edge-preserving smoothing filter for image processing on parallel hardware

prepared by

Simon Martin Reich

from Iserlohn

at the Third Institute of Physics

Thesis Period:	23rd July 2012 until 30th September 2012
Supervisors:	Alexey Abramov Jeremie Papon Babette Dellen
First Referee:	Prof. Dr. Florentin Wörgötter
Second Referee:	Prof. Dr. Reiner Kree

Abstract

Humans meet a large variety of different objects and materials which they can identify without any problems. In computer vision, however, the extraction of objects in an image continues to be a challenging task due to the great diversity in natural scenes. Also, objects often have a textured surface which leads to significant problems when the segmentation is based on color only. An object's texture is partitioned into several segments and the resulting segments do not coincide with the object. To date, there has not been an adequate algorithm to divide images containing different natural and heavily textured scenes into meaningful regions. In the case of robotic application this leads to noise in the perception-action loop. In this work, a filter is proposed which detects texture and smoothes the corresponding area with a uniform color while preserving edges. These areas are more accessible to color based image segmentation. The filter performs in real-time on parallel hardware and can be applied as an independent preprocessing step to segmentation. The proposed texture filter has been validated by comparison with state-of-thje-art smoothing filters and segmentation algorithms. The results show that our filter outperforms the other filters and that segmentation performance is significantly improved. As the filter performs in real-time on a modern GPU, the filter is accessible to the perception-action loop of robots. Object segmentation and classification of robotic agents may be improved leading to less noise in the perception-action loop and a more stable behavior of the agent.

Keywords: Computer Vision, Texture Filter, Image Segmentation, GPU, Real-time Processing, Edge-preserving Filter, Classification

Acknowledgements

This thesis would not have been possible without the support of many friends and colleagues. First of all I would like to thank my supervisor Prof. Florentin Wörgötter. I trust his assistance to have steered me along the right track, while allowing me to take the research in my own direction and to indulge ideas that have interested me since a long time. I particularly would like to thank him for the possibility of the research visit in Manchester (England), for the opportunity to publish my research results, and especially for offering me a Ph.D. position in his computer vision group. I also thank his wife and secretary Ursula as she was always very helpful and friendly. I also would like to express my gratitude to Prof. Reiner Kree who volunteered as a second referee for my thesis and took a deep interest in my work.

My special thanks go to Dr. Alexey Abramov, Dr. Eren Erdal Aksoy, Dr. Babette Dellen, and Jeremie Papon who made an outstanding contribution to this work. Also I thank all members of our very friendly, creative, and talkative group: Mohamad Javad Aein, Dr. Alejandro Agostini, Martin Biehl, Jan-Matthias Braun, Dr. Markus Butz, Sakyasingha Dasgupta, Faramarz Faghihi, Michael Fauth, Dennis Goldschmidt, Dr. Frank Hesse, Dr. Christoph Kolodziejski, Dr. Tomas Kulvicius, Dr. Guoliang Liu, Dr. Poramate Manoonpong, Dr. Irene Markelic Chanwit Musika, Timo Nachstedt, Dr. KeJun Ning, Vishal Patel, Dr. Karl Pauwels, Markus Schöler, Harm-Friedrich Steinmetz, Dr. Minija Tamosiunaite, Christian Tetzlaff, Birk Urmersbach, Thomas Wanschik, Xiaofeng Xiong, and Steffen Zenker.

This thesis is dedicated to my parents, Teresa and Michael, and to the rest of my loving family, who invested so much into my education.

Contents

Nomenclature

Variable	Meaning
$\hat{\boldsymbol{f}}\left(\boldsymbol{x},\boldsymbol{y}\right)$	Input Image in the continuous domain, bold letters mark a vector in the space domain, bold letters marked with a hat $\hat{}$ refer to an RGB-vector
$\hat{\boldsymbol{h}}\left(\boldsymbol{x},\boldsymbol{y}\right)$	Output Image in the continuous domain
Φ	Input Image in the discrete domain
$\hat{\boldsymbol{\varphi}}_{i,j}$	RGB color vector entry in Φ at position i,j
Θ	Output Image in the discrete domain
$\hat{\boldsymbol{\theta}}_{i,j}$	RGB color vector entry in θ at position i,j
Ψ	Subwindow in the discrete domain
$\hat{\boldsymbol{\psi}}_{i,j}$	RGB color vector entry in Φ at position i,j
$u,\ v$	Size of image
$k,\ l$	Size of subwindow
τ	Threshold for proposed texture filter

1. Introduction

Approximately 540 million years before you started reading this thesis, something interesting happened. Although our planet has existed for over 4.5 billion years and was inhabited for 3.5 billion years by living organisms, only 540 million years ago a rapid change appeared in the existence of major animal phyla. This change is called the Cambrian explosion (Cowen, 1991). Before this explosion most organisms were simple and composed of individual cells occasionally organized as colonies. During the next 70 to 80 million years a major diversification of organisms including animals, phytoplankton, and calcimicrobes took place, resulting in complex, multicellular life (Zhuravlev and Riding, 2000). Even Charles Darwin was surprised by this rapid appearance of fossils and saw it as one of the main objections against his theory "On the Origin of Species by Natural Selection" (Darwin, 1859). Later research, however, suggests that evolution does not happen in a gradual change, but consists more or less of bursts of species diversification (Venditti et al., 2009). Interestingly the oldest fossils with eyes are dated back to the Cambrian explosion. Consider the huge advantage of a species of predators with some form of sight in a world of blind prey. The survival of the prey species would depend on the adoption of new senses for defence. Still, the development of a new visual sensor is not enough. The brain must allow the species to comprehend the sensory input. Modern species are able to navigate through the world with a remarkable degree of understanding. This means that nature provides a proof of concept for systems of sensors and brains, which can see, understand, and learn.

Nature shows us a proof of concept and human beings have been wondering for centuries now, if we can build an artificial system. Even today, we fall short of building such a system. Nevertheless, with the development of the modern computer during the last sixty years, remarkable advancement has been achieved. Advances in physics, computer sciences, and engineering enable the construction of robots, which act autonomous in a very limited way. Still, learning and understanding continue to be challenging tasks for artificial systems.

1.1. The Rise of Robots

The word “robot” was invented in 1941 by the famous science fiction writer Isaac Asimov (Asimov, 1950). In many of his stories he predicted that today’s world would be populated more and more by robots. In contrast to his science fiction books, which feature mostly autonomous humanoid robots, today’s robots are mostly embedded systems assisting human beings. Understanding and learning is still a huge challenge for robots. Robots that learn are in most cases designed for a very limited range of tasks, e.g. in lane or car tracking, automatic analysis of aerial images, or object recognition. Especially the last task, object recognition, has become more and more important in recent years, as increasing computational capability makes real-time object recognition and classification feasible. While there is a broad range of near-field sensors (e.g. switches, capacitive or inductive sensors, Hall effect sensors, ultrasonic or optical techniques), far-field perception relies mostly on optical techniques, e.g. lasers or cameras. Optical systems may include more information than the optical wavelength human beings are able to perceive, for example infrared information or depth information acquired using time-of-flight. Nevertheless, the light spectrum visible to the human eye is de facto the most commonly used one by far-field-sensors.

Using the given sensory input we want the robot to learn about a specific goal. Naturally, when we talk about learning, we want the robot to learn how to pursue this goal. We will call this goal-directed behaviour, which will result in most cases in object manipulation tasks. Dividing the system into two parts, the environment and the agent, the agent perceives information about the environment through its eyes. This information is interpreted by the brain, which will control the arm to fulfill the manipulation task. This circle is called the perception-action loop. It may be seen in figure 1.1 on the left side. For a robot agent we have the same tasks performed by robot counterparts. A variety of sensors is used instead of the eye, a vision system instead of the human vision system, and the locomotion and action system of the robot performs the object manipulation task instead of the human arm. This may be seen on the right side of figure 1.1.

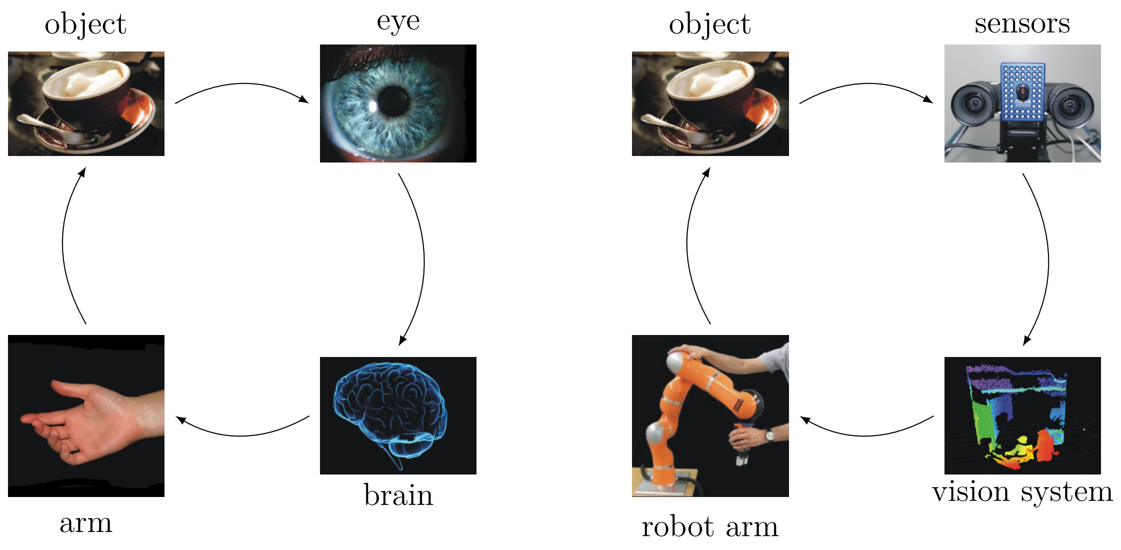


Figure 1.1.: Visual representation of the perception-action loop. On the left side the perception-action loop of a human being is shown, the right side shows the counterparts in a robotic system.

1.2. Robotic Vision

The current study is focused on the vision system in the perception-action loop. The vision system has various input streams. Most commonly, a color image and depth information are used. It will then extract objects, or their parts, and analyze different aspects, for example location, object relations, shape, or texture. The first step, object extraction, is called “image segmentation”. The second step strongly correlates with the goal of the robot and may be called “planning”. In segmentation, the input image is partitioned into meaningful segments. When one segment coincides with one object a perfect match is achieved. But even state-of-the-art segmentation algorithms cannot produce a perfect match in every scene. This is due to a great diversity of light conditions and an even greater choice of materials with different textures. In the Merriam-Webster dictionary “texture” is defined as (Merriam-Webster, 2008)

> [The] visual or tactile surface characteristics and appearance of something[.]

In this work the word “texture” will be used in a similar sense, but more closely defined. As we want the filter to work as generally as possible, we do not handle

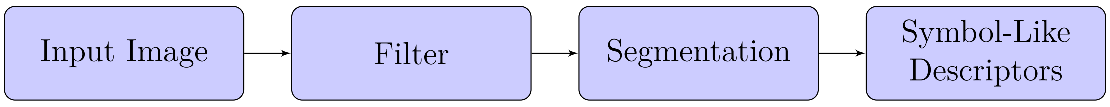


Figure 1.2.: Flowchart of the filter process.

any information about depth or surface structure. We see texture as color changes on an object and therefore as an intrinsic feature of the object. Also we assume the pattern of the texture to be small relative to the size of the object and we do not make assumptions about any texture patterns, even though they are repetitive in many cases. The texture of an object is seen as noise on the object. In contrast to texture we distinguish image noise. This noise is not an intrinsic feature of an object, but of the whole image. An image might contain noise due to sensor noise, post processing, or compression. As noise is also a small scaled color variation in the image, the filter should identify it as texture and remove it.

In this work a novel texture filter is proposed, which detects textures and fills the corresponding area with a mean color. By doing this, textured areas are not partitioned into multiple segments and the match between segments and real objects is significantly improved. Based on an enhanced segmentation, the planning phase can be executed with greater certainty. The process chain can be seen in figure 1.2. This work is focused on color based input streams and neglects any additional information the image might contain. The filter is therefore accessible to a wide range of applications, including low-cost robots, where depth information is often not available.

When developing a new filter the question of integration into the robot system remains. For this work a modular computer vision system called “Oculus” was used (Papon et al., 2012). Oculus provides a plug-in shell and currently can handle a variety of input streams, such as mono and stereo images, or depth information. It can calculate optical flow (Pauwels et al., 2010), compute disparity (Pauwels et al., 2010), segmentation and tracking (Abramov et al., 2010), dense disparity estimation, and generate semantic graphs and event chains (Aksoy et al., 2010). Oculus also provides access to parallelization using graphics processing units (GPUs) and multithreading on central processing units (CPUs).

Motivated by increasing computational capability, numerous segmentation algorithms for video streams have been introduced over the last two decades. The purpose of these algorithms is to segment the image, label the segments and use a fixed label for the same segment throughout the stream. Due to the large variety of textures and materials in our natural environment, this continues to be a challenging problem. When grouping image areas into segments, a similarity criterion needs to be defined. However, similarities may exist on different spatial scales, for example between adjacent pixels or groups of pixels, as is the case for texture. Segmentation algorithms thus need to take into account similarities occurring at these different scales, which can be computationally expensive. It is our goal to remove texture in real-time from an object, while preserving the object borders. The filter may be applied as a preprocessing step for any segmentation algorithm. We show that an implementation on parallel hardware can process images in real-time and therefore makes the algorithm accessible to the perception-action-loop of robots.

Parts of this work were submitted as a conference paper to the International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications 2013 (Reich et al., 2013). For further information please refer to (Visapp, 2012).

1.3. Thesis Structure

This work is organized as follows. First, a short introduction into image filtering and a brief overview of filtering techniques is given in section 2.1. As this is a fast expanding field, current methods are introduced, but are not explained in detail. Afterwards a description of the proposed texture filter in the continuous and discrete domain is given and an implementation on parallel hardware is shown in section 2.2. Finally, in section 2.3, we give an overview of segmentation algorithms.

Chapter 3 contains the experimental results and the time performance of the filter. Visual examples as well as comparisons to other recent methods are given. The filter is combined with several current real-time segmentation algorithms and an open image segmentation benchmark is used for a quantitative evaluation.

In chapter 4 the results are discussed. Chapter 5 concludes as well as discusses avenues for future work.

2. Methods

Texture removal consists of two main steps; first we want to detect texture and then subsequently we want to fill the detected area with a color defined by the surrounding area. Replacing pixel values with an averaged color allows us to make textured areas more uniform. These processes should base upon a very small set of user based variables that are easily optimizable. The image we will have to be analyzed in the spatial domain as well as in the color domain. In order to distinguish between these two domains, we will use the following notation. Vectors in the spatial domain will be printed bold in the form $\boldsymbol{\varphi} = (x\ y)^T$. The vectors in the color domain will be printed bold and denoted using a caret-symbol: $\hat{\boldsymbol{\psi}} = (r\ g\ b)^T$. Please note that for the color domain vector multiplication does not hold. Matrices will be marked using capital letters, such as A. Discrete values will be shown using Greek letters, the continuous domain will be denoted using Latin letters.

In this work there are various images shown. All images, as not otherwise directly noted, are taken from the Berkeley Segmentation Dataset and Benchmark (Martin et al., 2001). This dataset offers a wide variety of different indoor and outdoor scenes, contains ground truth information for image segmentation[1] and is widely used in the field for comparison purposes.

During the past two decades many types of filters for homogenizing textured areas have been proposed. The distinction between textured areas and non-textured areas is made using thresholds. These thresholds can be either learned using a training set of images, as in support vector machines (Yang et al., 2010) and neural networks (Muneyasu et al., 1995), or the threshold may be computed from the surrounding pixel values, as in (Du et al., 2011). Lev et al. (1977) identified similar pixels by

[1]Ground truth information refers to segmentations done by human beings and are assumed to be ideal. Machine segmentations are checked against the ground truth information, see also section 2.3.

detecting edges and iteratively replacing the intensity of the pixel by the mean of all pixels in a small environment.

2.1. Smoothing Filters

Texture filters are in most cases smoothing filters. The term “smoothing filter” is generally applied to all filters whose main application is to smooth the input data. “Smoothing” refers to the process of capturing the most prominent features of data, while leaving out the rest. The most commonly used smoothing filter is the Gaussian blurring filter which is described in the next section. Texture filters however are dedicated to the task of removing texture. This usually involves a two phased system. During the first phase texture is detected and during the second phase it is removed. The process of removal is in most cases based on smoothing.

2.1.1. Gaussian Blurring Filter

The most common smoothing filter in signal and image processing is the Gaussian blurring filter. A Gaussian filter is defined as having an impulse response, which is a Gaussian function. Since the Fourier transform of a Gaussian is another Gaussian, it acts as a low-pass filter with the high frequency domain of the input field filtered out. Therefore noise, as well as any detailed structure, is removed. For two dimensional input fields the kernel function $G(\boldsymbol{x})$ is defined as

$$G(\boldsymbol{x}) \quad = \quad \frac{1}{2\pi\sigma^2} e^{-\frac{\boldsymbol{x}^2}{2\sigma^2}}, \tag{2.1}$$

where $\boldsymbol{x} = (x\ y)^T$. The input signal $f(\boldsymbol{x})$ is modified by convolution with the kernel as follows

$$h(\boldsymbol{x}) \quad = \quad k_G^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\boldsymbol{\zeta}) \cdot G(\boldsymbol{x} - \boldsymbol{\zeta}) \mathrm{d}\boldsymbol{\zeta}, \tag{2.2}$$

where $h(\boldsymbol{x})$ is the filtered output signal. The normalization k_G is defined as

$$k_G(\boldsymbol{x}) \quad = \quad \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} G(\boldsymbol{x} - \boldsymbol{\zeta}) \mathrm{d}\boldsymbol{\zeta}. \tag{2.3}$$

While a digital image is composed of three color channels, namely red (R), green (G) and blue (B), the input signal f corresponds to one of these input channels.

