LEHRSTUHL FÜR REALZEIT-COMPUTERSYSTEME
TECHNISCHE UNIVERSITÄT MÜNCHEN
UNIV.-PROF. DR.-ING. G. FÄRBER

Design and Implementation of an FPGA-based Stereo Vision System for the EyeBot M6

Benedikt Dietrich

Diplomarbeit

Design and Implementation of an FPGA-based Stereo Vision System for the EyeBot M6

Diplomarbeit

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr.-Ing. Georg Färber

Executed Department of Electrical Engineering
University of Western Australia

Advisor: Dipl.–Ing. Martin Schäfer

Author: Benedikt Dietrich
Daudetstr 10a
81245 Munich

Submitted in February 2009

Acknowledgements

At first I would like to thank A/Prof. Dr. Thomas Bräunl for offering the opportunity to work on this project and making the stay in Australia possible.

Many thanks to Prof. Dr. Färber and Dipl. Ing. Martin Schäfer who supported me in my intention to do my diploma thesis abroad and helped me in my preparations of the proposal to have the DAAD scholarship.

Without the financial help of the DAAD (German Academic Exchange Service) this semester abroad would have not been possible. Therefore, I would like to thank for the great support.

I would like to thank Martin Geier and Azman M. Yusof for their support during the project and the many discussions which brought up a lot of good ideas.

Many thanks to all who helped proof-reading this thesis in the end. Especially I would like to thank Susi and Victoria who spent many hours on this task.

Finally, I would like thank my parents who supported me in all my plans and helped me pulling through my intentions.

Perth, February 2009

Contents

List of Figures

List of Tables

List of Symbols

BGA	Ball Grid Array
CPU	Central processing unit
EOL	End of Line
EOF	End of Frame
FIFO	First In First Out Memory
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
I2C	Inter-Integrated Circuit
LCD	Liquid crystal display
LSB	Least Significant Bit
LUT	Look-Up-Table
MSB	Most Significant Bit
NCC	Normalized Cross-Correlation
PAR	Place and Route
PCB	Printed Circuit Board
PWM	Pulse-Width Modulation
RCS	Lehrstuhl für Realzeit-Computersysteme
SAD	Sum of Absolute Differences
SLAM	Simultaneous Localization and Mapping
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Static Random Access Memory
SSD	Sum of Squared Differences
USB	Universal Serial Bus
UWA	University of Western Australia
VHSIC	Very-High-Speed Integrated Circuits
VHDL	VHSIC Hardware Description Language
VLIO	Variable Latency Input/Output

Abstract

The goal of this thesis is the design and implementation of an FPGA based stereo vision system on the EyeBot M6. The research platform named EyeBot M6 as the successor of M4 is built to allow the realization of stereo vision tasks for small robots. Stereo vision uses two cameras separated by distance and a process called triangulation to determine the depth information of the environment. Triangulation requires the knowledge where a point of the left camera image can be found inside the right image. The search space of the problem is huge and therefore a lot of computational power is required. However, the CPU of the EyeBot M6 is meant to deal later on with higher computer vision and control tasks only. Therefore, the main challenge of this thesis will be the implementation of such a stereo vision system on the fairly small FPGA of the EyeBot M6, a Spartan 3E 500. Due to the given limited resources high optimization and resource sharing will be necessary to fit major parts of the image processing for stereo vision into the FPGA. Two processing steps have been developed during this thesis.
With the help of rectification it is possible to reduce the search space for correlating points inside the pictures from 2D to 1D. This process removes distortion and alignment errors of the cameras that are directly connected to the FPGA. Due to the fact that the alignment is not fixed and depends on the application area of the EyeBot a fully automated toolbox has been developed which guarantees the synthesis of an optimized rectification unit at the end. Due to the lack of memory and therefore the missing possibility to store look up tables, the rectification itself is achieved with polynomials of third degree. These polynomials regress the undistortion and rectification functions gained with the help of a stereo camera calibration. Based on the results of the polynomials an address is calculated and passed to an SRAM controller. This controller returns four pixels for every coordinate pair which are then interpolated in stream to form rectified images.
A further reduction of the search space can be achieved with a feature based approach. The Harris Corner Detector returns a list of features with a high uniqueness for both images which then can be correlated. The quantity of depth information is highly reduced by this approach, but the calculation of a dense disparity map had to be dropped because of the given limited resources. The Harris has been implemented for the mono case first to later merge two Mono Harris to a Stereo Harris Corner Detector by sharing resources whereever possible.
The thesis closes with the description of the resulting hardware system and utilized software approach to finally match the features and therefore gain the depth information of the environment.

1 Introduction

Computer vision is the science and technology to learn machines to see. Huge progress has been achieved by theoretical research of the last decades. More and more complex algorithms have been developed to allow accurate realizations of tasks like stereo vision, object tracking or face recognition. The theoretical research is only one part which has to be solved. Because of the computational complexity of the algorithms powerful CPUs are required to achieve acceptable performance. However, to allow the appliance of such algorithms in every-days life the algorithms have to be real-time capable on affordable embedded systems with a low power consumption. The research of the last years has shown that many of the low level image pre-processing and processing parts which consume most of the CPU time can be highly efficient implemented in hardware. In the field of research FPGAs entered the world of computer vision to be utilized as hardware-accelerators for economical CPUs.
This thesis is based on the EyeBot M6 platform which is equipped with a FPGA as well. This platform has been designed at the Department of Electrical Engineering at the University of Western Australia. At first the EyeBot M6 is described to explain afterwards the goal of this thesis in section 1.2.

1.1 Eyebot M6

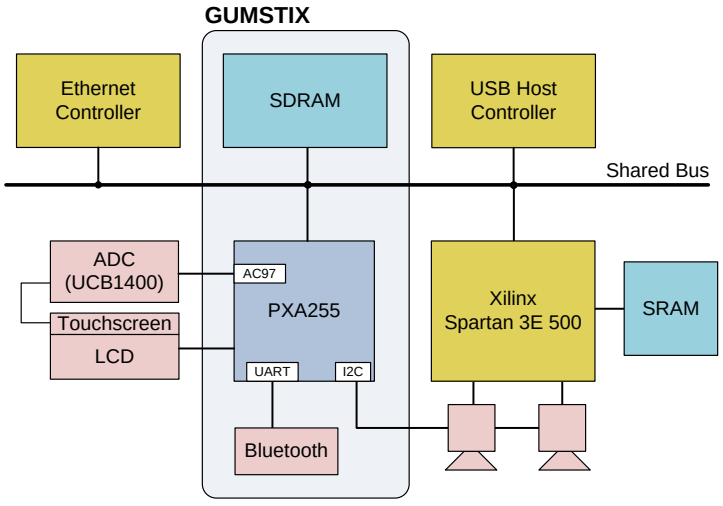


Figure 1.1: EyeBot M6 - System Overview

The heart of the EyeBot M6 is a PXA-255 running at 400MHz. To avoid BGA components a Gumstix Board [11] embedding the processor is used. This Gumstix Board is equipped with 64MB of SDRAM, 16MB of flash and a bluetooth module. The board is interconnected with the other components of the EyeBot M6 with help of an asynchronous bus interface called Variable Latency I/O (VLIO) as it can be seen in figure 1.1.
To free up the CPU from image processing tasks a Xilinx Spartan 3E 500 [32] has been utilized which is connected via the same VLIO interface to the Gumstix. The Spartan 3E comes with

- 9312 LUTs
- 20 block RAMs
- 20 Multipliers 18x18
- 158 I/O pins

As many as possible of the necessary image processing parts should be implemented within the Spartan. Therefore the Spartan is directly connected to two color cameras named OV6630 from OmniVision [26]. The cameras have a maximum resolution of 352×288 and deliver pictures with a maximum frame rate of 50 fps. They will be operated in the 8bit mode for this thesis and so with a clock rate of 18 MHz. For the image processing implemented during this thesis the YUV mode of the cameras has been chosen. All of the algorithms are based on grayscale images and by using the YUV mode the Y part of the incoming data can be directly used.
In general images, which are in our case of the size 99 kbytes (grayscale), are not stored inside the FPGA to avoid wasting precious block RAM. Therefore a common soultion is to connect external memory to the FPGA. For the EyeBot this is an SRAM with 18 Mbit which is connected directly to the FPGA and can be interfaced with 100 MHz. With memory of 18 Mbit it is possible to store 10 frames of full color inside the SRAM. Additionally the following components can be connected to the FPGA

- 14 servos
- 4 motors controlled by a FPGA generated PWM
- 4 encoders for a motor feedback
- 6 position sensing devices

At the time I started developing on the EyeBot M6 the interface between FPGA and CPU, Camera and SRAM was figured out to be unstable. Martin Geier worked on this problem for his thesis [10] and developed a framework for my image processing algorithms. The resulting interface between the FPGA and the CPU achieved a transfer rate of approximately 25 Mbyte/s which is equivalent to 247 fps, but without any processing of the CPU. Due to the fact that all components share one bus this number is drastically reduced in reality. Beside the already mentioned components the EyeBot M6 comes with

- AC97 audio and touchscreen controller

- LCD touchscreen
- USB Host Controller
- Ethernet Controller
- Infrared Sensor

A full linux operating system of the version 2.6.17 is running on the EyeBot M6. A program called M6 Main is used as a monitor program including a file browser which allows the user e.g. to start his programs directly from the touchscreen or request the status of the system. This program however was not used for the performance evaluations done in chapter 7 because it is still under development. The status of the touchscreen for example is at the moment requested by simple polling and therefore highly decreases the performance.

1.2 Project Scope

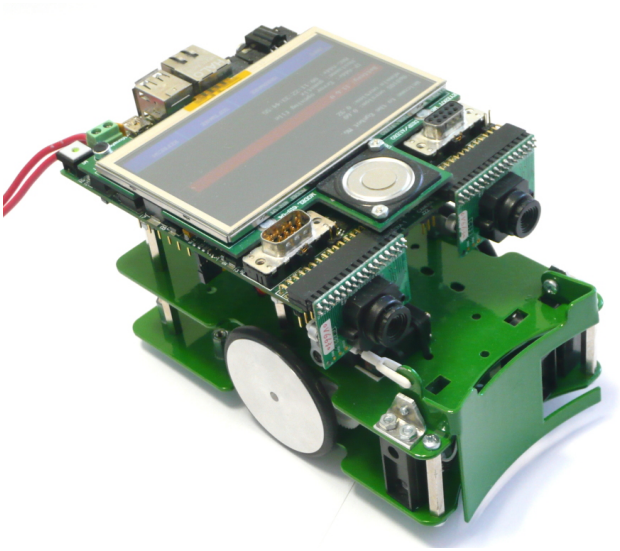


Figure 1.2: EyeBot M6 - Robot Platform

The primary goal of this thesis is the design and implementation of a stereo vision system on the EyeBot M6. The solution has to be suitable for the navigation of the robot shown in picture 1.2. Furthermore it has to be analyzed if the EyeBot with the current stereo rig constellation can be used for basic control tasks in the automotive area as well.
To achieve distance measurements in real-time the FPGA has to be utilized to free the CPU from time consuming image pre-processing tasks as e.g. the rectification of images. Appropriate algorithms have to be found and their suitability for the EyeBot M6 have to be evaluated. The main challenge will be the implementation of as much image processing tasks as possible with the given limited resources of the Spartan 3E. Especially the small

amount of block RAM, a single SRAM and the small number of multipliers force the design to be highly optimized and limit the choice of possible algorithms. Additionally a high portability, modularity and reusability has to be guaranteed to ease further development on the EyeBot M6.

1.3 Thesis Outline

Chapter 2 explains how basic calculations can be implemented inside the FPGA. Due to a missing FPU a transformation from the common floating point representation to a fixed point representation has to be performed.

Chapter 3 describes the given stereo rig and the basic theory necessary to gain depth information from the given camera setup. Furthermore errors naturally produced by lens distortions and misalignment of the cameras are discussed and a solution for the rectification is presented.

Chapter 4 deals with the hardware implementation of the rectification process and the efforts which have been made to keep the resulting FPGA footprint as small as possible.

Chapter 5 introduces an additional pre-processing to reduce the later on necessary CPU time for determining a depth map out of the rectified pictures. With the help of the Harris Corner Detector features inside the pictures are located to reduce the search space. In this chapter the theoretical background of the approach is given.

Chapter 6 documents the implementation details of the Harris Corner Detector inside the FPGA. At the end of the chapter the described hardware design is merged to a Stereo Harris Corner Detector in order to share resources whereever possible.