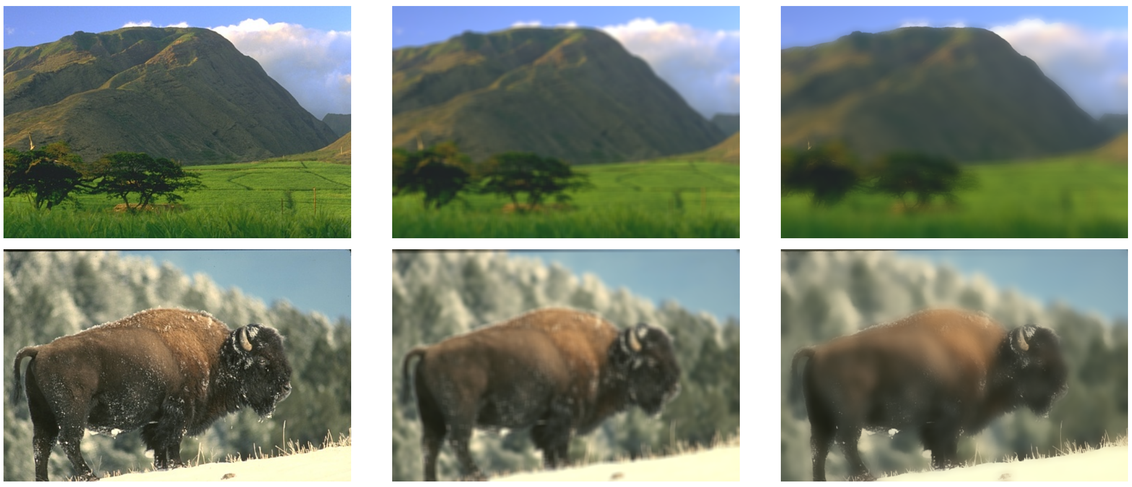


Figure 2.1.: Visual comparison of different filters. 2.1(a) Original image. 2.1(b) Gaussian blurring filter. 2.1(c) Bilateral filter.

The qualitative results of the smoothing filter are shown in figure 2.1. The original input image is shown in 2.1(a) and the filtered output image in 2.1(b). Please note that the edges are not preserved, but smoothed out. A Gaussian blurring filter has a wide field of applications, e.g. turning a halftone print into a gray scale image, removing outliers (noise) from a dataset, or reduce image details.

2.1.2. Bilateral Filter

The bilateral filter was introduced by Tomasi and Manduchi (1998). Instead of only one function measuring the geometric closeness, as in the Gaussian filter, a second function measuring the photometric similarity is introduced. Following the Gaussian example in the last section the geometric closeness $\hat{\boldsymbol{c}}$ is used as a convolution kernel

$$\hat{\boldsymbol{h}}(\boldsymbol{x}) \quad = \quad \hat{\boldsymbol{k}}_c^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{\boldsymbol{f}}\,(\boldsymbol{\zeta}) \cdot \hat{\boldsymbol{c}}\,(\boldsymbol{x},\ \boldsymbol{\zeta})\,\mathrm{d}\boldsymbol{\zeta}, \tag{2.4}$$

where k_c is the normalization as in equation (2.3). In cases where $\hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta})$ is shift invariant, it will only depend on the difference $\hat{\boldsymbol{c}}(\boldsymbol{x}-\boldsymbol{\zeta})$, like in the Gaussian smoothing filter. Now the photometric similarity is added, which is defined in a similar way as

$$\hat{\boldsymbol{h}}(\boldsymbol{x}) \;=\; \hat{\boldsymbol{k}}_s^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \hat{\boldsymbol{f}}(\boldsymbol{\zeta}) \cdot \hat{\boldsymbol{s}}\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{f}}(\boldsymbol{\zeta})\right) \mathrm{d}\boldsymbol{\zeta}. \tag{2.5}$$

Filtering only the color domain does not affect the input image much (Tomasi and Manduchi, 1998). Instead, when combining both approaches to

$$\hat{\boldsymbol{h}}(\boldsymbol{x}) \;=\; \hat{\boldsymbol{k}}_b^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \hat{\boldsymbol{f}}(\boldsymbol{\zeta}) \cdot \hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta}) \cdot \hat{\boldsymbol{s}}\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{f}}(\boldsymbol{\zeta})\right) \mathrm{d}\boldsymbol{\zeta}, \tag{2.6}$$

it leads to the bilateral filter. In the same way as the Gaussian filter in (2.2) replaces the pixel values at position $\boldsymbol{x}$ with an average of nearby pixels, the bilateral filter replaces the pixel values using an average of nearby and similar pixel values. Therefore the bilateral filter will smooth out weakly correlated differences in pixel values, but also will preserve sharp color edges.

Most commonly a Gaussian function is used for the geometric and photometric similarity, which yields

$$c(\boldsymbol{x},\ \boldsymbol{\zeta}) \;=\; \exp\left(-\frac{1}{2}\left(\frac{|\boldsymbol{x}-\boldsymbol{\zeta}|_2}{\sigma_c}\right)^2\right) \tag{2.7}$$

$$s(\boldsymbol{x},\ \boldsymbol{\zeta}) \;=\; \exp\left(-\frac{1}{2}\left(\frac{|f(\boldsymbol{x})-f(\boldsymbol{\zeta})|_2}{\sigma_s}\right)^2\right). \tag{2.8}$$

For small σ_s only very similar pixel values will be replaced, e.g. noise. In order to remove texture σ_s needs to be increased accordingly, which results in smoothing of any color edge, including object edges. However, the bilateral filter is a very interesting alternative to the Gaussian filter when removing noise. There are several algorithms running in real time, e.g. Gunturk (2010) or Yang et al. (2009), and thus the bilateral filter is accessible to the perception-action loop. A visual comparison to the Gaussian filter can be seen in figure 2.1(c).

2.2. Proposed filter

As the bilateral filter smoothes only small scaled noise and does not treat texture explicitly, but leaves the rest of the image mostly untouched (see figure 2.1(c) for a visual comparison) a new technique is required. The proposed filter must

- detect edges between regions,
- separate edges belonging to texture from edges belonging to objects, and
- fill the textured or noisy areas with color values found in the spatial neighborhood.

In section 1.2 the meaning of “texture” was defined. According to this definition we make two reasonable assumptions about the differences of texture and object edges:

- the color of the texture differs from the color of the object, and
- the texture of the object is significantly smaller than the object itself.

2.2.1. Texture Detection

Using the same nomenclature as in section 2.1.2, let $\hat{\boldsymbol{f}}(\boldsymbol{x})$ be the input image and $\hat{\boldsymbol{h}}(\boldsymbol{x})$ the filtered output signal. The geometric closeness is represented again by the function $\hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta})$ and the photometric similarity by $\hat{\boldsymbol{s}}\left(\hat{\boldsymbol{f}}\left(\boldsymbol{x}\right),\ \hat{\boldsymbol{f}}\left(\boldsymbol{\zeta}\right)\right)$. We require that the amount of texture detected depends upon one user defined parameter τ.

The parameter τ is used as a threshold, which distinguishes object borders and texture borders. Fine structures should be considered as texture, while others should be considered as objects. We therefore analyze the spatial neighborhood first and calculate the mean color as

$$\begin{aligned} \hat{\boldsymbol{m}}(\boldsymbol{x}) &= \hat{\boldsymbol{k}}_m^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \hat{\boldsymbol{f}}\left(\boldsymbol{\zeta}\right)\cdot\hat{\boldsymbol{c}}\left(\boldsymbol{x},\ \boldsymbol{\zeta}\right)\mathrm{d}\boldsymbol{\zeta} && (2.9)\\ \hat{\boldsymbol{k}}_m(\boldsymbol{x}) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \hat{\boldsymbol{c}}\left(\boldsymbol{x},\ \boldsymbol{\zeta}\right)\mathrm{d}\boldsymbol{\zeta}, && (2.10) \end{aligned}$$

where $\hat{\boldsymbol{c}}$ is now defined as a 2D step function

$$\hat{\boldsymbol{c}}\left(\boldsymbol{x},\ \boldsymbol{\zeta}\right) = \begin{cases} 1 & \boldsymbol{x}-\boldsymbol{a}\le\boldsymbol{\zeta}\le\boldsymbol{x}+\boldsymbol{b} \\ 0 & \text{else} \end{cases} \qquad (2.11)$$

with the condition $\boldsymbol{a},\boldsymbol{b},\boldsymbol{e}\in\mathbb{R}^2_{\ge 0}|\boldsymbol{a}+\boldsymbol{b}=\boldsymbol{e}$ using a fixed $\boldsymbol{e}$. This generates a rectangle of size $\boldsymbol{e}$ around $\boldsymbol{x}$. This definition is not feasible in the continuous domain as it generates a nonfinite number of subwindows to calculate. In the discrete case $\boldsymbol{a},\boldsymbol{b},\boldsymbol{e}\in\mathbb{N}^2_{\ge 0}|\boldsymbol{a}+\boldsymbol{b}=\boldsymbol{e}$, however, every pixel is checked and updated according to

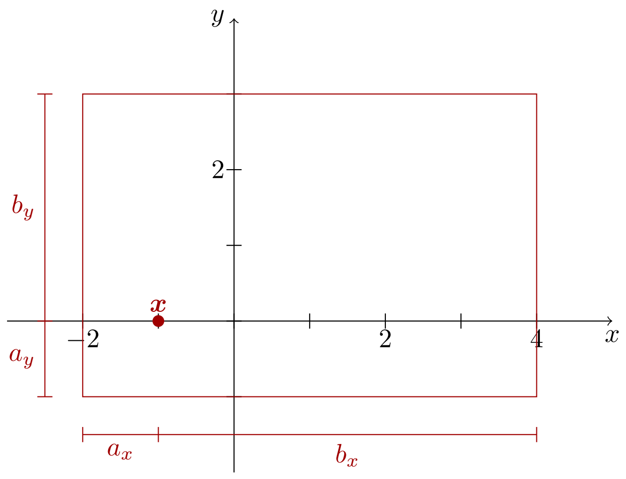


Figure 2.2.: Schematics of subwindows for one point $\boldsymbol{x} = (-1\ 0)^T$ in the discrete domain.

its neighborhood $\boldsymbol{e}$. For a better understanding of the implication, consider figure 2.2. It shows one possible example subwindow using $\boldsymbol{e} = (6\ 4)^T$ and $\boldsymbol{x} = (-1\ 0)^T$. The subwindow is then defined as $\boldsymbol{a} = (1\ 1)^T$ and $\boldsymbol{b} = (5\ 3)^T$. In the discrete case there would be 24 subwindows to be checked per point using a subwindow size of $6 \times 4\,\text{px}$.

Now, let $\boldsymbol{e} = (k\ l)^T$ and analyze just one subwindow of size $k \times l$. We define a distance function, which will allow thresholding:

$$d\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{m}}(\boldsymbol{x})\right) = \left|\hat{\boldsymbol{f}}(\boldsymbol{x}) - \hat{\boldsymbol{m}}(\boldsymbol{x})\right|_2. \tag{2.12}$$

Please note that the Euclidean metric is used here for a vector in the color domain $\hat{\boldsymbol{f}}_i = (r_i\ g_i\ b_i)^T$, which is defined the same way as in the spatial domain

$$d\left(\hat{\boldsymbol{f}}_1,\ \hat{\boldsymbol{f}}_2\right) = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}. \tag{2.13}$$

Now d yields the pixelwise color distance and integration

$$p(\boldsymbol{x}) = k_p^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} d\left(\hat{\boldsymbol{f}}(\boldsymbol{\zeta}),\ \hat{\boldsymbol{m}}(\boldsymbol{x})\right) \cdot \hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta})\, \mathrm{d}\boldsymbol{\zeta} \tag{2.14}$$

$$k_p(\boldsymbol{x}) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta})\, \mathrm{d}\boldsymbol{\zeta}, \tag{2.15}$$

results in the mean pixelwise color distance $p\left(\boldsymbol{x}\right)$ in the subwindow.

Now d and p represent a measure for noise and we can make a binary decision. When facing noise or texture in the subwindow, d will contain only a low variance in the subwindow and p should be close to d. On the other hand, an object edge will lead to a greater distance. For further understanding consider figure 2.3(a). The left bar shows ten noisy pixels, which serve as an input example. The right bar shows the filtered output. The corresponding calculations for each pixel are shown in figure 2.3(d). The mean pixel color value is computed from the input pixels[2] and the resulting pixelwise color distances and the mean pixelwise color distance are shown in blue. As expected they are both small and below a user based threshold $\tau = 20$ and thus detected as noise or texture and are filtered.

A large noisy color step on the other hand is shown in figure 2.3(b). The left bar shows the input pixels and the right bar the filtered output. The computational steps involved are visualized in figure 2.3(e). The mean pixel color value is of the order of half the step size of the input pixel color values. This results in both the pixelwise color distances and mean pixelwise color distance being above the user based threshold. This way an object edge (belonging to an object bigger than the subwindow) can be found. The corresponding pixels are therefore not filtered.

The binary decision based upon the distances d and p is therefore defined as

$$f\left(\boldsymbol{x}\right) = \begin{cases} 1 & d\left(\boldsymbol{x}\right),\ p\left(\boldsymbol{x}\right) \leq \tau \\ 0 & \text{else} \end{cases}, \qquad (2.16)$$

where 1 stands for a textured or noisy pixel value. As the calculations of all three mean distances m, d and p are susceptible to outliers, the image is prefiltered using a Gaussian low-pass filter and a very small spatial neighborhood as described in equation (2.2).

[2]Please note that in this work Greek symbols are used for discrete values and Latin letters for the continuous domain.

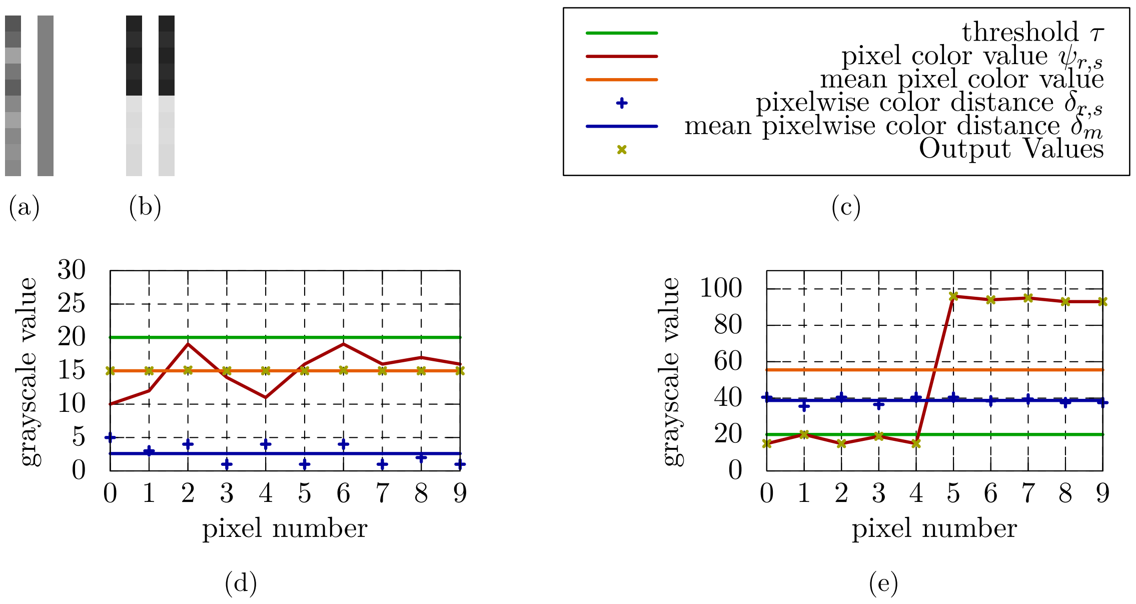


Figure 2.3.: 2.3(a) Low level white noise (left) in 10 pixels is used as input for filtering. The right bar shows the same 10 pixels after filtering. The computational steps involved in this image can be seen in figure 2.3(d). 2.3(b) A noisy step function is used for input. After filtering input and output are still identical and the edge is preserved. As the subwindows shift the noise will be filtered out eventually. The computational steps are shown in figure 2.3(e). 2.3(d) The pixelwise and mean pixelwise distance are both below the threshold τ. The output is filtered according to equation (2.18) and moved to the mean color. 2.3(e) The pixelwise and mean pixelwise distance are above the threshold and the output is not filtered. The edge is therefore preserved.

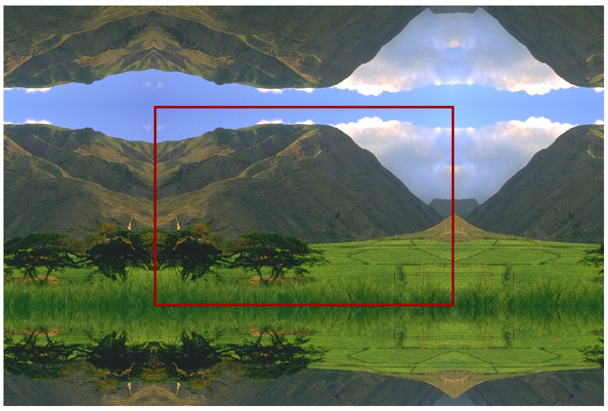


Figure 2.4.: In our approach periodic mirrored boundary conditions are used, as they lead to a good representation of the average texture at the border.

2.2.2. Texture Removal

After successfully identifying textured pixels, we need to fill the corresponding area with pixel values surrounding the textured pixel. We compute the mean color values in the subwindow $\hat{\boldsymbol{c}}(\boldsymbol{x})$ and use it to replace $\hat{\boldsymbol{f}}(\boldsymbol{x})$. The pixel color value is multiplied by the mean and weighted by the squared distance of d and the user based threshold τ. This leads to a photometric similarity function defined as

$$\hat{\boldsymbol{s}}\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{f}}(\boldsymbol{\zeta})\right) = \left(\tau - d\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{m}}(\boldsymbol{x})\right)\right)^2 |\hat{\boldsymbol{m}}(\boldsymbol{x})|_2 . \tag{2.17}$$

This equation results in the texture equation, yielding for $f(\boldsymbol{x}) = 1$

$$\hat{\boldsymbol{h}}(\boldsymbol{x}) = \hat{\boldsymbol{k}}_R^{-1}(\boldsymbol{x}) \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \hat{\boldsymbol{f}}(\boldsymbol{\zeta}) \cdot \hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta}) \cdot \hat{\boldsymbol{s}}\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{f}}(\boldsymbol{\zeta})\right) \mathrm{d}\boldsymbol{\zeta} \tag{2.18}$$

$$\hat{\boldsymbol{k}}_R(\boldsymbol{x}) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \hat{\boldsymbol{c}}(\boldsymbol{x},\ \boldsymbol{\zeta}) \cdot \hat{\boldsymbol{s}}\left(\hat{\boldsymbol{f}}(\boldsymbol{x}),\ \hat{\boldsymbol{f}}(\boldsymbol{\zeta})\right) \mathrm{d}\boldsymbol{\zeta} \tag{2.19}$$

and for $f(\boldsymbol{x}) = 0$

$$\hat{\boldsymbol{h}}(\boldsymbol{x}) = \hat{\boldsymbol{f}}(\boldsymbol{x}). \tag{2.20}$$

Lastly the question of boundary condition remains. As we want to approach the average noise, periodic mirrored boundary conditions are used. They are visualized in figure 2.4.