Chapter 7 finally joins the together developed systems together. It is described how all developed hardware modules are combined to a hardware image processing unit and how then a so called disparity map is gained inside the CPU with the pre-processed data from the hardware image processing unit.

Chapter 8 summarizes the work done within this thesis to lead over to recommendations for Future Work in **Chapter 9**.

2 Floating Point to Fixed Point Conversion

2.1 Basic Design Flow

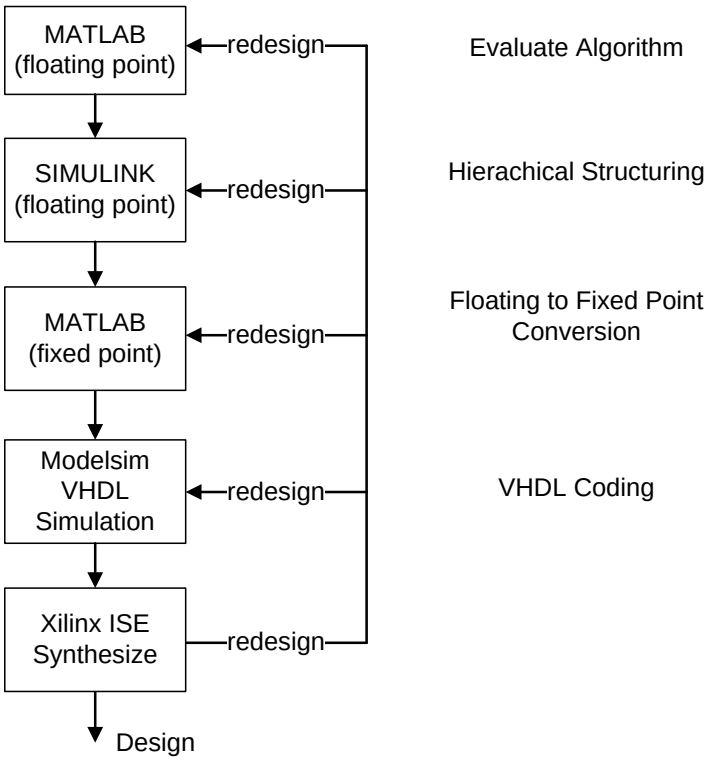


Figure 2.1: Design Flow from MATLAB to a synthesizable Design

Task of this thesis is as already mentioned the development of a stereo vision system on the EyeBot M6. The processing of the images should be done in hardware as far as possible to save precious CPU time. However, the complexity of the necessary algorithms for image preprocessing demands deep understanding and a lot of planning before the step from the simulation environment into the hardware design can be done. To avoid time-consuming redesigns due to errors in the early stage of designing, the algorithms have been developed according to the design flow in figure 2.1.

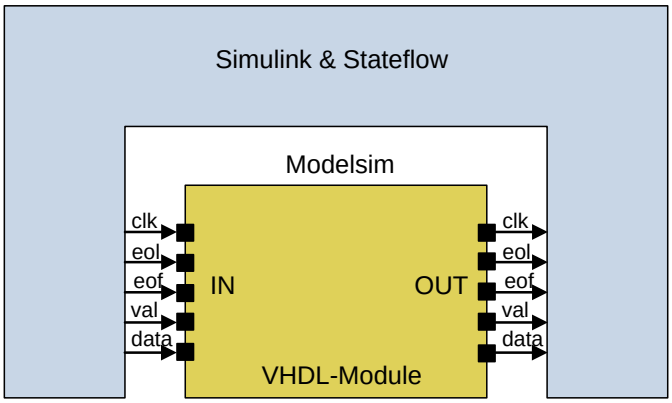


Figure 2.2: Modelsim embedded in Simulink and Stateflow

The basic development and evaluation of algorithms, which have been taken in consideration for this thesis, has been done in MATLAB [1]. Once an algorithm proved functionality the step towards a hierarchical model has been achieved with the development of a Simulink model. Simulink was chosen because it offers the possibility to design a system in the same hierarchical way as the resulting hardware. Early design errors can be avoided by simulating the built hierarchy and verifying ports and necessary signals between modules. Simulink furthermore offers the possibility to create a link to a VHDL-Simulator called Modelsim [2]. This simulator has been used to validate all later on programmed VHDL-modules. The connection between Simulink and Modelsim has been evaluated, with good results. With this interface it is possible to replace modules inside Simulink by real VHDL code, feed necessary data into Modelsim and read back the results. If post place and route (PAR) simulation has been performed the same results are gained as they would be produced later on in the real hardware. The main advantage of this procedure is to save time which otherwise would have to be spent on the development of test benches. The simulation results of Modelsim can be directly compared in Simulink with the data produced by the Simulink-modules. Figure 2.2 illustrates how Modelsim is embedded in Simulink and shows the resulting data flow. However, with a growing complexity of the modules these simulations become extremely slow and debugging of the modules requires high patience. Especially the introduction of line delays inside the models (see section 6.2.1) and the creation of the image stream signals as described in section 6.1 with the help of Simulink Stateflow decreased the benefits of this procedure a lot due to high simulation times. For that reason this method has been dropped and Modelsim on its own was used for simulating the VHDL-modules. Simulink was then mainly used for creating concepts and testing the models in respect to functionality and structuring.
Once a working design has been developed with the help of MATLAB and Simulink the next step is the floating to fixed point conversion. Simulink calculates, if not specified, based on floating point accuracy what is not possible inside our FPGA. Before a hardware description can be programmed all floating point calculations have to be transformed to a fixed point representation. For this transformation step the basic design flow has been ex-

tended with additional steps. First of all the developed hierarchy gained by modelling the system in Simulink is mapped back into MATLAB where bit-true fixed point calculations can be simulated much faster than in Simulink. Within MATLAB the transformation to a fixed point design is done according to section 2.5. As soon as all calculations are transformed it is easy to extend the existing floating point Simulink model in a way that the found fixed point settings are used. This step is only done for further verification before the actual VHDL code is programmed within Modelsim. After the VHDL code is verified in Modelsim the hardware is synthesized and again simulated in Modelsim with a post place & route simulation before the bitstream finally is loaded into the FPGA.
Before we now step into the implemented vision algorithms we first need to understand how calculations are done inside a FPGA. Thus the fixed point representation is introduced and the process of transforming floating to fixed point and it's challenges are described in the next sections.

2.2 Fixed Point Representation

First of all the concept of fixed point numbers will be described briefly by taking a look at a binary number B of length n.

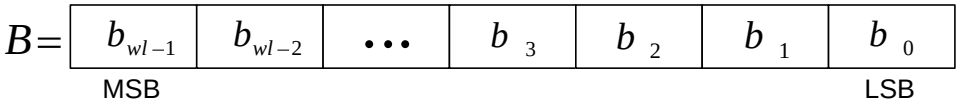


Figure 2.3: Binary Number Representation

b_{wl-1} is called the most significant bit (MSB) and b_0 the least significant bit (LSB) and wl is defined as word length. B can represent an integer number in the range of 0 to $2^{wl} - 1$ if unsigned or for a signed number from -2^{wl-1} to $2^{wl-1} - 1$. A signed number uses the MSB as sign bit and is represented in two's complement. However, for many of the calculations the resolution given with the integer representation is not sufficient. We need fractional parts of numbers. A common way is setting a fixed point within B.

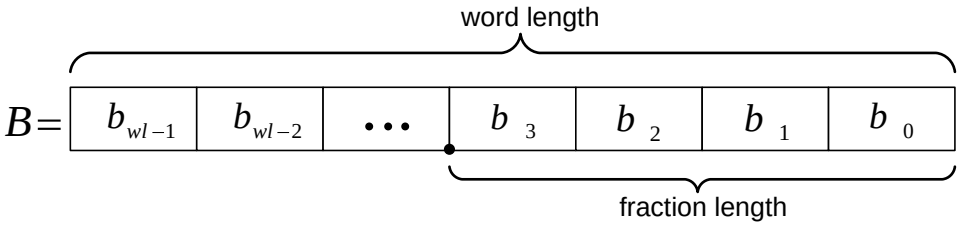


Figure 2.4: Fixed Point Number

For this example the fraction length fl is four so the binary point is four bits left of the LSB and the resulting resolution of the fractional number is 2^{-4}. To get back from a fixed

point representation to a real world value we use the equation $x = (B)_{10} \cdot 2^{-fl}$.
The fraction length can be both, positive and negative. If the fraction length fl is positive the fraction point is shifted to the left of the LSB. The resolution is increased but the range becomes smaller. Whereas if the fraction length is negative the point is shifted to the right hand side of the LSB what results in a bigger representable range but a decreased resolution. There are different ways to describe a fixed-point number e.g. with a signed attribute s, a word length wl and a fraction length fl. Another possibility is to use the integer word length iwl instead of the word length. Throughout this thesis the following notation will be used:

- `ufix` defines an unsigned fixed point number. Therefore all bits are used to represent a number
- `sfix` is a signed fixed point number. The MSB is used as sign bit and the rest of the bits for representing the number in two's complement
- The suffix `[wl,fl]` describes a number with word length `wl` as total number of bits and a fraction length of `fl`

This notation is adopted from MATLAB and is widespread. Table 2.1 gives some examples of the representation, their minima and maxima and the resulting resolution. The first two representations describe the normal unsigned and signed integers. The third example describes a fixed point number with a resolution of 2^{-5} and the fixed point 5 positions left of the LSB. The fourth number's fixed point is shifted to the right hence the range is increased but precision is lost. The last example points out that the fraction length can as well be bigger than the word length.

representation	min	max	resolution
`ufix[12,0]`	0	$2^{12} - 1$	1
`sfix[10,0]`	-2^{10-1}	$2^{10-1} - 1$	1
`ufix[5,5]`	0	0.9688	2^{-5}
`sfix[10,-5]`	-16384	16352	32
`ufix[10,15]`	0	0.0312	$3.0518 \cdot 10^{-05}$

Table 2.1: Examples for the used Fixed Point Notation

A fixed point number has a fixed range. This range is given by

$$B_{signed} \in \left[-2^{wl-fl-1}; 2^{wl-fl-1} - 2^{fl}\right]$$
$$B_{unsigned} \in \left[0; 2^{wl-fl} - 2^{fl}\right]$$

To avoid overflows and underflows during calculations it is important to understand how the ranges are changed by mathematical operations.

2.3 Fixed Point Calculations

Let us assume that we stick to the simple binary number representation with a fraction length of zero and have a look at mathematical operations. The common operations are multiplications and additions of two binary numbers. The following table points out how these operations change the theoretical number of bits needed to represent the results depending on the incoming size and type of the two binary numbers which are here however assumed to be the equal.

type	operation	result
`ufix[n,0]`	$+$	$n+1$
`sfix[n,0]`	$+$	$n+1$
`ufix[n,0]`	$*$	$2n$
`sfix[n,0]`	$*$	$2n-1$

Table 2.2: Resulting Bit Width of Basic Mathematical Operations

The impact of this table on our calculations is easier to understand with help of an exemplification, the Harris Corner Detector as described in chapter 5. The theoretical bit width needed to represent the results are listed in table 2.3.

System	input[bit]	output[bit]
Camera	∞	8
Sobel	8	11
Window	11	26
Cornerness	26	52

Table 2.3: Theoretical Input and Output Widths of the Harris Detector Modules

It is fairly obvious that if we keep full precision first of all the hardware implementation will be limited very fast by the available FPGA resources and additionally long logical paths are created and consequently the maximum clock rate of our design is reduced drastically. Therefore we need to reduce the bit width of our calculations and use only the bit width needed to gain the necessary accuracy of the final output. In order to reduce the bit width for the given Harris the decimal point has to be shifted to the right for representing bigger numbers or to the left for the representation of small numbers as used by the rectification unit (see chapter 4). If we know in which range the values will move we can crop the bit positions we do not use. The resulting representation is the already described fixed point representation. In order to calculate with these numbers it is essential to take care about the position of the fixed point. A multiplication of two fixed point numbers `ufix[wl1,fl1]` and `ufix[wl2,fl2]` results in a number with a shifted fixed point and increased word length `ufix[wl1+wl2,fl1+fl2]`.

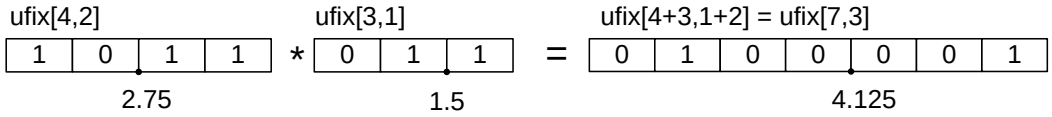


Figure 2.5: Example - Fixed Point Multiplication

The input numbers of the multiplication have to be of the same kind, signed or unsigned. If one of the numbers is of the type signed the other number has to be transformed before the multiplication is executed.
For adding two fixed points numbers they both have to be not only of the same type but also of the same fraction length. To keep the precision of the given numbers the number with the smaller fraction length has to be shifted to the left. Before doing so the word length has to be increased by the number of necessary shifts to ensure that no bits are lost.