2.2.3. Discrete Domain

While the continuous domain is very helpful for analyzing the exact behaviour of the filter, the actual implementation must be done in the discrete domain. The filter running on a single threaded central processing unit (CPU) performs far from real-time, see table 3.1. As the filter is required as a pre-processing step in real-time, acceleration is needed. An implementation on a graphics processing unit (GPU) shows significant improvements (also shown in table 3.1). In this section we show a serial version of the filter in the discrete domain and the following section holds a parallel implementation. The consecutive order of the steps are shown in the flowchart in figure 2.5. Step 1 to 3 describe the texture detection phase, while step 4 handles the texture removal.

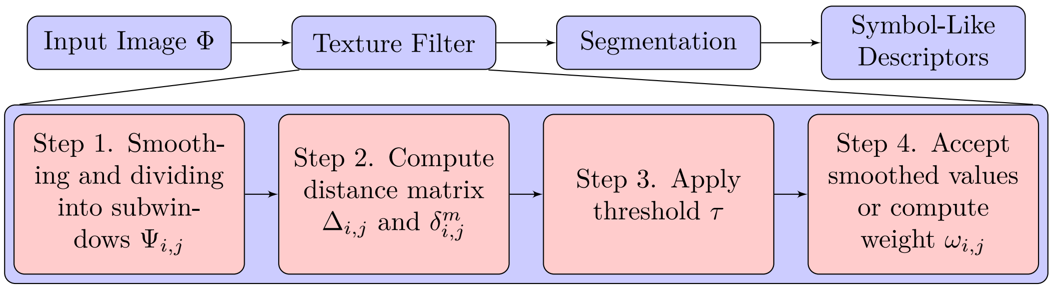

Input Image Φ
Texture Filter
Segmentation
Symbol-Like Descriptors
Step 1. Smoothing and dividing into subwindows $\Psi_{i,j}$
Step 2. Compute distance matrix $\Delta_{i,j}$ and $\delta^m_{i,j}$
Step 3. Apply threshold τ
Step 4. Accept smoothed values or compute weight $\omega_{i,j}$

Figure 2.5.: Schematic of the proposed filter.

Step 1: Smoothing and Division. As noted in equation (2.2) the Gaussian blurring filter is based on a convolution. Even though the filter can be applied in real-time on modern hardware, it is not the fastest approach. As our goal is to remove outliers efficiently, we downscale the image instead of blurring it, which gives similar results, but in a more efficient way. In our approach we used a downscaling factor of 0.75, meaning that the image size is reduced to $\frac{3}{4}$ of its original size.

Afterwards, the image is loaded into the matrix Φ where each entry is a color vector $\hat{\boldsymbol{\varphi}}_{i,j} = (\varphi^r_{i,j}\ \varphi^g_{i,j}\ \varphi^b_{i,j})^T$. To create a subwindow of the size $k \times l$ as described in the step function (2.11), the pixels $\hat{\boldsymbol{\varphi}}_{i,j}$ to $\hat{\boldsymbol{\varphi}}_{i+k,j+l}$ are copied into a subwindow Ψ. Please note that $(r\ s)^T$ is in the range of the subwindow size $k \times l$, and $(i\ j)^T$ is in the range of the image size.

Step 2: Computation of distance matrix. The integral in (2.9) is equivalent to the arithmetic mean in the discrete domain, which yields

$$\hat{\boldsymbol{\psi}}_m = \frac{1}{N}\left(\sum_{r,s}\psi^r_{r,s}\ \sum_{r,s}\psi^g_{r,s}\ \sum_{r,s}\psi^b_{r,s}\right)^T, \tag{2.21}$$

where $N = k \cdot l$ denotes the total number of pixels in the subwindow. The pixelwise distance can be computed as

$$\delta_{r,s} = |\hat{\boldsymbol{\psi}}_{r,s} - \hat{\boldsymbol{\psi}}_m|_2, \tag{2.22}$$

where $\delta_{r,s}$ makes up the so called distance matrix Δ of the subwindow. Lastly, the mean pixelwise distance is calculated as

$$\delta_m = \frac{1}{N}\sum_{r,s}\delta_{r,s}. \tag{2.23}$$

Step 3: Thresholding As already described in the previous section, $\delta_{r,s}$ and δ_m are now used for noise detection. Analogous to (2.16), we define the thresholding decision according to

$$\varphi_{r,s} = \begin{cases} 1 & \delta_{r,s},\ \delta_m \leq \tau \\ 0 & \text{else} \end{cases}, \tag{2.24}$$

where 1 stands for a textured or noisy pixel.

Step 4: Updating of RGB color values. In equation (2.11) it is noted that each pixel is checked once for every possible position of the subwindow. For simplification we use sliding subwindows and update every pixel in these sliding subwindows. Therefore, a global weight $\omega_{i,j}$ is used, which is initialized with zeros and gets updated $N = k \cdot l$ times:

$$\omega_{i,j} \longleftarrow \omega_{i,j} + \varphi_{r,s} \cdot (\tau - \delta_{r,s})^2 . \tag{2.25}$$

The global weight $\omega_{i,j}$ will be used for normalization later. For the updated pixel values we need to create a third image frame of the size of the original image, which we call Θ. Θ is initialized with zeros and is updated according to

$$\hat{\boldsymbol{\theta}}_{i,j} \longleftarrow \hat{\boldsymbol{\theta}}_{i,j} + \varphi_{r,s} \cdot (\tau - \delta_{r,s})^2 \cdot \hat{\boldsymbol{\psi}}_m \tag{2.26}$$

for each subwindow, which means that every $\hat{\boldsymbol{\theta}}_{i,j}$ will be updated $N = k \cdot l$ times. Still, we have to consider that we are using a resized image. We rescale the image Θ to its original size and let the global weight $\omega_{i,j}$ make up the weight matrix Ω, which is rescaled to the same size. Then the original image and Θ are added and normalized using[3] Ω.

2.2.4. Implementation on Parallel Hardware

While the algorithm is very expensive on traditional CPUs, it is also parallelizable. The core idea is to start one thread for every subwindow. This results in starting

[3]Even though it is highly improbable that any entry in Ω equals zero, 1 is added to every entry. As in general $\omega_{i,j} \gg 0$ and specifically $\omega_{i,j} \geq 0$ is true, this does not change the outcome significantly.

about as many threads as there are pixels in the image[4]. Obviously, one can run almost any moderately improved algorithm in real-time, if the hardware is fast enough. As it is our goal to use the filter on traditional, mid-level GPUs, several enhancements are needed.

GPU-Architecture

For understanding the parallelization a short introduction into GPUs is given. Multithreading on GPUs using the programming languages C and C++ has been an interest to several projects, for example Dagum and Menon (1998) or Aoki et al. (2010). In this work we will focus on GPUs by NVIDIA. The only project offering full support of all features for NVIDIA graphic cards is the Compute Unified Device Architecture (CUDA), which was developed by NVIDIA and is used for this work. NVIDIA divides its graphic cards into several classes of computational capability, where each class offers different features. A summary of the specifications of different classes can be seen in table A.1 in the Appendix. In this work a card offering the capability of 2.0 was used (Cook, 2012, Farber, 2011, Sanders and Kandrot, 2010).

Short Introduction to CUDA

For a short introduction, let us consider the example of adding two vectors, each having a length of 1024 integer values. For full parallelization, we want to have one thread handle one entry of the vector, which means we have to generate 1024 threads in total. Using the command

```
Add <<<2, 512>>> (...);
```

we can start a function "Add" using 2 blocks and each block containing 512 threads: the result is shown in figure 2.6. Now each thread can do one single summation and return the result. The question remains why we had to divide the vector onto two blocks. The answer is very simple: the maximum number of threads per block is limited and may vary between 512 and 1024, depending on the hardware. Blocks are organized in a grid.

For this work, GPUs by NVIDIA were used. For parallel programming NVIDIA offers a C++ library called CUDA which handles the interface between the so called

[4]Due to the boundary conditions there are slightly more threads. To be exact there are $(u + 2(k-1)) \cdot (v + 2(l-1))$, where $u \times v$ represent the amount of pixels in the original image, and $k \times l$ the size of the used subwindows.

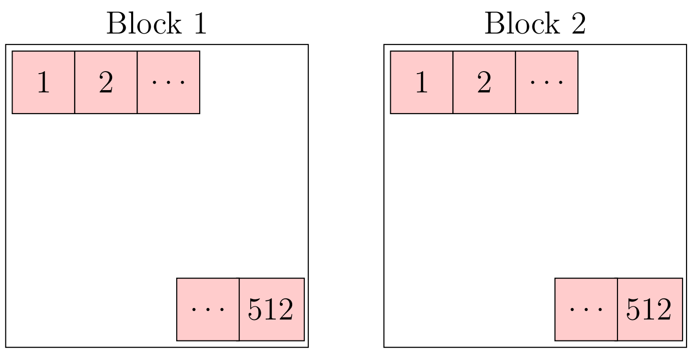


Figure 2.6.: Example of a multithreading process. Two blocks are generated, each starting 512 threads.

"host" (the CPU) and the "device " (the GPU). In fact, the "Add" command above is handled by the CUDA library. While this would be enough to parallelize a program, CUDA also offers to place blocks and threads on a 2D or 3D lattice, which is very convenient for 2D or 3D input data. An example is given in figure 2.7, which could be generated using the following code:

```
dim3 dimGrid (2, 2, 1);
dim3 dimBlock (5, 2, 1);
Add <<<dimGrid, dimBlock>>> (...);
```

The function "Add" is called a kernel function, which is executed on the GPU using 4 blocks and each block starting 10 threads, as specified by dimGrid and dimBlock.

This becomes interesting as soon as we want to allocate memory. On a GPU, similar to a CPU, we mainly use two different kind of memories: global memory and shared memory. Global memory can be accessed by all threads on all blocks and may have a size of several gigabytes. On the contrary, each block has a fixed amount of shared memory, which can only be accessed by threads from the same block. For example a thread in block (0,1,0) cannot access shared memory on block (0,0,0). Shared memory has a typical size of several kilobytes[5], but is, compared to global memory, significantly faster. In the case of very intense calculations, shared memory has to be used to achieve real-time performance.

[5]We use a NVIDIA GPU with a graphics capability of 2.0 offering 48 kB of shared memory per block.

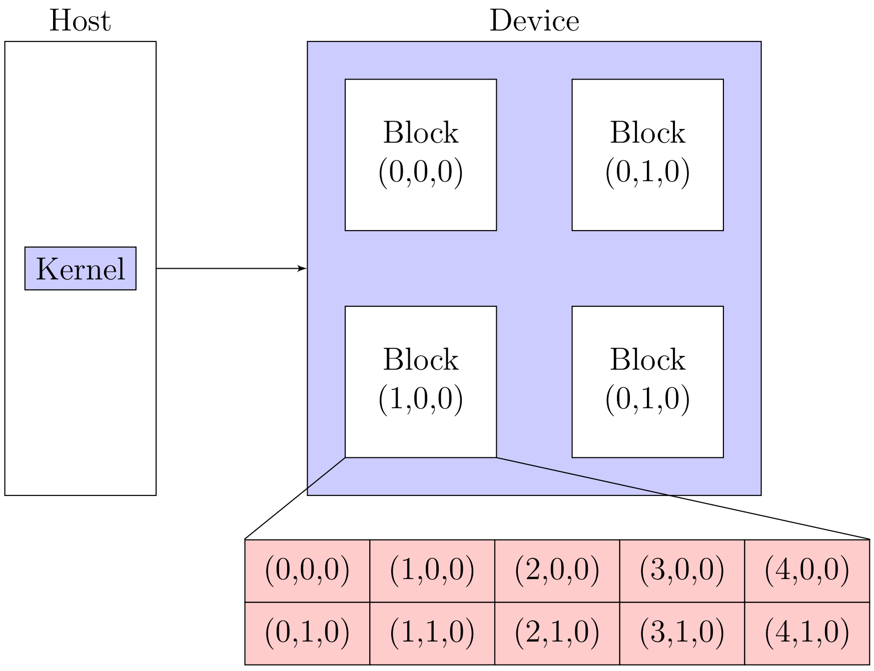


Figure 2.7.: Example of a multithreading process. A grid of 2×2 blocks is generated, each starting 10 threads (also on a 2D system of 5×2 threads).

Filter Implementation

As already shown, we have almost as many subwindows as there are pixels in an image, which is in the range of $(1 \ldots 9) \cdot 10^5$ subwindows to compute, and one thread handles one subwindow. Best performance results were achieved using 512 threads in a lattice of 32×16 threads per block. A visualization of how the image is divided into several blocks is shown for the first two blocks in figure 2.8. The overlap is needed for the shifting subwindows. Since the pixels of the original image have to be accessed very often, they are loaded into shared memory as well. As there is one thread per pixel, one thread copies one pixel into shared memory. A syncing command ensures that all threads are finished copying before starting with the computational steps as described in section 2.2.3.

Performance Enhancement

To achieve real-time performance numerous enhancements were necessary. Two steps were already mentioned:

- Instead of using a Gaussian blurring filter, the image is downsized. Thus, outliers are removed and less pixels remain to be computed.

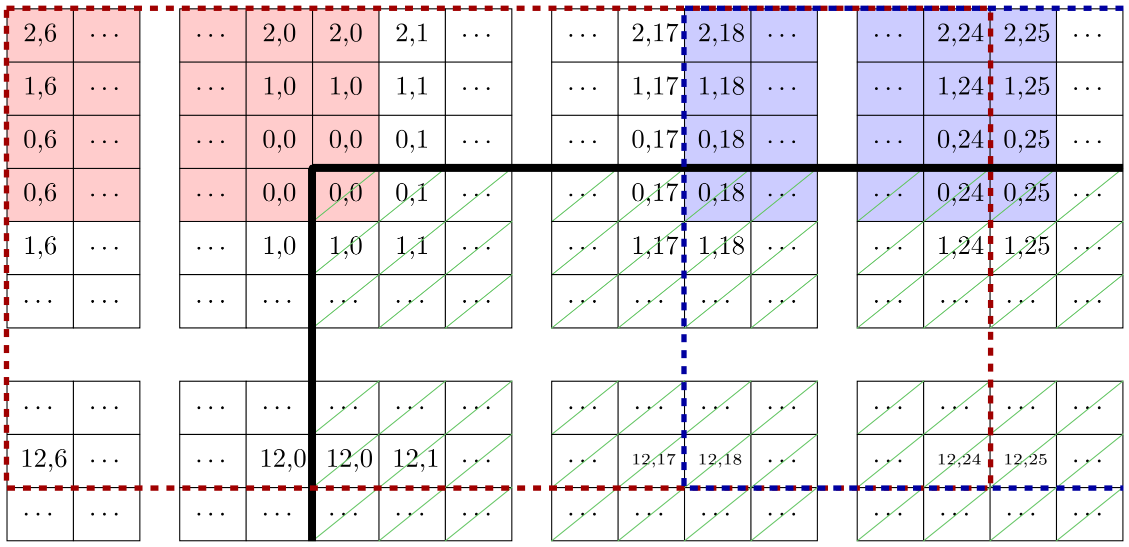


Figure 2.8.: Division of input image onto several blocks. The pixels being read and computed by the first block are marked with a dashed red line, the pixels computed by the second block are marked with a blue dashed line. Shown with a red background is the first subwindow of the first block for the pixel at position (0, 0). The first subwindow of the second block for the pixel (0, 25) is marked with blue. A solid black line marks the image boundaries. The image area itself is marked using green stripes. Please note the mirrored periodic boundary conditions.

- Because global memory is significantly slower than shared memory on a GPU, shared memory and registers are used for variables that need to be accessed often.

Time performance improvement was a major obstacle in this work and several more improvements had to be implemented. The most important ones are described here. As shown in section 2.2.3, the arithmetic mean needs to be calculated. The core clock of the GPU typically has a range of several hundred MHz and is much slower than modern CPUs. Obviously, calculating the mean scales linearly with the number of pixels in one subwindow. Now, consider the following one-dimensional example, where we want to compute the two shifting averages of four variables in a sequence of five variables $\varphi_1 \ldots \varphi_5$

$$\delta_1 = \frac{1}{4} \cdot \sum_{i=1}^{i=4} \varphi_i \tag{2.27}$$

$$\delta_2 = \frac{1}{4} \cdot \sum_{i=2}^{i=5} \varphi_i. \tag{2.28}$$

If δ_1 is already computed, the computation of δ_2 may be shortened according to

$$\delta_2 = \delta_1 + \frac{1}{4} (\varphi_5 - \varphi_1) . \tag{2.29}$$

While in this toy example only one summation can be saved, this process runs significantly faster sequentially on a modern CPU than computing the full average for each subwindow in parallel on a GPU.

Another, more significant, improvement comes from the number of threads and the subwindow size. Due to the two dimensional nature of an image we also want a two dimensional system of threads per block. The hardware implementation of a GPU has the limitation that the system of threads should be a multiple of 8. Therefore, a system of 32×16 threads is used, resulting in 512 threads per block. The subwindow size was chosen to be a window of 8×4 px, which is a trade off between high quality results and speed. As already mentioned the subwindow size is the spatial environment with which one pixel is compared. For this work, cameras on robots, which offer typically a low resolution, are most important. Therefore the subwindow size can be chosen to be comparatively small.