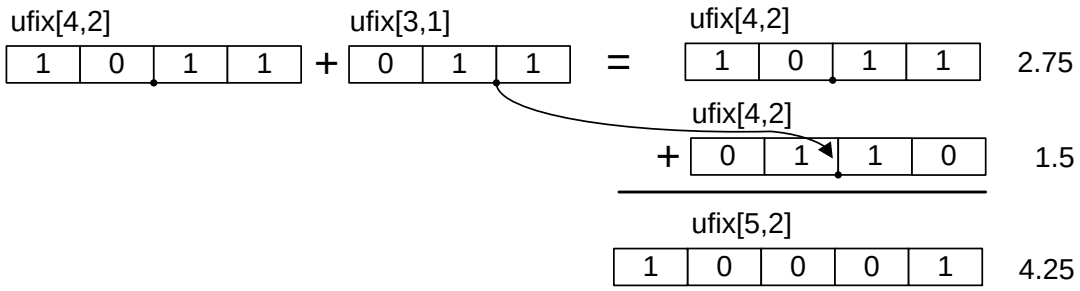


Figure 2.6: Example - Fixed Point Addition

Fixed point calculations as described are not yet part of the VHDL standard. To avoid reflecting the movement of the fixed point and necessary extensions of word length before shifting etc. for every addition and multiplication a fixed point package has been programmed in VHDL in the course of this thesis. This package is based on the numeric.std library and therefore uses the types signed and unsigned. As parameter values of the implemented arithmetical functions the input signals of the operation and their fraction length and the fraction length of the result are passed on to the functions. For every combination of signed and unsigned input and output types the according functions have been written. The functions of different input types, but same operation, are overloaded inside the package and thus have all the same name. This makes it much easier to program and is furthermore needed for the automatic package creation of the rectification unit (see section 4.2.2). In the following you can see an example function for the addition of a signed and unsigned number.

```
function plus_fp( L : signed; L_FL : integer; R : unsigned; R_FL : integer; ↩
    result_WL : integer; result_FL : integer) -- unsigned = signed + unsigned
    return unsigned is
    constant L_WL : integer := L'LENGTH;
    constant R_WL : integer := R'LENGTH;
    constant tmp_FL : integer := maximum(L_FL, R_FL); -- temporary result
```

```
        constant L_t_WL : integer := L_WL + tmp_FL - L_FL + OVERFLOW_BIT; -- WL for ↩
            temporary left
        constant R_t_WL : integer := R_WL + tmp_FL - R_FL + OVERFLOW_BIT + SIGNED_BIT; ↩
            -- ...
        constant tmp_WL : integer := maximum(L_t_WL,R_t_WL); -- WL for temporary result
        variable L_t : signed(L_t_WL-1 downto 0) := signed(shift_left(resize(L,L_t_WL),↩
            tmp_FL - L_FL));
        variable R_t : signed(R_t_WL-1 downto 0) := signed(shift_left(resize(R,R_t_WL),↩
            tmp_FL - R_FL));
        variable tmp : signed(tmp_WL-1 downto 0);
        variable result : unsigned(result_WL-1 downto 0); -- result
    begin
        tmp := L_t + R_t;
        if tmp_FL >= result_FL then
            result := resize(unsigned(shift_right(tmp, tmp_FL - result_FL)),result_WL);
        else
            result := resize(unsigned(shift_left(resize(tmp,tmp_WL + result_FL-tmp_FL),↩
                result_FL - tmp_FL)), result_WL);
        end if;
        return result;
    end function plus_fp;
```

The inputs are transformed as described above. First the word length is increased by the necessary amount of bits. Be aware that the numeric.std addition returns a number of word length MAX(L,R) and therefore does not care about overflow. This is the reason why the OVERFLOW_BIT is added to both the temporary word length L_t_WL and R_t_WL. Furthermore the unsigned number needs an additional SIGNED_BIT before the transformation to signed can be performed without loss. After both numbers have been transformed they can be simply added. The temporary result is shifted and resized to achieve the defined return_WL before it is finally returned. The usage of these functions highly simplifies the source code and the programming with fixed point numbers.
Now that the basic arithmetics are defined the question is how the word and fraction length have to be chosen in order to keep a defined accuracy and using minimum bit width at the same time. This problem is known as Floating Point to Fixed Point Conversion (FFC).

2.4 Floating Point to Fixed Point Conversion

Image processing algorithms are in general first developed with the help of tools like MATLAB or OpenCV [3]. At the very beginning the algorithms are based on floating point precision to make the design as easy as possible. Once the floating point algorithm is specified and satisfies the expectations of the developer it is necessary to transform the calculations to fixed point representation. There are two different approaches to perform this transformation:

- The **analytical approach** is based on examining the behavior of the calculations like described in section 2.3. This method is mainly applied to LTI systems as digital filters or FFT [20]. However, with a rising complexity of the system or the influence of noise the facility of this method is limited fast.

- The **bit-true simulation** makes it possible to collect statistical data of the system. The data of each quantization node is logged and the dynamic ranges of signals can be determined. These simulation results help the developer to find a solution of the FFC problem.

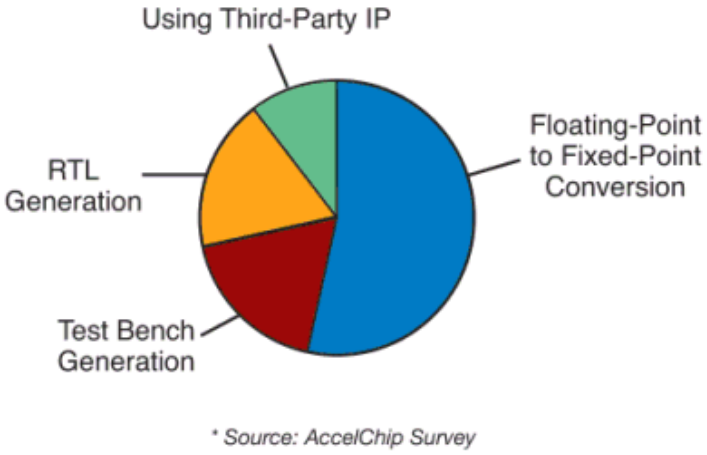


Figure 2.7: The Challenges of FPGA Design rated by Developers

Depending on the complexity but irrespective of which approach is chosen the transformation process consumes a lot of time. If done manually research has shown that up to 50% of the development time can be spent on solving this problem [30]. Figure 2.7 shows the results of a survey conducted by AccelChip Inc. [4] (now owned by Xilinx) about the difficulties of a FPGA Design. 53% identified the floating to fixed-point conversion as the most difficult part. The high complexity is not only caused by the actual algorithm's complexity, but is as well defined by the high dimensionality of the design space. The final goals of the optimization process can be accuracy, chip space, memory size, power consumption or for example the throughput of the resulting system. Furthermore the error produced by quantization is of nonlinear nature and highly depends on the stimuli and thus the evaluation of the design space can hardly be done without extensive simulation [7].
These simulations run as bit-true simulations meaning that the same behavior of binary calculations as found in the target system have to be modeled. There are fixed point bit-true simulation libraries for C++ available [18, 19] which come along with a performance reduction of one or two magnitudes [7] compared to a pure floating point program. MATLAB in combination with the Fixed-Point Toolbox offers as well the possibility to support the developer during the fixed-point transformation. It is possible to specify quantizers, calculate with fi-math objects (fixed point arithmetic objects), log data, recognize overflows or visualize data histograms of the used bit width. The big disadvantage coming with MATLAB is the low simulation speed. Compared to a C++ implementation it can amount to a factor of 1000. However, the effort of reprogramming all the MATLAB code in C++ was considered as too time-consuming. There are commercial solutions to help avoiding the reprogramming of the MATLAB code by directly creating C++ fixed-point programs out of the MATLAB m-files. One popular tool is AccelDSP Synthesis [4]. For

this thesis unfortunately none of these programs was available and another solution had to be found.
For the Harris Corner Detector, as described in chapter 5, a mixture of the analytical and bit-true simulation approach has been used. Based on statistics and experiments a working quantization was found which had an acceptable resource consumption.
The rectification of pictures as described in chapter 3 requires the computation of four polynomials of third degree in hardware, each including 9 multiplications and 10 additions which have to be quantized. With the increasing complexity of the system as the calculation of these polynomials it becomes more and more difficult and time-consuming to do the floating point to fixed point conversion by hand and based on mere statistics.
The coefficients of the third degree polynomials are the results of a stereo camera calibration and a following regression. Because of this it can happen that these coefficients change due to a rearrangement of the cameras or a modified focus. The impact of these changes on our coefficients is hard to predict because a calibration takes a long time. Additionally these changes will not alter the coefficients linearly. The complexity of this problem is increased even more by the highly non-linear behavior of the quantization error produced by the quantization of the coefficients and the calculation steps themselves.
To avoid a time-consuming reconversion in the case of camera rearrangements one possibility is to enlarge the used bit width for the calculations in a way that we definitely keep the a defined precision. However, enlarging the used bit width because of changes which might be applied is not an appropriate solution for the used FPGA. The resulting coordinate warper would consume way too many resources. Consequently for each camera constellation an optimized fixed point model would have to be derived and programmed in VHDL. To avoid this huge effort a fully automated algorithm, partially based on [27], has been developed.

2.5 Automatic Floating Point to Fixed Point Conversion

With the help of the developed automated FFC algorithm described in the following it is possible to generate an optimized fixed-point quantizer setting directly out of the results of a camera calibration (see section 3.5). Furthermore a VHDL-package is automatically generated which then can be included in the VHDL project before a new optimized rectifier is synthesized. In the following the concept of this algorithm will be presented after the introduction of the MATLAB functions used by the algorithm.

The MATLAB Fixed-Point Toolbox [1] comes with a function called `quantizer()` which returns a handle to a quantizer of the specified type. A quantization then is executed with the call of the function `quantize()` with the quantizer handle and the value which has to be quantized as parameters. As a result we get e.g. `a_q` which is returned as a floating point number. However, this floating point number can be represented with a fixed-point number of the given quantizer settings e.g. of the type `sfix[12,2]`.

```
% Example code for usage of quantizers in MATLAB
a = 12.43;
q = quantizer('fixed', 'floor', 'wrap', [12 2]);
a_q = quantize(q, a);
```

After the quantization the quantizer object enables the user to access statistics about minimum, maximum or for example occurred overflows during all appeared quantizations with this object. Common attributes of a quantizer object are 'fixed' for a signed or 'ufixed' for an unsigned fixed-point quantization. 'floor' or for example 'ceil' describe the rounding method used by the quantizer. To avoid increasing complexity of the resulting hardware simple truncation was used throughout this thesis what corresponds to the 'floor' rounding method. Additionally you can select an overflow mode and choose between 'wrap' or e.g. 'saturation'. Again the mode with the smallest overhead was used which is 'wrap'.

The developed floating point code e.g. the calculation of a polynomial of third degree is now reprogrammed in MATLAB based on the described quantizer functions. The transformed floating point function additionally gets a vector of quantizer handlers as parameter value. Based on these quantizers the calculation is performed and a fixed point results returned at the end. The transformation of the source code from a floating point design to a fixed-point design is done within two steps as described by Sanghamitra [27].

With the first step called **levelization** all calculations are transformed into single step per line calculations. For example

```
a(1:100) = b(1:100) .* c(1:100) + d(1:100);
```

becomes after levelization

```
b = in_1;
c = in_2;
temp = b(1:100) .* c(1:100);
a = temp + d(1:100);
```

With the levelization we gain a function which still calculates with floating point precision. We need to introduce a quantizer for every extension, input vector and all fixed parameters to ensure that all calculations can be represented as fixed-point numbers and thus our result will be bit-true. Therefore we now need a quantizer for every line of our levelized code.

```
b = quantize(q_1, in_1);
c = quantize(q_2, in_2);
temp = quantize(q_3, b .* c);
a = quantize(q_4, temp + d);
```

The next step described by Sanghamitra [27] called **scalarization** is not needed for my design because no internal functions of MATLAB as `sum()` for vectorized data are used.