Another optimization comes from a Cuda feature known as "streams". Streams are a system of fifo buffers[6] for CUDA instructions. All data, that is used on the GPU, needs to be copied from the device to the host memory. The notable aspect is that the copy pipeline is independent from the processing pipeline. Copying and processing may therefore be done in parallel as well. For example, when using two streams, the first stream can be used to copy an array of data to the GPU and afterwards process it. At the moment the copying is finished the copy pipeline is idle, which means that the second stream may start using it for copying another array of data: an example can be seen in table 2.1. In this work four streams were

[6]Fifo is short for "first in, first out".

Time Step	Stream 1	Stream 2
0	copy array 1 from host to device	-
1	process array 1	copy array 2 from host to device
2	copy array 1 from device to host	process array 2
3	-	copy array 2 from device to host

Table 2.1.: Visualization of the fifo system of streams. As soon as the copy pipeline is empty (copying of array 1 is finished), stream 2 starts. Therefore, processing and copying may also be parallelized. A more detailed introduction can be found in (Cook, 2012, Farber, 2011, Sanders and Kandrot, 2010), the specifications and details about the hardware implementation can be found in (NVIDIA, 2012).

used, which operate in a similar way.

Lastly, a huge performance improvement results from memory coalescing. This means that all threads must read global and shared memory in a consecutive order, meaning threads 0, 1, 2, and 3 must read memory 0x0, 0x4, 0x8, and 0xc for the best performance. For example, consider the following matrix A:

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & a & b \end{pmatrix}. \tag{2.30}$$

If there are four threads available in this example, we need to ensure that

$$\begin{aligned} \text{thread 0 accesses}: &\ 0,\ 4,\ 8 \\ \text{thread 1 accesses}: &\ 1,\ 5,\ 9 \\ \text{thread 2 accesses}: &\ 2,\ 6,\ a \\ \text{thread 3 accesses}: &\ 3,\ 7,\ b \end{aligned}$$

for a coalesced access. Related to coalescence are bank conflicts. Shared memory is organized in so called banks. A bank conflict occurs when several threads try to access the same memory module at the same time, in which case the access is serialized, losing the advantages of parallel access[7].

[7]As simple as it sounds, a lot of time was spent on this exact enhancement, since there are several matrices in the memory of the GPU. The time performance increased significantly after full implementation.

2.3. Segmentation

The task of image segmentation is to divide the image delivered by robotic sensors into meaningful regions. For a perfect object segmentation, one region corresponds with one object. We want segmentation to be done online, without human supervision. This means that no prior knowledge about the scene is given and therefore no assumptions about the number or shape of regions can be made. On the contrary, human beings have a model based approach to segmenting images, meaning that humans from a certain age on normally have learned what specific objects look like (Li and DiCarlo, 2010). In the past decades, numerous different approaches have been proposed, an overview is given in e.g. Abramov (2012). As already noted we want to increase performance of the segmentation algorithm and thus decrease noise in the perception-action loop. The segmentation algorithm should be based on color only and work otherwise as generally as possible, which results in several specific requirements:

- Data-driven: instead of being based on models, the algorithm should analyze the data only.
- Automatic: the algorithm should not use prior knowledge about the data.
- Unsupervised: any parameters should be found automatically. There should be no tuning to different scenes (e.g., indoors, outdoors, . . .) and no supervised learning necessary.
- Online: the algorithm should use only preceding information of the video information, as future perception is undefined in a robotic system.
- Temporal coherency: segments should keep their labels throughout the video.
- Dense: every pixel in every frame should be analyzed.
- Real-time: the algorithm should be able to process at least 25 fps on modern hardware.

There are several online techniques, e.g. Abramov et al. (2010), Hedau et al. (2008), Liu et al. (2008), Paris and Durand (2007), Vazquez-Reina et al. (2010), but only Abramov et al. (2010), Paris and Durand (2007) may process data in real-time under the mentioned conditions. The mean-shift algorithm by Paris and Durand (2007) may process only gray scale images in real-time and needs all past data. As it is our goal to apply the filter to video streams, only the algorithm by Abramov et al. (2010) could be used. To allow for a comparison with other segmentation techniques single images are filtered and partitioned first. In that way neither real-time, nor temporal coherency is needed. Still we want the algorithms being data-driven, automatic, dense, and unsupervised. As both the mean-shift algorithm (Paris and Durand, 2007) and the so called graph-based algorithm (Felzenszwalb and Huttenlocher, 2004) are standard segmentation algorithms and fulfill our demands, we will compare those to the algorithm by Abramov et al. (2010) called Superparamagnetic Clustering of Data, which meets all requirements mentioned above (the algorithm Superparamagnetic Clustering of Data is based on the Metropolis algorithm (Beichl and Sullivan, 2000) and will be referred to as "Metropolis algorithm" in this work).

As we want to test the segmentation on different scenes (e.g. indoors, outdoors, different light sources, wide selection of objects), the Berkeley Segmentation Dataset (Martin et al., 2001) is used. The dataset contains images, as well as several "ground truth" segmentations. Ground truth is the segmentation found by humans by manually partitioning images. It is assumed to be the ideal segmentation. In figure 2.9, a visual comparison of several ground truth segmentations is given. The Berkeley image dataset offers several human segmentations per image and normally there is not one true segmentation, as objects boundaries are perceived differently by people. In this work all given human segmentations were averaged. The dataset is publicly available and used by various universities and is the de facto standard for comparison[8].

Originally, the mean-shift algorithm was presented by Fukunaga and Hostetler (1975). In mean-shift segmentation every pixel is associated with a feature vector, e.g. color, position, texture, or range values. These feature vectors are considered as samples from an underlying probability density function that needs to be segmented. Mean-shift computes a weighted mean of feature vectors for every pixel around a local neighborhood and finds maxima in the density function. Each feature vector is associated with the nearby peak of the datasets probability density function (Cheng, 1995, Comaniciu and Meer, 2002, Paris and Durand, 2007). For a given set of n feature points $\{\boldsymbol{x}_i\}$ and a starting point $\boldsymbol{y}_0$, a series $\{\boldsymbol{y}_i\}$ of successive averages is

[8]The dataset is available at https://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/.

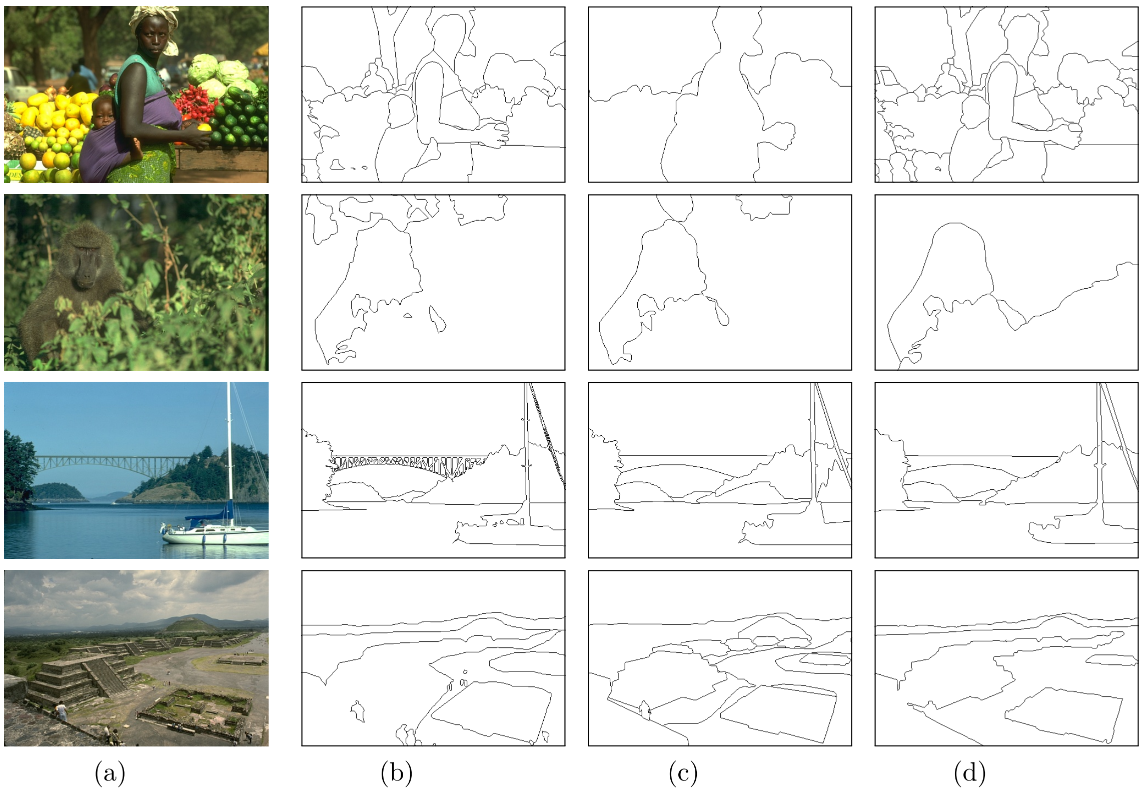


Figure 2.9.: Visual comparison of ground truth information. 2.9(a) Original input image. 2.9(b) - 2.9(d) Ground truth segmentation by different humans. As perception of objects differs, so does the ground truth information.

computed using a weighting kernel function (e.g. Gaussian function). By assigning each $\boldsymbol{x}_i$ a seeding value $\boldsymbol{y}_0^i$ the corresponding series can be computed and, when converging to the same limit, grouped.

In graph-based methods the image is represented using a weighted undirected graph. Each pixel has one edge to each of its eight adjacent neighbors. An edge weight defines the similarity between the pixels. Then the algorithm merges pixels into regions based on their relative dissimilarity of a local neighborhood. The algorithm was first proposed by Felzenszwalb and Huttenlocher (2004) and runs linearly with the number of graphs. An implementation by Grundmann et al. (2010) processes images in real-time and is therefore applicable to video segmentation.

The Metropolis algorithm may compute stereo and monocular images in real-time on parallel hardware and was proposed by Abramov et al. (2010). It is based on the method of superparamagnetic clustering of data. The input image is represented using a Potts model of spins (Potts and Domb, 1952), where each pixel is represented by one spin vector comparable to the Ising model in two dimensions. The image is segmented into regions by finding the equilibrium states of the energy function of the ferromagnetic Potts model in the superparamagnetic phase. Each spin may have n different states. Depending on the systems temperature, three system phases are observed: the paramagnetic, the superparamagnetic, and the ferromagnetic phase. In the paramagnetic phase all spins are in disorder, in the ferromagnetic phase all spins are aligned. Clusters of similar spin states refer to segments in the input image (Abramov et al., 2010). The algorithm includes four parameters α_1, α_2, n_1, and n_2. The output is affected most by the parameter α_1, which influences the coupling strength of the spins and therefore the final number of segments.

Segmentations are commonly quantitatively evaluated using the precision and recall method by Martin et al. (2004). It defines a measure for over- and under-segmentation. An image is considered to be over-segmented if it is split too finely compared to the ground truth information. Conversely, an image is considered to be under-segmented if it is split too coarsely compared the ground truth information. Given a machine segmentation S and a ground truth segmentation S', precision measures the fraction of boundary pixels in the machine segmentation that correspond to boundary pixels in the ground truth segmentation. It is therefore sensitive to over-segmentation:

$$\text{precision} = \frac{\text{matched}\,(\text{S},\ \text{S}')}{|S|}. \tag{2.31}$$

Recall measures the fraction of boundary pixels in S which are also found in S' . It is sensitive to under-segmentation:

$$\text{recall} = \frac{\text{matched}\,(\text{S}',\ \text{S})}{|S'|}. \tag{2.32}$$

This results in precision and recall values in the range $[0;1]$, where 0 represents no match and 1 is a perfect match to ground truth information. Both precision and recall can be summarized into one value, the F-score. For the general case it is defined as

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}} \tag{2.33}$$

for a positive and real β (Rijsbergen, 1979). For segmentation the harmonic mean is used, which corresponds to $\beta = 1$ and will be referred to as F-score in this work

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}. \tag{2.34}$$

3. Experimental Results

3.1. Texture Filter Results

As shown in the last chapter, we expect the filter to remove small scaled texture and noise in a defined spatial neighborhood. Consider figure 3.1 for a first impression of the filter. It shows an artificially created testing image with one color border, several small stripes, and dots of three different sizes to resemble noise. In the filtered output the small dots, and the small blue stripes are filtered out, while the color edge, and the bigger dots and stripes remain.

3.1.1. Time Performance

In table 3.1 the average frame rates for images of different sizes may be seen. A visual comparison of the time performance of the CPU and GPU implementation is given in figure 3.2. Images were filtered 100 times and the average taken. As already shown in section 2.2, the computational complexity is independent of the threshold used. The GPU version is roughly 30 times faster than the CPU approach, independent of the image size. For images of size 480×320 px real-time is achieved. For processing the following experimental environment was used: CPU 2.2 GHz AMD Phenom(tm) 9550 Quad-Core Processor (using a single core) with 4 GB RAM and GPU GeForce GTX 580 (with 1.5 GB device memory). Each image was filtered twice using a subwindow size of 8×4 px and 4×8 px.

3.1.2. Effect of Filter Parameters

The filter should replace areas of similar color with the mean color and therefore remove small scaled texture and noise. There are two parameters which have an effect on the filtered output image. First, the subwindow size as defined in figure 2.2, and second, the user based threshold τ as defined in equation (2.16). The subwindow size defines the spatial neighborhood in which texture is detected. It

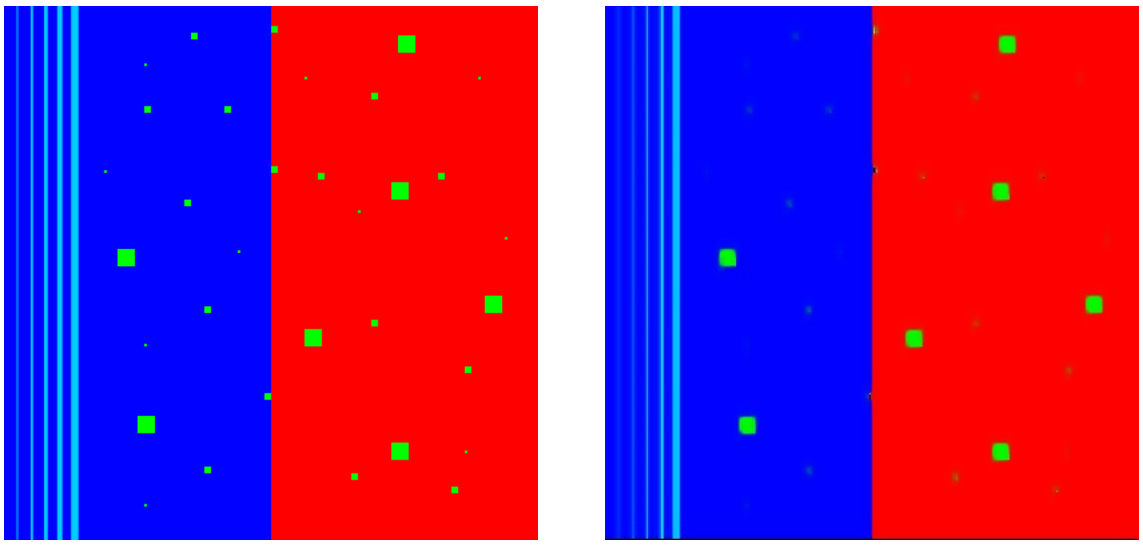


Figure 3.1.: Effect of filtering on an artificial input image using the proposed texture filter and a threshold of $\tau = 30$. Left is the original input image, to the right is the filtered output image.

Image Size	CPU		GPU	
[px]	[Hz]	[s]	[Hz]	[ms]
240×180	3.03	0.33	80.38	12.4
320×240	1.66	0.60	48.00	20.8
400×300	1.05	0.95	30.26	33.0
480×320	0.80	1.25	23.81	42.0
640×480	0.40	2.50	12.35	81.0
800×600	0.24	4.17	7.65	130.7
1024×768	0.15	6.67	4.24	235.9

Table 3.1.: Time performance for images of different sizes. The test image was taken from the training set of the Berkeley Segmentation Dataset and Benchmark Martin et al. (2001). 100 measurements were taken and averaged.

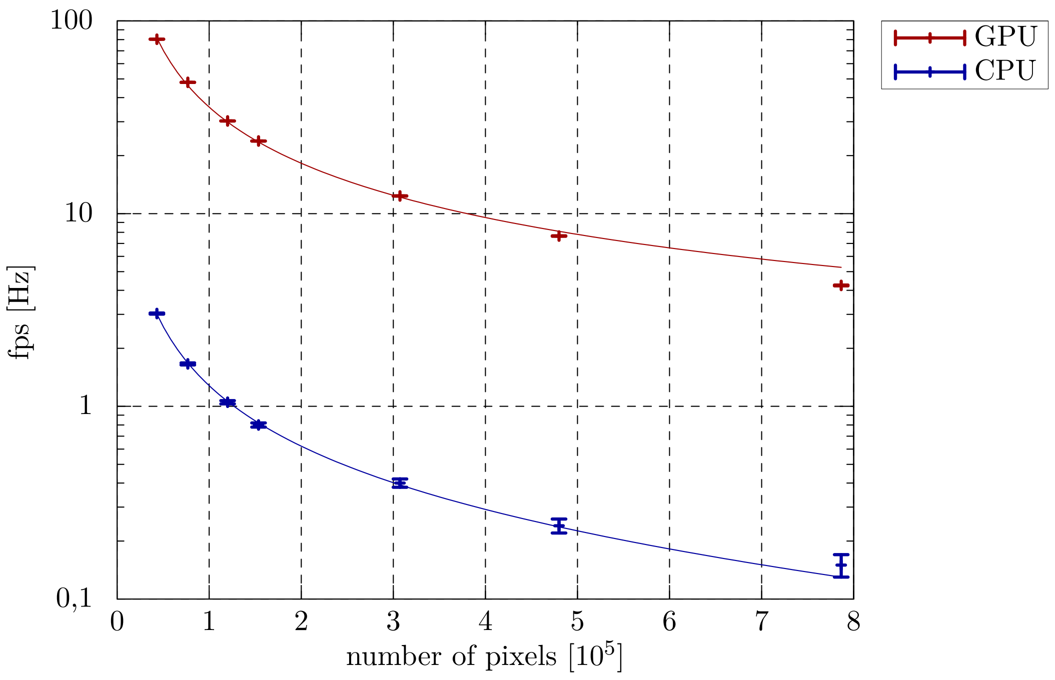


Figure 3.2.: Visual comparison of the time performance of the single threaded CPU version and the parallelized GPU version. All data is also shown in table 3.1. Interpolation was used to approximate values between data points.

depends on the robot application, and the quality and size of the input images. As the Oculus system provides a video stream sized 320×256 px, a subwindow of 8×4 px offers good results. For symmetric reasons each frame is filtered a second time, using a subwindow sized 4×8 px.