Below the final levelized and quantized matlab code used for calculating the polynomials is listed. Altogether there are 26 quantizers (the 16 listed below and 10 for the coefficients) whose parameters have to be determined. Figure 4.4 visualizes how this equation has been modeled in Simulink and later on in hardware with the 26 quantizers.

```
function [ mapping ] = calc_fixpoint( p, quant, data )
...
% stage 1
product1 = quantize(quant(11),p(8).*Y);
product2 = quantize(quant(12),p(10).*X);
product3 = quantize(quant(13),p(7).*X);
product4 = quantize(quant(14),p(9).*Y);
% stage 2
sum1 = quantize(quant(20),p(5)+product1+product2);
sum2 = quantize(quant(21),p(4)+product3+product4);
% stage 3
product5 = quantize(quant(15),sum1.*Y);
product6 = quantize(quant(16),p(6).*X);
product7 = quantize(quant(17),sum2.*X);
% stage 4
sum3 = quantize(quant(22),product5 + p(3) + product6);
sum4 = quantize(quant(23),product7 + p(2));
% stage 5
product8 = quantize(quant(18),sum3.*Y);
product9 = quantize(quant(19),sum4.*X);
% output stage
sum5 = quantize(quant(24),p(1) + product8 + product9);
```

This function now can be called with a vector `q` of 26 quantizers and the data-points as parameter values. Bit-true values are then returned based on the quantizers defined in `q`. Now we have two versions of functions, the original floating point based function and the function based on fixed point accuracy. The next step is to determine quantizer settings so that the error between the floating and fixed point based function stays within a specified range while using a minimal bit width for the calculations. The automated determination of the quantizer settings is performed within three steps.

Calculation of value ranges determines the theoretical ranges of the calculation results and consequently uses the floating point function to ensure highest precision. With the function `calc_ranges()` we determine minima and maxima of our levelized calculation. Based on these results a vector of quantizers (for each calculation step, parameter, etc. one quantizer) is constructed. Important for this step is to gain quantizers which have the worst case accuracy but will not permit any overflow.

```
[mins maxs] = calc_ranges(polynomial, points,direction);    % with all points!!
quantizer_settings = {'floor','wrap'};

for j = 1:26
    max_val = max(abs(maxs(j)), abs(mins(j)));
    y = -1 * log2(max_val);
    if mins(j) < 0 % check if signed or unsigned needed
        q(j) = quantizer('fixed', quantizer_settings{:}, [1+1 ceil(y)]);
    else
```

```
            q(j) = quantizer('ufixed', quantizer_settings{:}, [1 ceil(y)]);
        end
    end
```

To gain the worst case quantization we set the word length to 1 (or 2 if signed) and the fraction length to $\lceil -log_2(max_{val}) \rceil$. The negative sign in front of the logarithm is needed because we use the fraction and not the exponential representation. After this step we have a list of quantizers which now can be used to call our levelized and quantized code and get back a fixed-point result based on this list of quantizers.
The current settings of the quantizers will produce the highest error but prevent the calculations from overflow. In order to judge the quality of our fixed point based calculations we need a metric which allows us to compare the fixed point with the floating point results. A popular error metric is the so called signal to quantization noise ratio (SQNR). However, for the quantization of the polynomials a more descriptive measure was used, the mean square distance in subpixels between floating point and fixed point mapping results. The function `calc_error()` gets as parameter values the floating point and fixed point results and returns the mean square error. If another error measure is wished it can easily be replaced inside this function.

As already mentioned, with the quantizer settings resulting from the calculation of value ranges we will definitely exceed every defined maximum error constraint but will not permit overflows. In the next step called **coarse quantization** we will increase the bit width of all quantizers by the same amount of bits until the maximum error conditions are met. With the used notation for the quantizers it is easy to increase their precision and to avoid overflows at the same time. By increasing both the word and fraction length by the same number of bits we keep the representable maximum value of the number and simultaneously increase the accuracy. A quantizer q_0 saved in `q` with the characterization `[wl_0,fl_0]` is therefore changed to `[wl_0+p,fl_0+p]`. The parameter p is added to all quantizers $q_0, ..., q_n$ and the resulting error is calculated with the new settings. The necessary minimum p is determined by a divide and conquer approach to reduce the execution time of our algorithm.

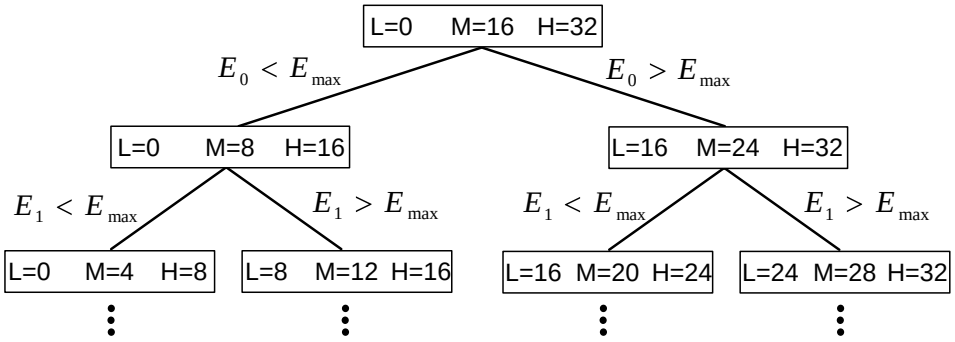


Figure 2.8: Example for Divide and Conquer Steps; L=0 and H =32

Figure 2.8 shows the first three steps of the divide and conquer algorithm. We define the range in which p is allowed to move as $p \in [L_0, H_0]$. At the beginning M_0 is set to

$\lceil \frac{1}{2}(H_0 + L_0) \rceil$. The functions with floating and fixed point accuracy are then executed with a $p = M_0$. Afterwards the error of the fixed point result is determined with the help of the `calc_error()` function. If the resulting error E_0 is bigger than the maximum error E_{max}, defined by the user, the current p is too small and L_1 for the next iteration is set to M_0, while H_1 is H_0. If the resulting error is smaller than E_{max}, H_1 is set to M_0 and L_1 to L_0. Now M_1 is calculated the same way as above with the new L_1 and H_1 and the procedure is repeated again. This is done iteratively until the difference between H_n and L_n is one. If the produced error is still bigger than the defined maximum error and M_{n+1} is constantly H_0 the constraint can not be met with a maximum bit width of H_0 and the algorithm failed. Otherwise we found a minimum p which guarantees us that the maximum error is no longer violated. As described this p is applied to all quantizers which results in a large overhead of calculation accuracy because not all quantizers need to be that accurate.