The parameter τ defines the threshold of the color step for variations, which are detected as either objects or small scaled texture (or noise). If τ is too small, no filtering is done and the image only gets blurred due to the Gaussian blurring of the input image as described in equation (2.16). From a certain size of τ, the filtered output image should not change much, since every pixel is detected as texture or noise and filtered. An image sequence for different τ values may be seen in figure 3.3. After a threshold of $\tau \geq 30$ the output does not change much. As later will be shown, $\tau = 30$ leads to the best segmentation results. For a quantitative evaluation for the amount of texture in the filtered image Chen and Shih (1995) and Fah et al. (2003) suggest four measures. For the input image Φ, the output image Θ, and the image dimensions $u \times v$ they are as follows.

The mean absolute error (mae)

$$mae = \frac{1}{uv} \cdot \sum_{i=0}^{u-1} \sum_{j=0}^{v-1} \left| \hat{\boldsymbol{\varphi}}_{i,j} - \hat{\boldsymbol{\theta}}_{i,j} \right| \tag{3.1}$$

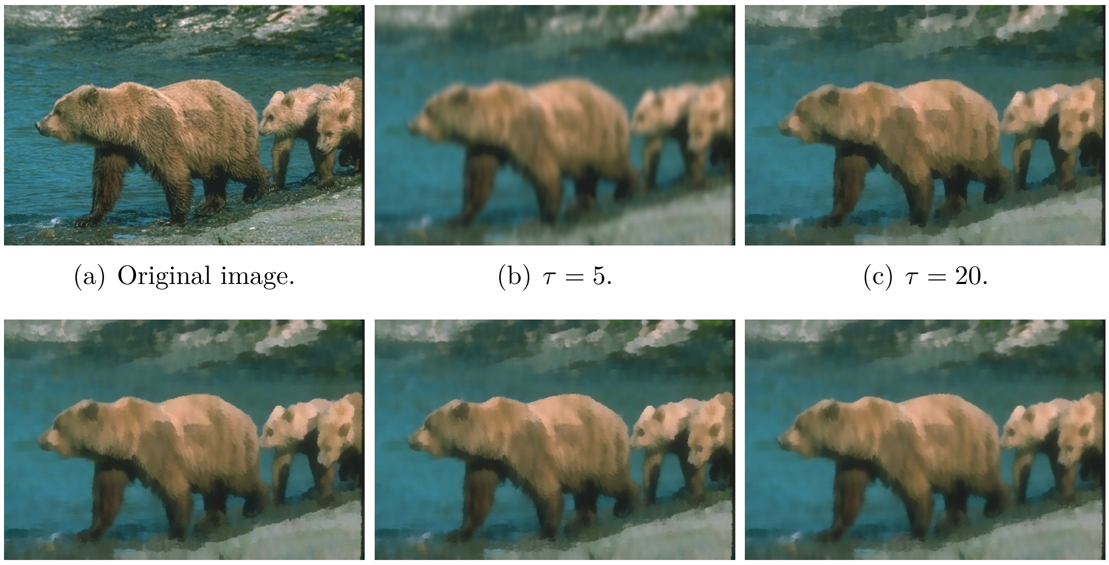
is used in statistics to measure how close a prediction is to the actual outcome. In this case it is used to compare the difference between the filtered and original image. The root mean square error (rmse)

$$\begin{aligned} rmse &= \sqrt{\frac{1}{uv} \cdot \sum_{i=0}^{u-1} \sum_{j=0}^{v-1} \left(\hat{\boldsymbol{\varphi}}_{i,j} - \hat{\boldsymbol{\theta}}_{i,j} \right)^2} \\ &= \sigma_e \end{aligned} \tag{3.2}$$

computes a similar value but weights the difference quadratic. In contrast the signal to noise ratio (snr)

$$snr = 10 \cdot \log_{10} \left(\frac{\sigma^2}{\sigma_e^2} \right), \tag{3.3}$$

where σ^2 is the variance of the original image and σ_e^2 is the variance of the filtered image, compares the level of the desired signal to the background noise level. Here



(a) Original image. (b) $\tau = 5$. (c) $\tau = 20$.

(d) $\tau = 30$. (e) $\tau = 40$. (f) $\tau = 60$.

Figure 3.3.: Effect of different thresholds τ. For low thresholds the image is more blurred, after $\tau = 30$ it does not change much.

the desired signal is the original image, which is compared to the filtered output. Lastly the entropy remains as

$$entropy \quad = \quad -\sum_{i=0}^{u-1}\sum_{j=0}^{v-1} P(i,j)\cdot \log P(i,j), \tag{3.4}$$

where $P(i,j)$ denotes the probability that the pixel (i,j) occurs. The entropy is computed independently for all three color channels and later combined according to

$$entropy \quad = \quad \sqrt{entropy_r^2 + entropy_g^2 + entropy_b^2}. \tag{3.5}$$

The entropy is a measure for the predictability of a random variable. Each pixel is treated as one random variable. In case of an image only containing white noise the entropy is as high as possible, whereas for a uniform colored image the entropy is as low as possible. Therefore, the entropy should decrease in cases where large regions of an image are being filled with the same color, as it is the case for the proposed texture filter.

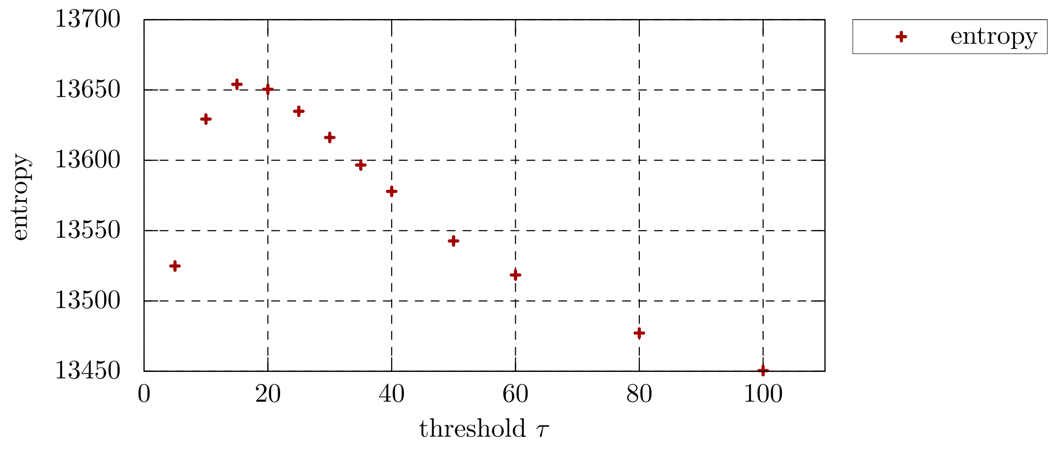


Figure 3.4.: Entropy according to equation (3.5) of the proposed texture filter for various thresholds τ.

These four parameters were computed for all filtered images of the Berkeley Image Dataset and averaged. The results are shown in the Appendix B.1 to B.4. The mean absolute error and root mean square error show similar behaviour: an exponential decline. As for small $\tau \leq 20$, the image is only blurred and no color replacement is being done, so the original and filtered images differ significantly. As τ is increased ($\tau > 20$) pixel color values are replaced again by mean colors and the image borders sharpen again. From a certain τ on also image borders are replaced by mean values and the image looks more similar to the original image. This τ value, which separates small scaled texture and objects, depends on the image scenery. For natural images, as it is the case for many robotic applications and for the Berkeley dataset, the sensible range is $\tau \in [25;\ 40]$.

Figure B.3 confirms these observations. As for small τ, the images are very dissimilar, they get more and more similar as τ is increased. Entropy, on the other hand, is a measure for randomness and it can be used for measuring the amount of texture. In figure B.4 the entropies for all three color channels can be seen. All three increase and show a maxima at around $\tau = 20$. The maxima can be seen more clearly in figure 3.4, which shows the total entropy according to equation (3.5). For τ values before the maxima the image is only blurred or single pixels are being replaced. Thus the randomness of the image is increased. At $\tau > 20$, bigger segments are formed and the randomness declines again. However, the maxima is very small.

filter	mae	rmse	snr	entropy r	entropy g	entropy b	length
Original	–	–	–	19508	19950	19640	34132
Gauss	18.8	29.9	8.9	7076	7374	8570	13544
bilateral	19.5	27.2	9.2	7095	7442	8764	13687
texture	104.1	846.6	−2.9	7054	7387	8705	13616

Table 3.2.: Quantitative comparison of Gaussian blurring filter ($\sigma = 13$), the bilateral filter ($\sigma_s = 20$, $\sigma_c = 200$), and the proposed texture filter ($\tau = 30$). The last digit of each result is uncertain, due to rounding and error propagation.

A visual comparison to the Gaussian blurring filter as in equation (2.2) and to the bilateral filter (2.6) is shown in figure 3.5. While the Gaussian blurring filter acts as a low-pass filter, high variance of the images is removed. This includes noise as well as color borders. The bilateral filter removes noise more efficiently and preserves object borders. The filter does not distinguish between texture and object borders. Therefore either both are preserved or smoothed out. This effect is shown in figure 3.6.

For a quantitative evaluation, the four measures (3.1) to (3.4) were calculated for all images in the Berkeley dataset and averaged. The results are shown in table 3.2 and compared to the Gaussian and bilateral filter. While the Gaussian blurring filter and the bilateral filter achieve noise values in the same range, the proposed texture filter has highly increased mean absolute error, root mean square error, and signal to noise ratio values. As both, the Gaussian and bilateral, act as noise reduction filters, the changes are rather subtle. The proposed filter however is supposed to replace large areas of color variations and replace them with the mean color value. This results in increased error values and an increased signal to noise ratio. However, the entropy represents the “randomness” of an image and contains a measure of the diversity of pixel color values. All filters lower the entropy significantly, but stay in the same range. Even though texture is preserved by the bilateral and Gaussian filter, it does not raise the randomness fundamentally.

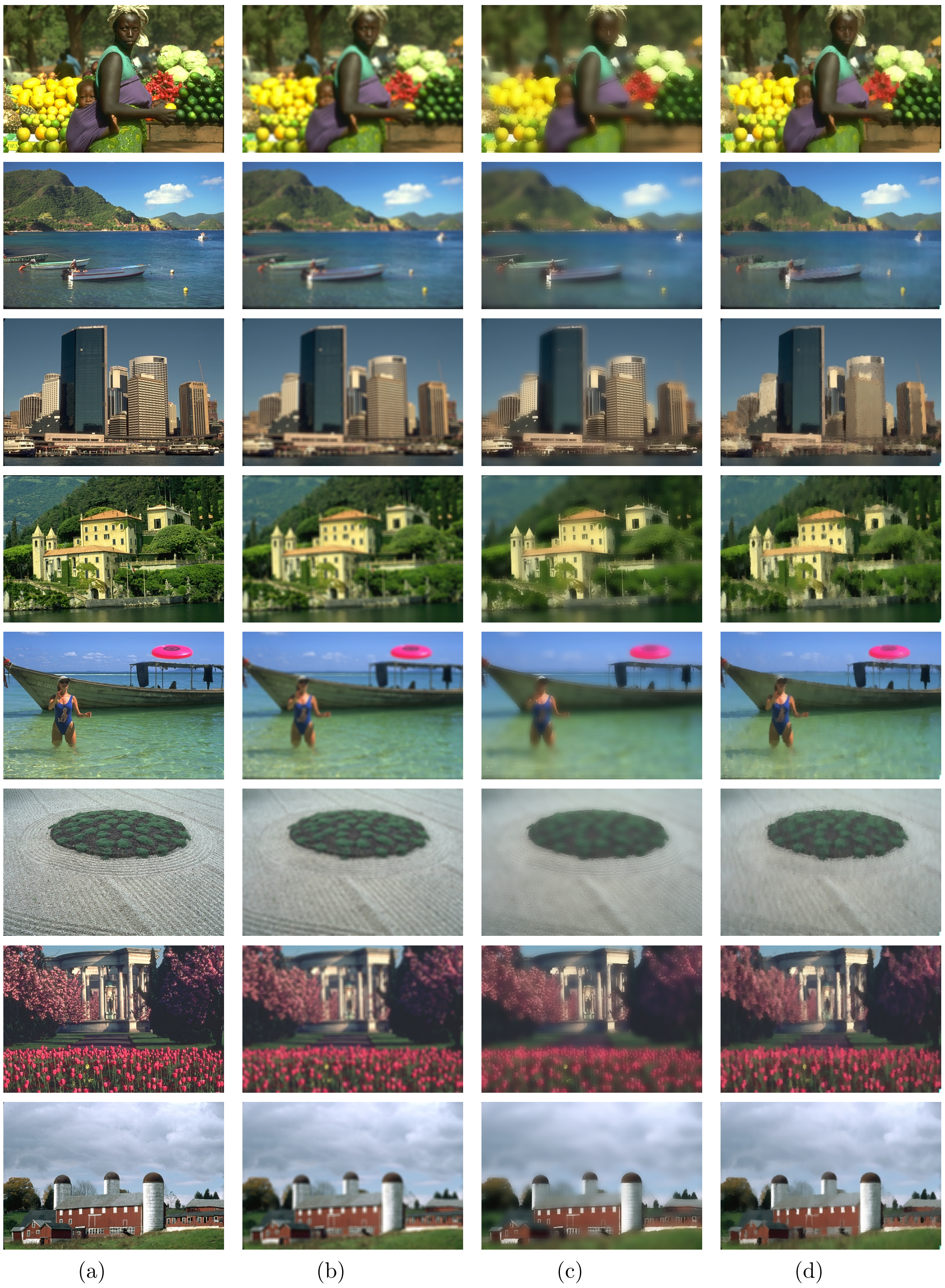


Figure 3.5.: Visual comparison of different filters. 3.5(a) Original input image. 3.5(b) Gaussian blurring filter. 3.5(c) Bilateral filter. 3.5(d) Proposed texture filter.

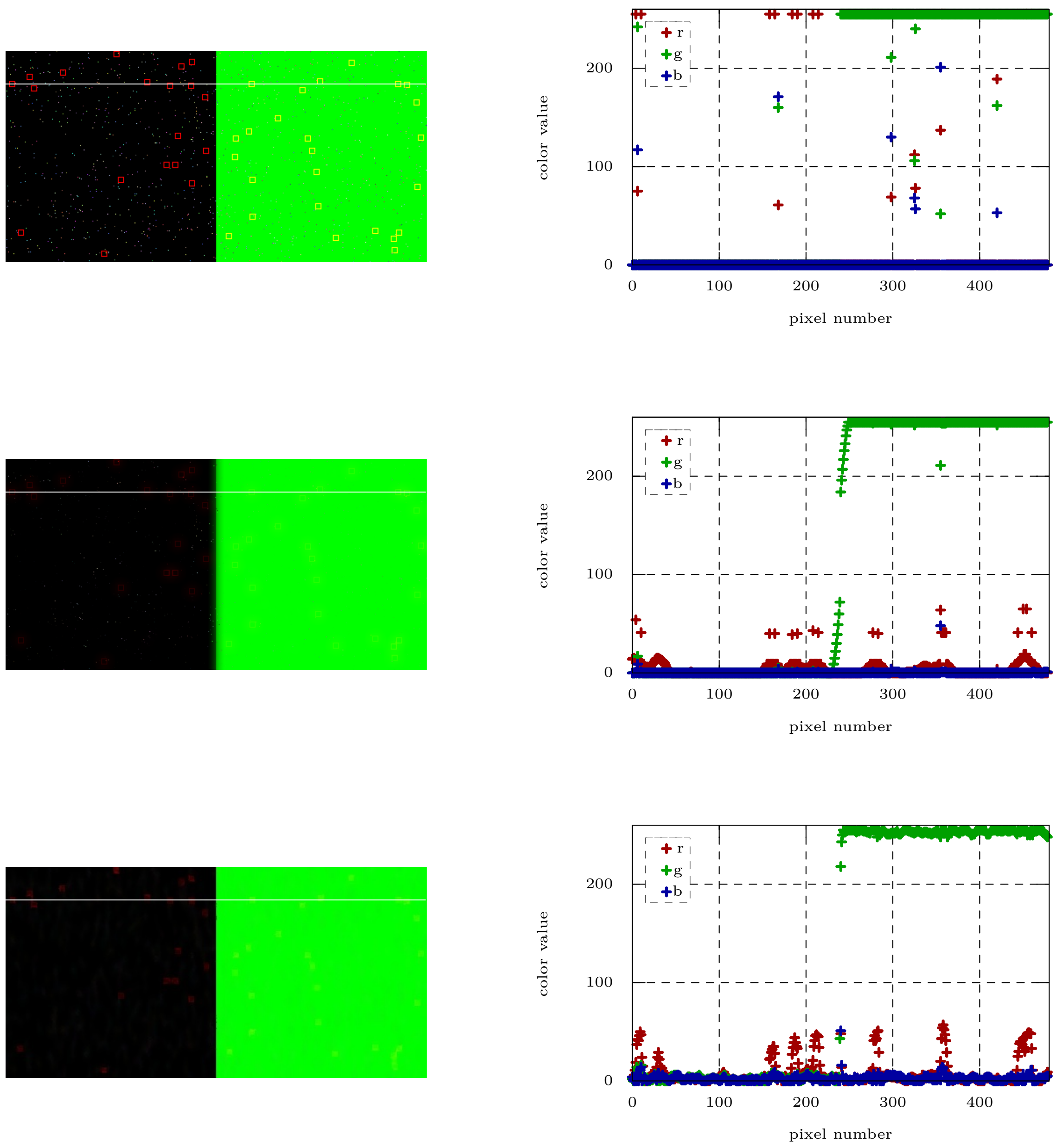


Figure 3.6.: On the left side an artificial image with three features is shown: a black-green color edge, texture is simulated using red squares 7 px wide, and white noise. A cross section is marked using a white line and the corresponding pixel color values are shown on the right side. From top to bottom: original input image, bilateral filter, and proposed texture filter. For the bilateral filter an arbitrary large kernel of 20 px in the space domain and using $\sigma_c = 200$ for the color domain was used to remove all squares and most of the noise. The color edge of the bilateral filter has a width of 18 px. For the proposed texture filter on the other hand the edge remains sharp and all texture, as well as noise is smoothed out. Since the channels are not treated independently, there is a small variance in all color channels.

3.2. Segmentation Results

The filter acts as a preprocessing step for segmentation. As described by Paris and Durand (2007) for the mean-shift and by Felzenszwalb and Huttenlocher (2004) for the graph-based segmentation algorithm, both perform their own preprocessing steps using averages of a spatial neighborhood. No filtering is advised for these algorithms. Still, all three segmentation algorithms need some user based parameters. The mean-shift segmentation needs three input parameters: the Gaussian parameters $\sigma_c \in [5;\ 10]$ for the color domain and $\sigma_s \in [4;\ 256]$ for the spatial domain, and a persistence threshold $\tau_p \in [0;\ 5]$. The ranges are recommended by the authors. Experimentally the values $\sigma_c = 2$, $\sigma_s = 8$, $\tau_p = 1$ were found to show best performance on the Berkeley Dataset (Abramov, 2012).

Similarly, the mean-shift algorithm takes three parameters. σ is used for a Gaussian smoothing before segmenting the image. k represents a value used for the threshold function, and min is a minimum component size enforced by post-processing. The authors recommend $\sigma = 0.5$, $k = 500$, and $min = 20$ for segmenting arbitrary images.

The Metropolis algorithm uses four parameters: n_1 is the number of the first Metropolis iterations, and n_2 the number of iterations during relaxation. α_1 represents the coupling strength during the first n_1 iterations, and similarly α_2 the coupling strength during relaxation. The author recommends $n_1 = 10$, $n_2 = 20$, $\alpha_1 = 1.0$, and $\alpha_2 = 4.0$. However, as α_1 is the most influential parameter, it will be evaluated on the Dataset in this work. For the other three parameters the recommendations are used and kept constant.

As the segmentation is supposed to perform in automatic mode, a fixed value for the threshold τ has to be found. As shown in figure 3.3, the output does not change significantly when $\tau \geq 30$, and so the same effect is expected for segmentation. In figure 3.7 the precision and recall for the Metropolis algorithm is shown for various τ, and all parameters of the Metropolis algorithm are kept fixed. The F-score shows a maxima at $\tau = 30$, which signifies a good value.

As a best value for τ is found, the Metropolis algorithm needs to be optimized. The most significant parameter for Metropolis is the α_1 parameter, which influences the coupling strength of the spin states and therefore determines into how many segments the image is partitioned. In figure 3.8 the performance of the Metropolis

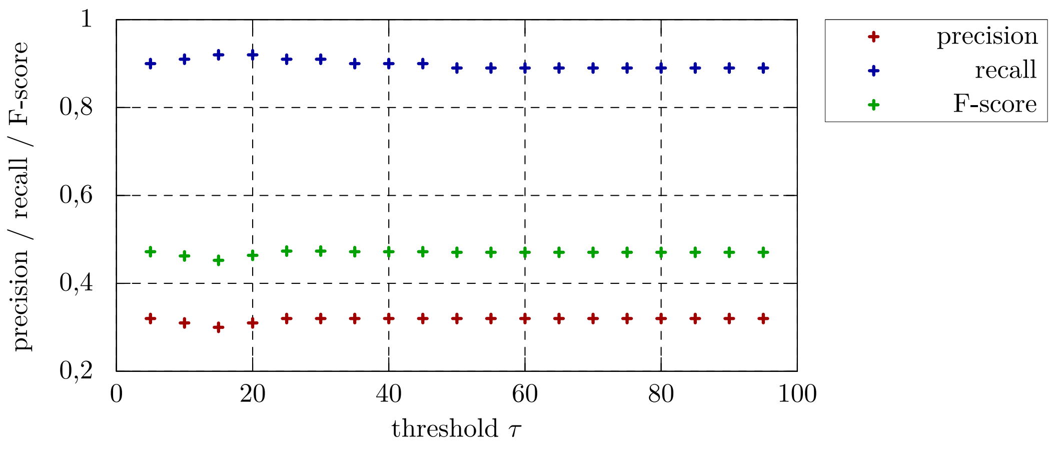


Figure 3.7.: Metropolis algorithm performance with fixed parameters using filtered images for various thresholds τ. All images from the Berkeley Dataset and Benchmark (Martin et al., 2001) were segmented and the results averaged.

algorithm is shown for various α_1. As α_1 grows, the coupling strength is enhanced and less segments are formed, since more pixels tend to belong to one segment. Recall, which is a measure for under-segmentation, declines significantly. Even though the F-score does not have its maximum at $\alpha_1 = 1$, it is the best trade-off between precision and recall. For more convenience, precision is often plotted against recall, as shown in figure 3.9. As both precision and recall peak at 1, performance improves as the results approach the value (1 1). As described by Martin et al. (2004), curves from the left upper corner to the right bottom corner are expected when tuning one parameter of the segmentation algorithm. This means that there is a trade-off between over- and under-segmentation. As one improves, the other one is expected to get worse.

In figure 3.10 a comparison between the Metropolis algorithm without any filter, Metropolis using the bilateral filter, Metropolis using the proposed texture filter, the graph-based algorithm, and the mean-shift algorithm can be seen. Even without any filter the Metropolis algorithm outperforms graph-based and mean-shift segmentation. Preprocessing the image using the bilateral filter improves results, but best performance is shown using the proposed texture filter.

A visual comparison between graph-based, mean-shift, and Metropolis algorithm is shown in figure 3.11. The qualitative results seem to imply that the filter reduces

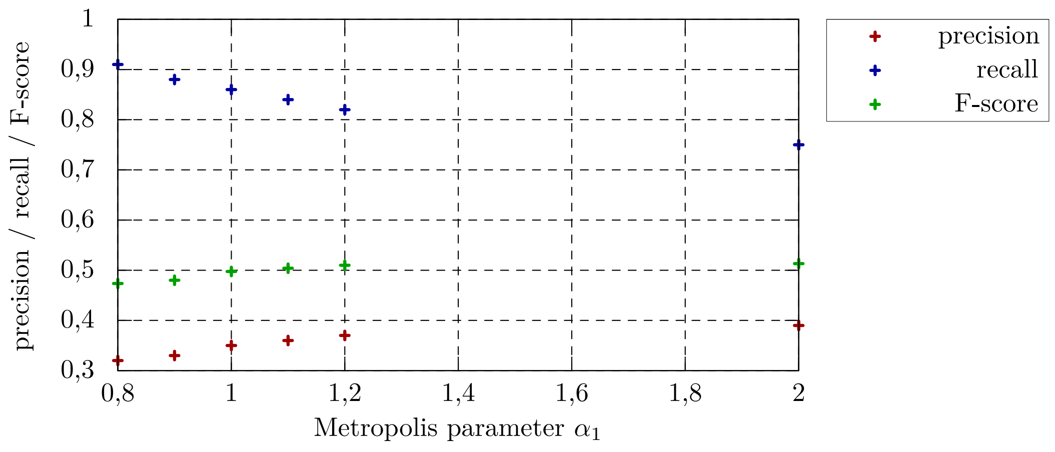


Figure 3.8.: Metropolis algorithm performance using filtered images with $\tau = 30$ for various α_1. As shown by Abramov et al. (2010) $\alpha_1 \in (0; 10]$ and is expected to be $\alpha_1 \approx 1$. Due to the very high computational cost, values between $1.2 < \alpha_1 < 2.0$ are not shown. All images from the Berkeley Dataset and Benchmark (Martin et al., 2001) were segmented and the results averaged. The same data is shown in figure 3.9 as a precision-recall graph.

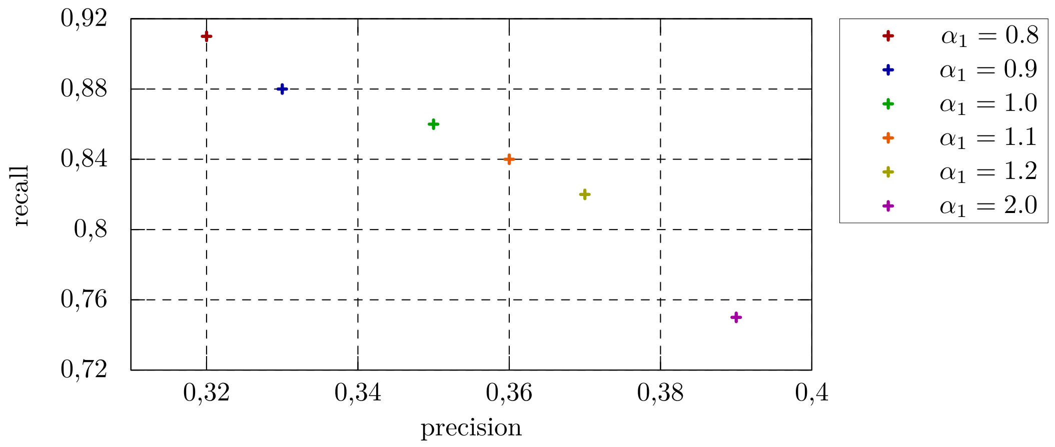


Figure 3.9.: Metropolis algorithm performance using filtered images with $\tau = 30$ for various α_1.

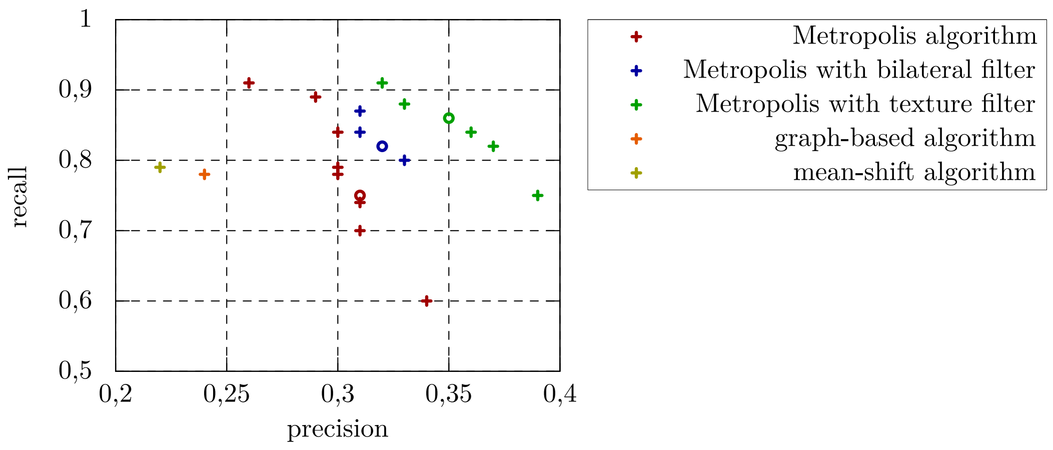


Figure 3.10.: Comparison between graph-based, mean-shift and Metropolis algorithm. Metropolis segmentation is shown for various α_1 values, $\alpha_1 = 1.0$ is marked with a circle.

over-segmentation significantly. Moreover, areas containing almost the same color, as it is often the case for the background, are being partitioned into the same segment more often. More segmentation results of single images can be seen in Appendix B.5 to B.8.

Frames from the segmentation of a short video sequence using Metropolis can be seen in figure 3.12. Two segmentations are shown. In 3.12(c) the input image is segmented without filter. 3.12(d) shows the same frames of the video stream, but is filtered using the texture filter before segmentation. As the Metropolis algorithm is based on a spin model there are problems with domain fragmentations (Abramov, 2012, Eckes and Vorbrüggen, 1996). Domain fragmentation describes areas being split into several segments despite high attractive forces across them. These can be seen in the last two frames in 3.12(c) and 3.12(d) on the white styrofoam pad, as well as in the last two frames in 3.12(c) on the ground. Domain fragmentation happens when the systems starting temperature is low or decreased rapidly. They cannot currently be handled in real-time.

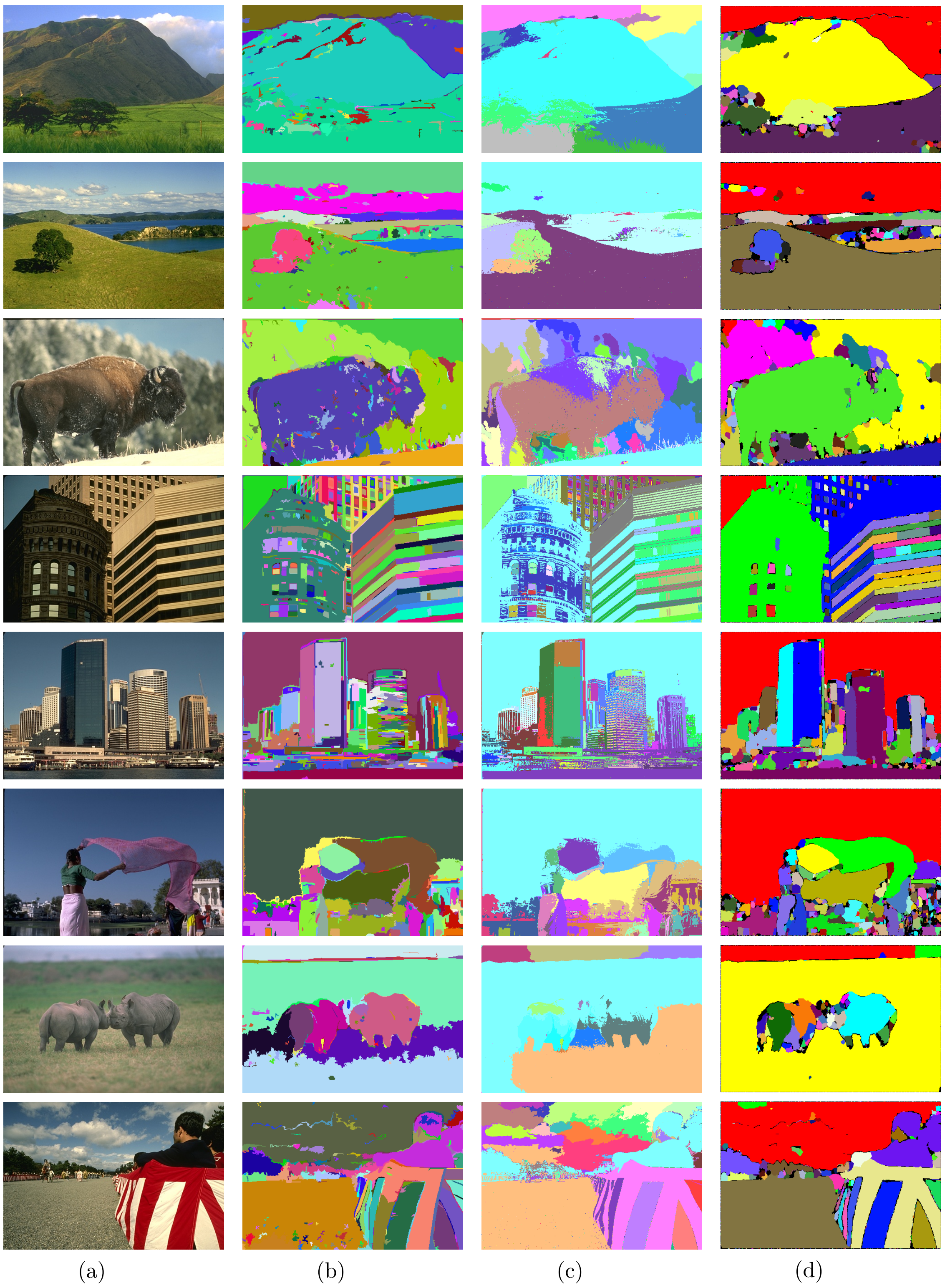


Figure 3.11.: Visual comparison of segmentation methods. 3.11(a) Original image. 3.11(b) Graph-based segmentation. 3.11(c) Mean-shift segmentation. 3.11(d) Proposed texture filter in combination with Metropolis algorithm.

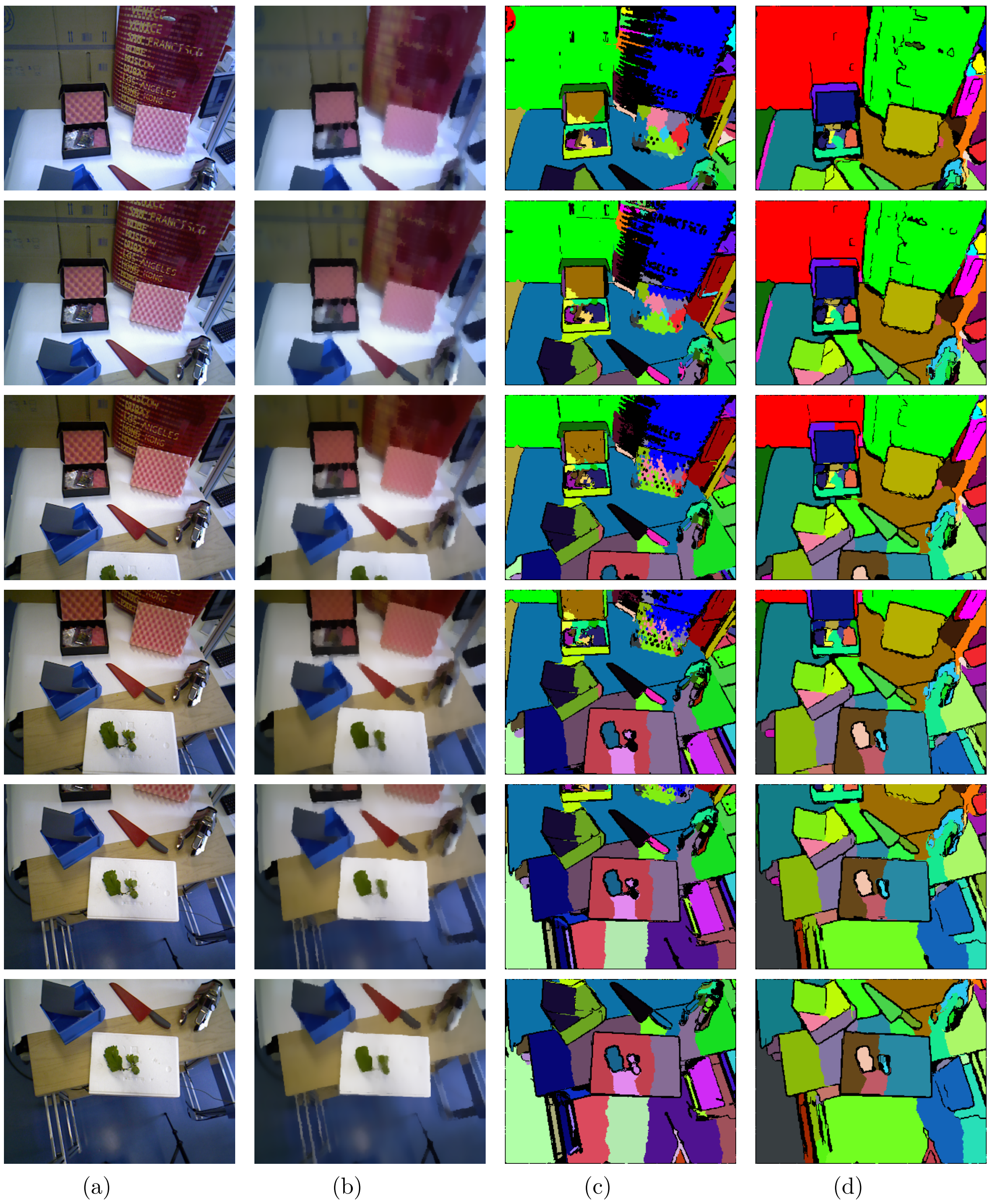


Figure 3.12.: Visual comparison of a short video sequence. 3.12(a) Original frames from sequence. 3.12(b) Filtered frames from sequence using the proposed texture filter. 3.12(c) Segmentation using the Metropolis algorithm without filter. 3.12(d) Segmentation using the Metropolis algorithm and the proposed filter. There is significant less over-segmentation in textured areas (e.g. suitcase, pad).

4. Discussion and Outlook

In this chapter we shall discuss the filter in its entirety, specifically its pros and cons, and future work. One of the main goals of this work was to find a way to improve the image segmentation process and reduce noise in the perception action loop. The results support the hypothesis that the reduction of texture in an image leads to less over-segmentation and improves the segmentation results. For previous filters it was not common to distinguish between object and texture borders. Texture detection enables the filter to remove the texture while preserving object edges. The results show that in color based segmentation, prefiltered images show significantly less over-segmentation and the overall segmentation is strongly improved. An implementation of the filter on parallel hardware performs in real-time and is therefore accessible to the perception-action loop of robots.

We would like to fill areas belonging to one object with the objects mean color in order to enhance the segmentation process. In contrast, the Gaussian blurring filter uniformly blurs edges as well as noise, and filters based on the bilateral filter only smooth noise, but preserve texture. Denoising filters (Devi et al., 2010, L. et al., 2009) try to enhance the visual perception of an image by reducing the amount of noise. The proposed filter enhances the segmentation process significantly and, as it is independent from the segmentation algorithm, it can be applied as a preprocessing step to any segmentation algorithm.

As filling areas with their mean color is the goal, not preserving the original image, a rectangular neighborhood function for the spatial domain was used. This leads to good time performance, but round edges are not preserved. This can be seen in figure 4.1. The neck of the swan shows pixels instead of a smooth boundary. The pixels are not due to low resolution, since the other side of the neck is preserved correctly. This effect can be demonstrated on almost all high contrast edges. A neighborhood window based on a Gaussian function probably solves this problem,

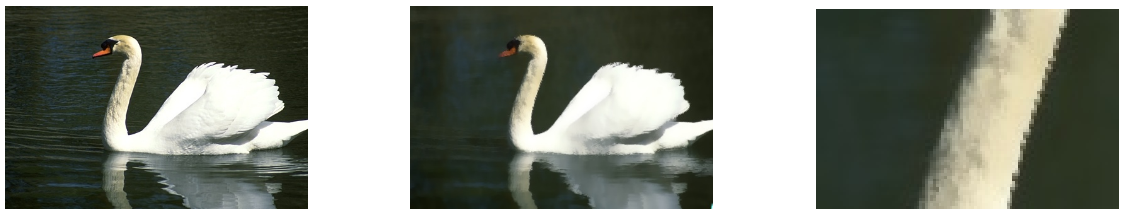

(a) Original image. (b) Filtered image. (c) Filtered image magnified.

Figure 4.1.: Visual comparison of proposed filter. As a rectangular spatial neighborhood was chosen, round edges are not preserved.

but also leads to a significantly higher computational cost.

The approach used to achieve real-time performance is based on modern GPUs. Modern GPUs may consume up to several hundred watts. This is not feasible for mobile, autonomous applications. However, an implementation on a microcontroller, specifically an FPGA[1], is feasable and could be an option for mobile applications. An FPGA has a power consumption of only several watts and may process up to several million computations simultaneously. A major drawback for an FPGA application is the long design time. The implementation of the algorithm on a microcontroller for fully mobile applications remains future work.

[1]FPGA is short for “Field Programmable Gate Array”.

5. Conclusion

Optical illusions are probably so fascinating for human beings because they not only give some insight in the working of the brain, but also demonstrate that sometimes the human vision system can fall short. Our vision system is usually relentlessly flawless and a failure is a novelty to us! In computer vision on the other hand, you soon meet plenty of obstacles and a great complexity and often a solution can only be given for a small subset of applications. In this work the author overcame some of these obstacles and in this final chapter, we shall make a critical assessment of the applicability and value of this work.

In everyday life we meet tedious tasks, which we would like to have completed automatically. In fact, if we were to sit down and write a list of all things that we can do with our eyes, but would like to have done by a computer, it would be a rather long list. The applications of computer vision are seemingly endless. Many of these wishes on the list will include interaction with objects and therefore object recognition. Object recognition is based on classification, which again is based on segmentation.

A novel filter was proposed which removes texture and noise by replacing the corresponding pixel values with a mean color. Object edges are preserved and the filter processes images in real-time. The proposed algorithm needs two user based parameters and an evaluation is shown for both. Values are presented for arbitrary images and therefore the filter can be used in automatic mode.

The proposed texture filter is only a very small step in the computer vision world. The main goal is to improve the vision of robots and eventually arrive at fully mobile and autonomous robots. There are many different avenues that interested parties could explore, but the most reasonable conclusion is that there is still a lot more work to do in the world of computer vision.

A. Appendix: Methods

A.1. Compute Capabilities of NVIDIA GPUs

Technical Specifications	Compute Capability (version)				
	1.0	1.1	1.2	1.3	2.0
Maximum dimensionality of grid of thread blocks	2	2	2	2	3
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535	65535	65535	65535	65535
Maximum dimensionality of thread block	3	3	3	3	3
Maximum x- or y-dimension of a block	512	512	512	512	1024
Maximum z-dimension of a block	3	64	64	64	64
Maximum number of threads per block	512	512	512	512	1024
Maximum number of threads per multiprocessor	768	768	1024	1024	1536
Maximum number of resident blocks per multiprocessor	8	8	8	8	8
Maximum amount of shared memory per multiprocessor	16 kB	16 kB	16 kB	16 kB	48 kB

Table A.1.: Comparison of compute capabilities of NVIDIA graphic cards, see also (NVIDIA, 2012). In this work a GPU with the capability 2.0 was used.

B. Appendix: Experimental Results

B.1. Effect of Filter Parameters

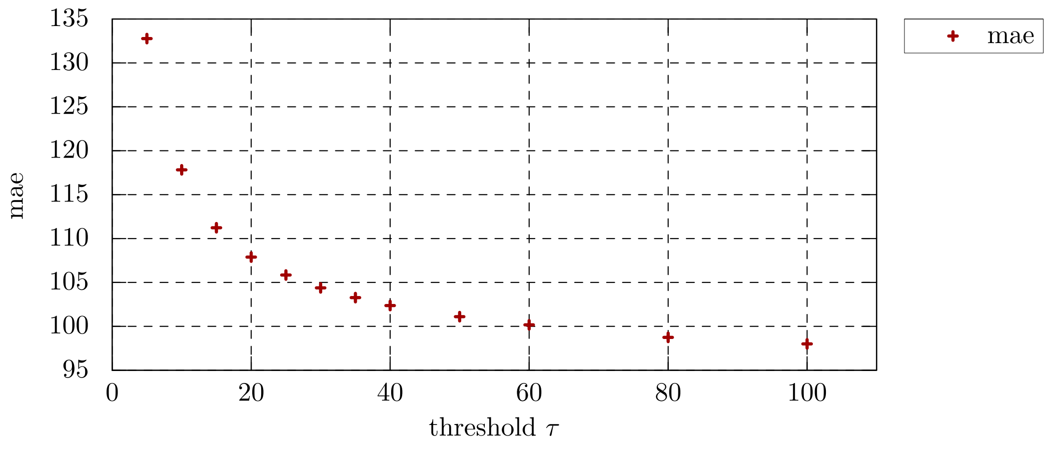


Figure B.1.: Mean absolute error according to equation (3.1) of the proposed texture filter for various thresholds τ.

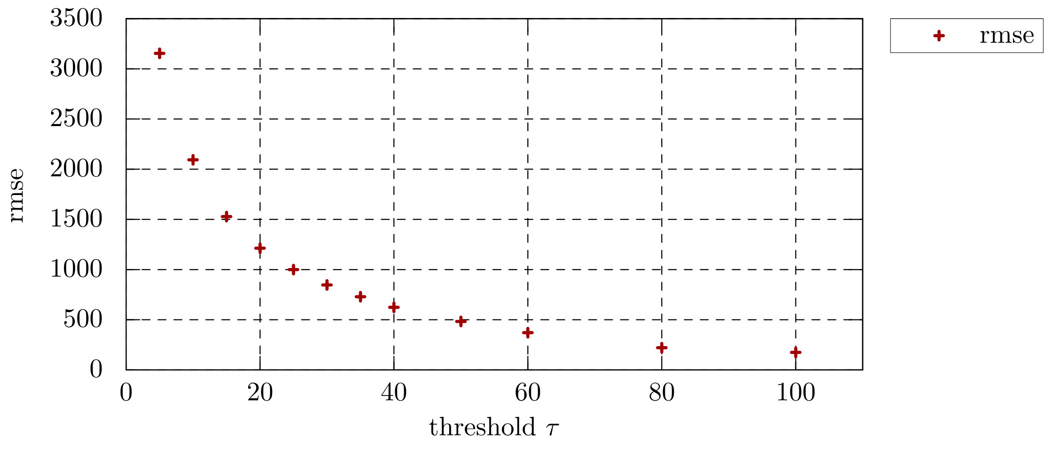


Figure B.2.: Root mean square error according to equation (3.2) of the proposed texture filter for various thresholds τ.

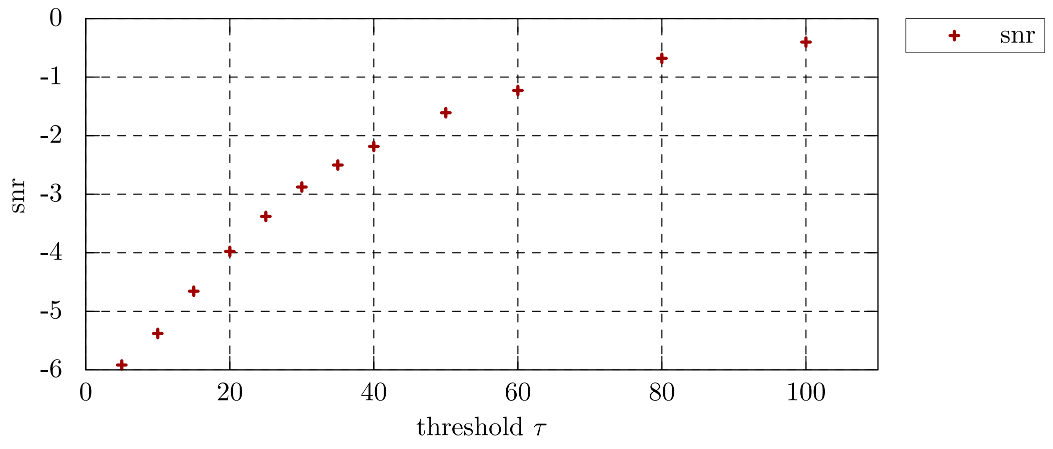


Figure B.3.: Signal to noise ratio according to equation (3.3) of the proposed texture filter for various thresholds τ.

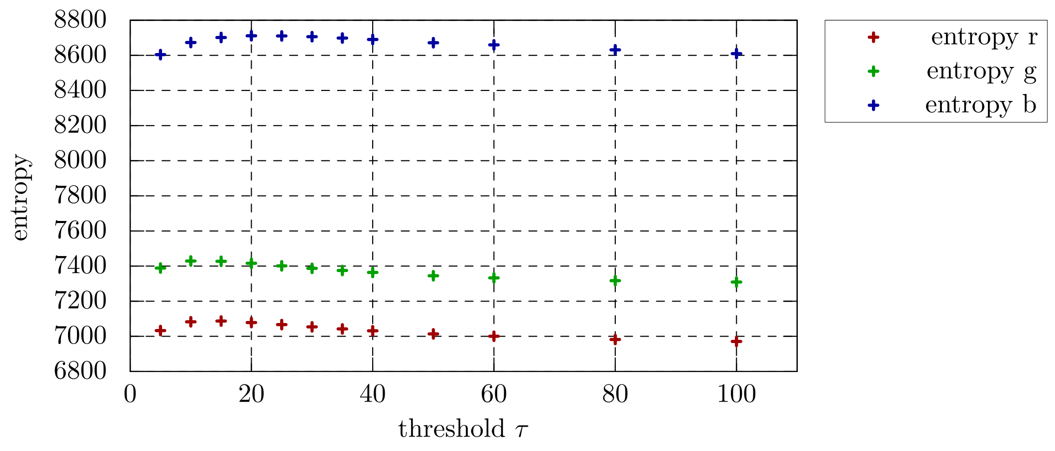
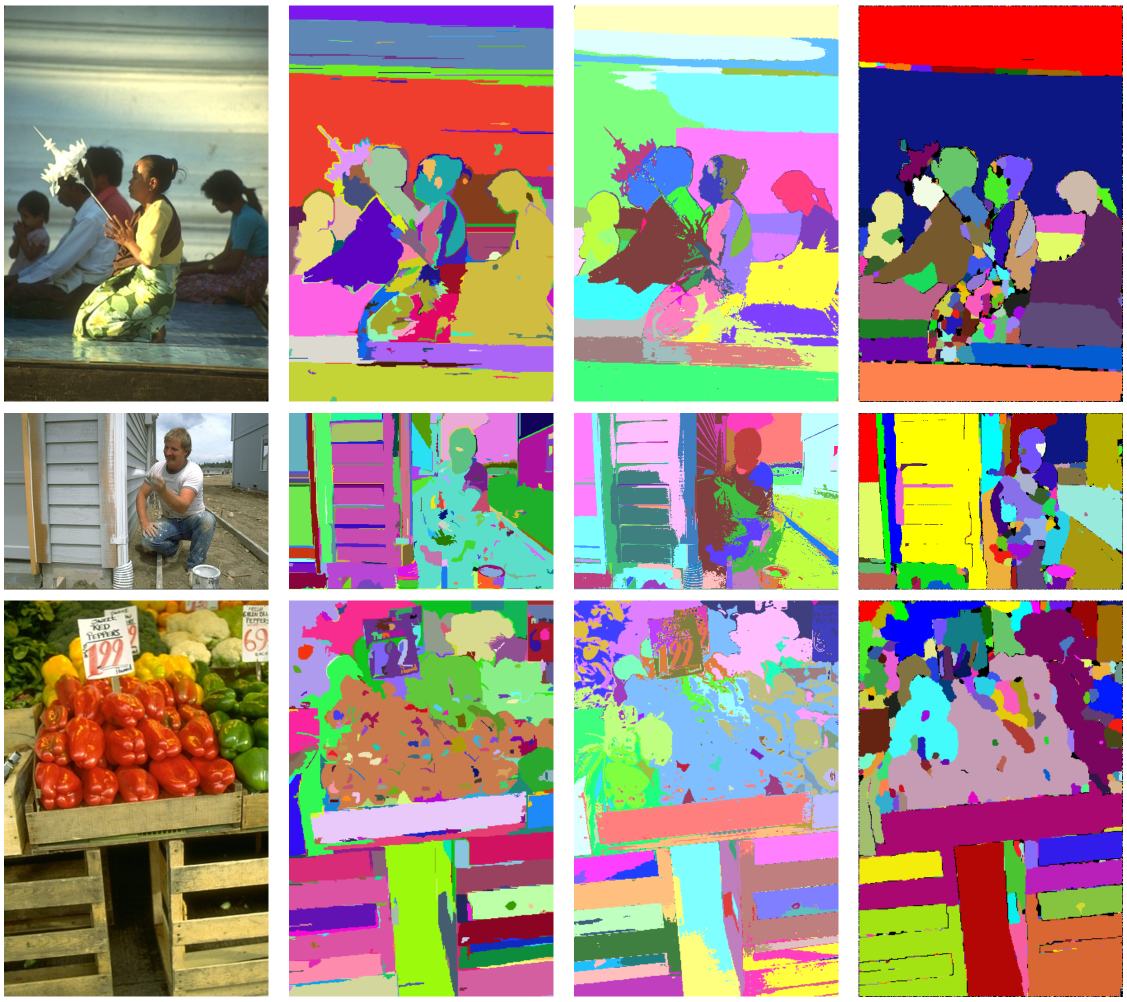


Figure B.4.: Entropy according to equation (3.4) of the proposed texture filter for various thresholds τ.

B.2. Segmentation Results



(a) (b) (c) (d)

Figure B.5.: Visual comparison of segmentation methods. B.5(a) Original image. B.5(b) Graph-based segmentation. B.5(c) Mean-shift segmentation. B.5(d) Proposed texture filter in combination with metropolis algorithm.

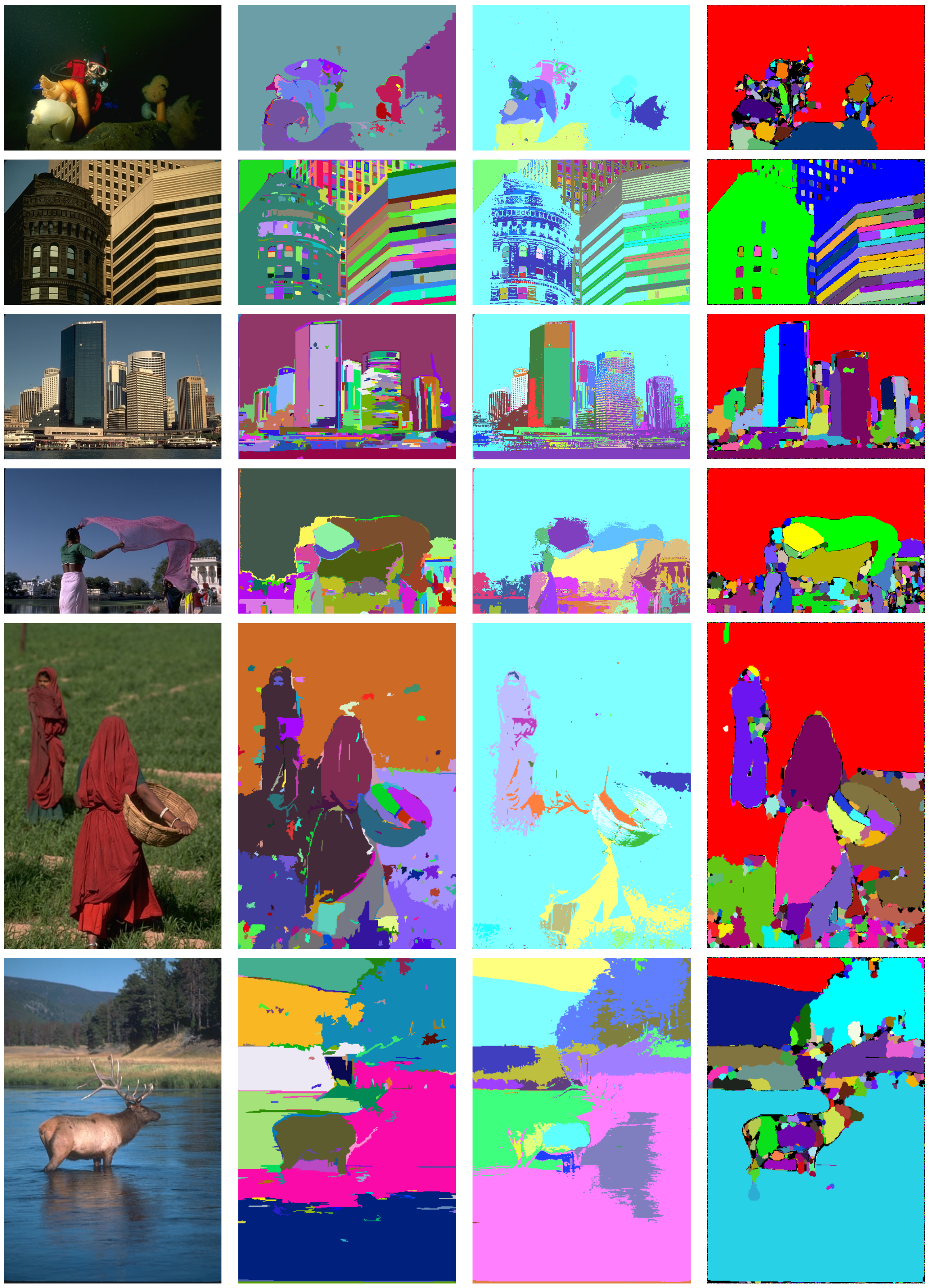


Figure B.6.: Visual comparison of segmentation methods. B.6(a) Original image. B.6(b) Graph-based segmentation. B.6(c) Mean-shift segmentation. B.6(d) Proposed texture filter in combination with metropolis algorithm.

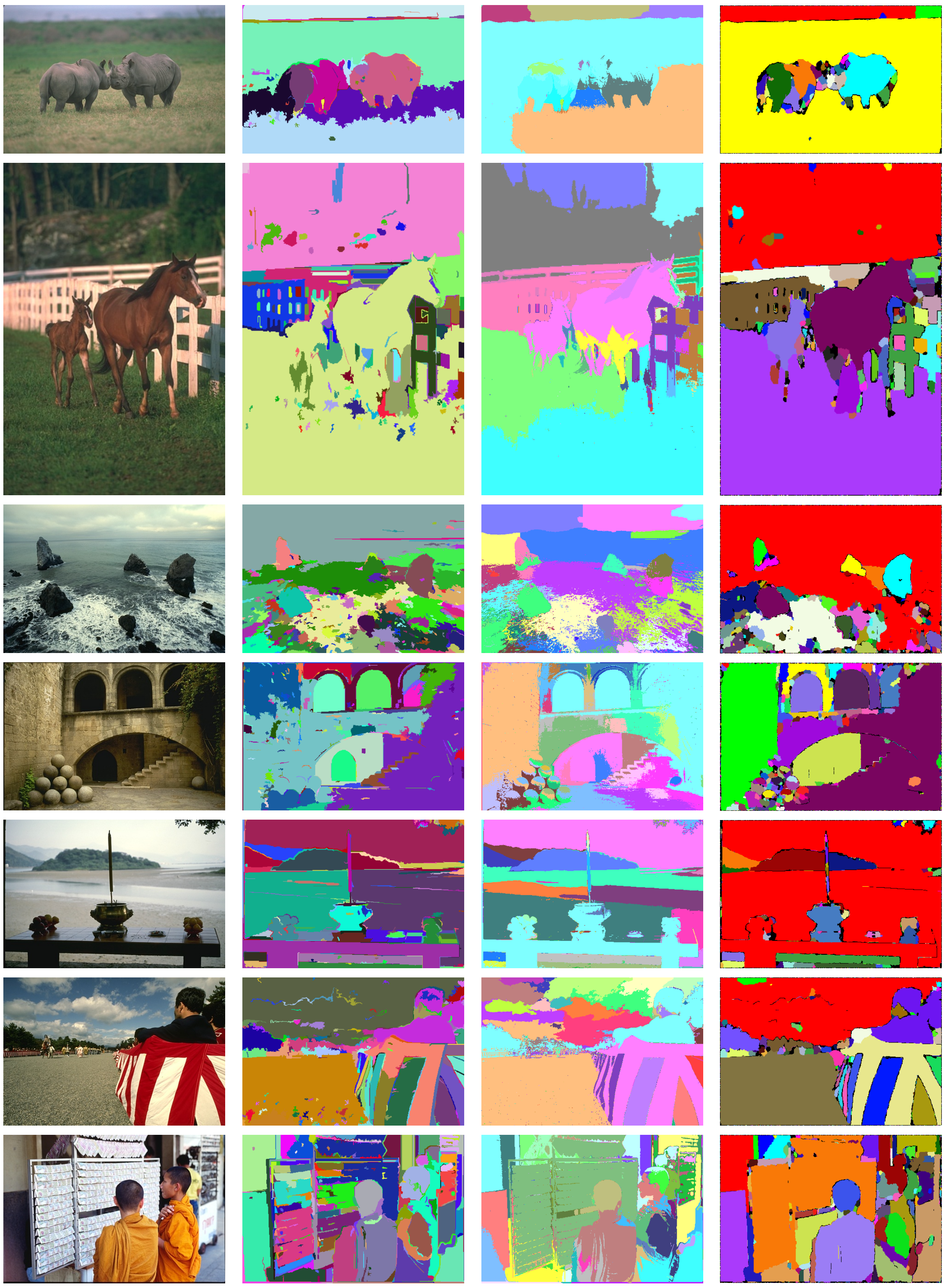


Figure B.7.: Visual comparison of segmentation methods. B.7(a) Original image. B.7(b) Graph-based segmentation. B.7(c) Mean-shift segmentation. B.7(d) Proposed texture filter in combination with metropolis algorithm.

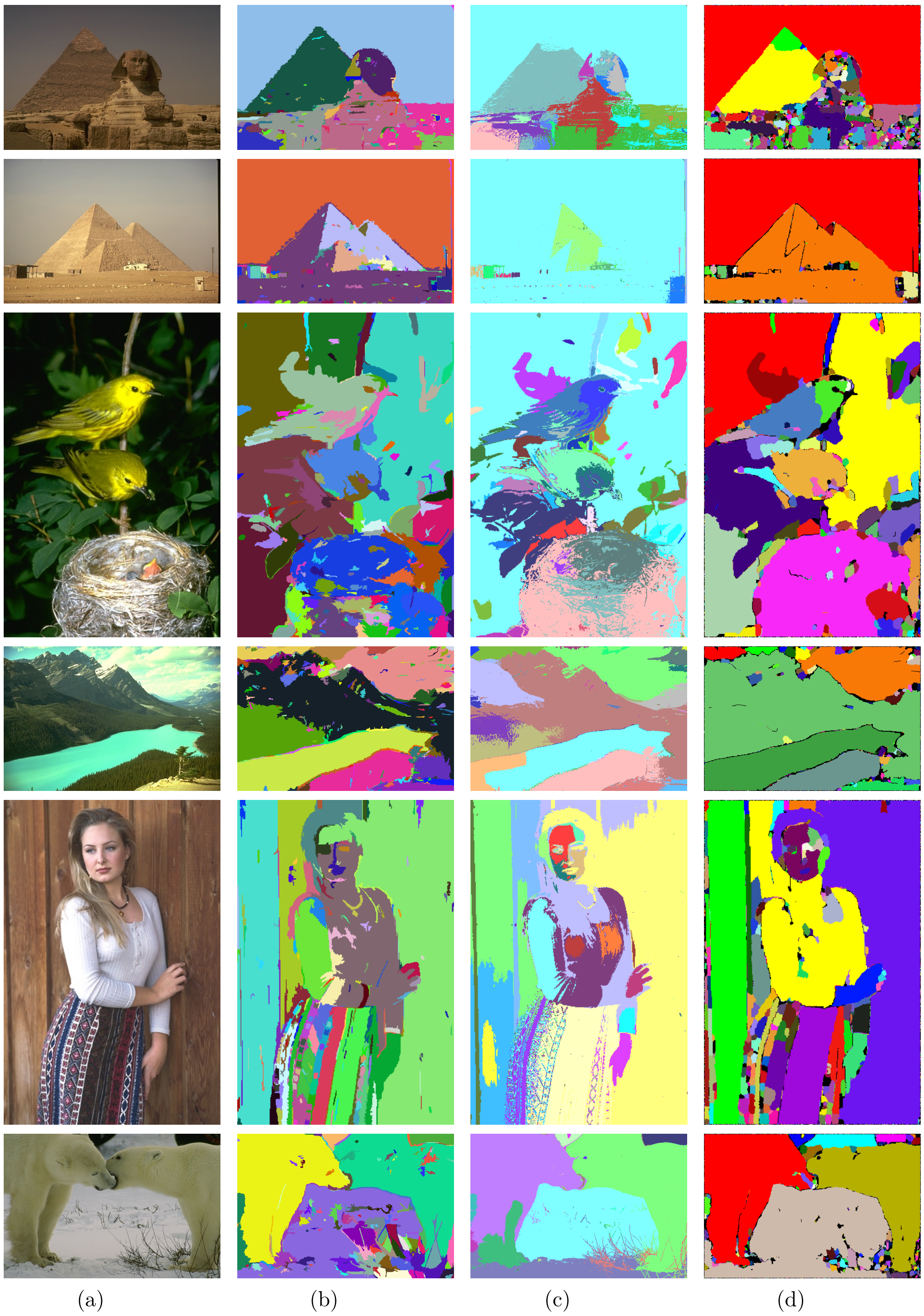


Figure B.8.: Visual comparison of segmentation methods. B.8(a) Original image. B.8(b) Graph-based segmentation. B.8(c) Mean-shift segmentation. B.8(d) Proposed texture filter in combination with metropolis algorithm.

Bibliography

A. Abramov. *Compression of the visual data into symbol-like descriptors in terms of the cognitive real-time vision system.* PhD thesis, Georg-August-Universität Göttingen, 7 2012.

A. Abramov, E. Aksoy, J. Dörr, F. Wörgötter, K. Pauwels, and B. Dellen. 3d semantic representation of actions from efficient stereo-image-sequence segmentation on gpus. In *5th International Symposium 3D Data Processing, Visualization and Transmission*, pages 1–8. University Clermont, 5 2010.

E. Aksoy, A. Abramov, F. Wörgötter, and B. Dellen. Categorizing object-action relations from semantic scene graphs. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 398 –405, 5 2010.

R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura. Improving hybrid opencl performance by high speed networks. In *First International Conference Networking and Computing*, pages 262 –263, 11 2010.

I. Asimov. *I, Robot.* Gnome Press, 12 1950.

I. Beichl and F. Sullivan. The metropolis algorithm. *Computing in Science Engineering*, 2(1):65 –69, 1 2000.

S. Chen and T. Shih. On the evaluation of edge preserving smoothing filter, 1995.

Y. Cheng. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):790 –799, 8 1995.

D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24 (5):603 –619, 5 2002.

Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs (Applications of GPU Computing Series).* Morgan Kaufmann, 1 edition, 9 2012.

Richard Cowen. *History of Life.* Wiley-Blackwell, 4th edition, 1 1991.

L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science Engineering*, 5(1):46 –55, 1-3 1998.

C. Darwin. *On the Origin of Species by Natural Selection.* London: Murray, 2 1859.

E.S. Devi, S. Sasikumar, S. Selvakumar, M.B. Bose, and T. Elizabeth. Performance comparative study on pca based denoising with joint demosaicing & denoising. In *IEEE International Conference on Computational Intelligence and Computing Research*, pages 1 –4, 12 2010.

W. Du, X. Tian, and Y. Sun. A dynamic threshold edge-preserving smoothing segmentation algorithm for anterior chamber oct images based on modified histogram. In *4th International Congress on Image and Signal Processing*, volume 2, pages 1123 –1126, 2011.

C. Eckes and J. Vorbrüggen. Combining data-driven and model-based cues for segmentation of video sequences. In *In Proc. World Congress on Neural Networks*, pages 868 –875, 1996.

C.Y. Fah, O.M. Rijal, and N.M. Noor. Image quality and noise evaluation. In *Seventh International Symposium on Signal Processing and Its Applications*, volume 1, pages 465 – 468, 7 2003.

Rob Farber. *CUDA Application Design and Development.* Morgan Kaufmann, 1st edition, 11 2011.

P. Felzenszwalb and D. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59:167–181, 2004.

K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32 –40, 1 1975.

M. Grundmann, V. Kwatra, M. Han, and I. Essa. Efficient hierarchical graph-based video segmentation. In *Computer Vision and Pattern Recognition*, 2010.

B. Gunturk. Fast bilateral filter with arbitrary range and domain kernels. In *17th IEEE International Conference on Image Processing*, pages 3289 –3292, 9 2010.

V. Hedau, H. Arora, and N. Ahuja. Matching images under unstable segmentations. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1 –8, 6 2008.

Zhu L., Y. Zhu, H. Mao, and M. Gu. A new method for sparse signal denoising based on compressed sensing. In *Second International Symposium on Knowledge Acquisition and Modeling*, volume 1, pages 35 –38, 12 2009.

A. Lev, S. Zucker, and A. Rosenfeld. Iterative enhancemnent of noisy images. *IEEE Transactions on Systems, Man and Cybernetics*, 7(6):435 –442, 1977.

N. Li and J. DiCarlo. Unsupervised Natural Visual Experience Rapidly Reshapes Size-Invariant Object Representation in Inferior Temporal Cortex. *Neuron*, 67(6): 1062 –1075, 09 2010.

S. Liu, G. Dong, C. Yan, and S. Heng Ong. Video segmentation: Propagation, validation and aggregation of a preceding graph. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1 –7, 6 2008.

D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th International Conference on Computer Vision*, volume 2, pages 416–423, 2001.

D.R. Martin, C.C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):530 –549, 2004.

Merriam-Webster. *Merriam-Webster's Collegiate Dictionary.* Merriam-Webster, Inc., 11th edition, 4 2008.

M. Muneyasu, T. Maeda, T. Yako, and T. Hinamoto. A realization of edge-preserving smoothing filters using layered neural networks. In *IEEE International Conference on Neural Networks*, volume 4, pages 1903 –1906, 1995.

NVIDIA. *NVIDIA CUDA C: Programming Guide.* NVIDIA, 4.2 edition, 4 2012.

J. Papon, A. Abramov, E. Aksoy, and F. Wörgötter. A modular system architecture for online parallel vision pipelines. In *IEEE Workshop on Applications of Computer Vision*, 1 2012.

S. Paris and F. Durand. A topological approach to hierarchical segmentation using mean shift. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1 –8, 6 2007.

K. Pauwels, N. Krüger, M. Lappe, F. Wörgötter, and M. Van Hulle. A cortical architecture on parallel hardware for motion processing in real time. *Journal of Vision*, 10, 2010.

R. B. Potts and C. Domb. Some generalized order-disorder transformations. *Proceedings of the Cambridge Philosophical Society*, 48:106, 1952.

S. Reich, A. Abramov, J. Papon, F. Wörgötter, and B. Dellen. A novel real-time, edge-preserving smoothing filter. In *VISAPP*, 2013. submitted.

C. J. Van Rijsbergen. *Information Retrieval.* Butterworth-Heinemann, 2nd edition, 3 1979.

Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley Professional, 1st edition, 7 2010.

C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision*, pages 839 –846, 1998.

A. Vazquez-Reina, S. Avidan, H. Pfister, and E. Miller. Multiple hypothesis video segmentation from superpixel flows. volume 6315 of *Lecture Notes in Computer Science*, pages 268 –281. Springer Berlin / Heidelberg, 2010.

C. Venditti, A. Meade, and M. Pagel. Phylogenies reveal new interpretation of speciation and the Red Queen. *Nature*, 463(7279):349–352, 12 2009.

Visapp. Homepage. http://visapp.visigrapp.org/, 9 2012.

Q. Yang, K. Tan, and N. Ahuja. Real-time o(1) bilateral filtering. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 557 –564, 6 2009.

Q. Yang, S. Wang, and N. Ahuja. Svm for edge-preserving filtering. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1775 –1782, 2010.

A. Zhuravlev and R. Riding, editors. *The Ecology of the Cambrian Radiation.* Columbia University Press, 10 2000.

Erklärung nach §18(8) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 24. September 2012

(Simon Martin Reich)