The **fine tuning** solves this problem by changing the settings of the quantizers individually and searching for an optimized setting. Due to the high simulation time of the bit true simulation a simple Greedy algorithm was used for optimization which can be easily exchanged if a better optimization is desired.
The Greedy algorithm reduces the quantizers' accuracy since we already meet our maximum error constraint after the coarse optimization. To reduce the accuracy the setting of an individual quantizer p_j given with `[wl,fl]` is changed to `[wl-1,fl-1]`. The floating and fixed point function is then called with a quantizer vector $(q_0, ..., q_j - 1, ..., q_n)$ and the error e_j is determined. This is done for every quantizer on its own and an error vector $\mathbf{E}$ is created

$$\mathbf{E} = \begin{bmatrix} e(q_0 - 1, q_1, q_2, ..., q_n) \\ e(q_0, q_1 - 1, q_2, ..., q_n) \\ \vdots \\ e(q_0, q_1, q_2, ..., q_n - 1) \end{bmatrix}$$

The element k with the smallest entry and therefore with the smallest increase of error compared to the error produced with the setting $(q_0, q_1, \ldots, q_n)$ is picked. The quantizers' settings used for the next iteration are set to $(q_0, q_1, \ldots, q_k - 1, \ldots, q_n)$. This again is done iteratively until the defined maximum error is no longer met. Our optimized quantizer settings are then the one from the step before, where the error constraint was just not violated.
The problem of Greedy is that it does not reconsider decisions made in the past and so ends up in a local minimum with a high probability. The decisions are only based on the results of the current step. Sanghamitra proposed his own algorithm [27] which however requires with higher simulation times without producing better results compared to Greedy. Therefore I retained the Greedy algorithm whose results turned out to be sufficient for my purposes.

Now that it is known how calculations are realized inside the FPGA and how to determine the fixed point representations the first and very important pre-processing step, the rectification, will be described. At first the theoretical background will be presented in the following chapter.

3 Theory of Rectification

To understand why rectification is necessary the given stereo rig constellation of the EyeBot M6 will be presented in an idealized form. This model will be extended during this chapter to get step by step closer to reality.

3.1 Parallel Stereo Camera Constellation

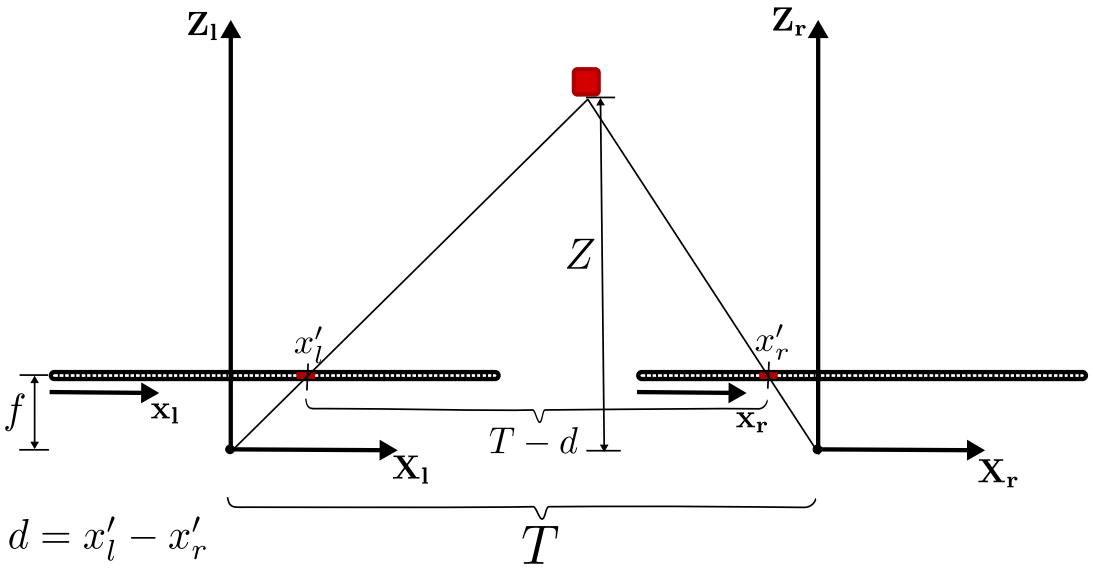


Figure 3.1: Triangulation under idealized Circumstances

The EyeBot M6 platform was designed to mount two cameras directly on the PCB. The resulting constellation is called a frontal parallel camera constellation and is shown in an idealized form in figure 3.1. With two cameras it is possible to determine depth information of the environment. In order to gain this information, images from both cameras have to be processed and correlating points have to be found (see section 7.3). Once a point of the left picture can be spatialized to a point in the right picture the depth can be easily derived with the theorem on intersecting lines applied on the given geometry in figure 3.1. Neglecting distortion and assuming perfectly row aligned cameras the points will lie in the same image line and the equation for the depth Z results in

$$Z = \frac{f \cdot T}{x'_l - x'_r}.$$

$x'_l - x'_r$ is also known as disparity d. According to this formula and due to the fact that pixels of our camera have a finite size the quality of depth information depends on the distance of the object. The closer the object is to the camera the better the resolution of depth. The depth resolution is given with the formula

$$\Delta Z \approx \frac{pZ^2}{fT}, \text{ with the pixelsize } p = \frac{chipsize}{resolution} \text{ (both in x-direction)[5].}$$

Figure 3.2 visualizes the behavior of ΔZ depending on Z and with the current setup of the EyeBot M6 stereo rig ($f = 4.9mm$, $T = 66mm$, $p = 9\mu m$). Concluding from this figure one of the goals, the usage of the EyeBot M6 with the current camera constellation inside a car, has to be dismissed. The planned vehicle tracking does not make sense with a maximum depth resolution of more than 2.5m at a distance of 10m because the occurring distances within traffic will be even higher and therefore the resolution worse. To improve the resolution the basis width T has to be increased and calculations based on subpixel accuracy have to be done. For the second task planned, the usage of the EyeBot M6 for robot tasks as e.g. the RoboCup, the objects will move inside a shorter range and thus the resulting resolution will be much higher what will allow appropriate navigation of the EyeBot in his environment based on computed disparity maps (see chapter 7.3).

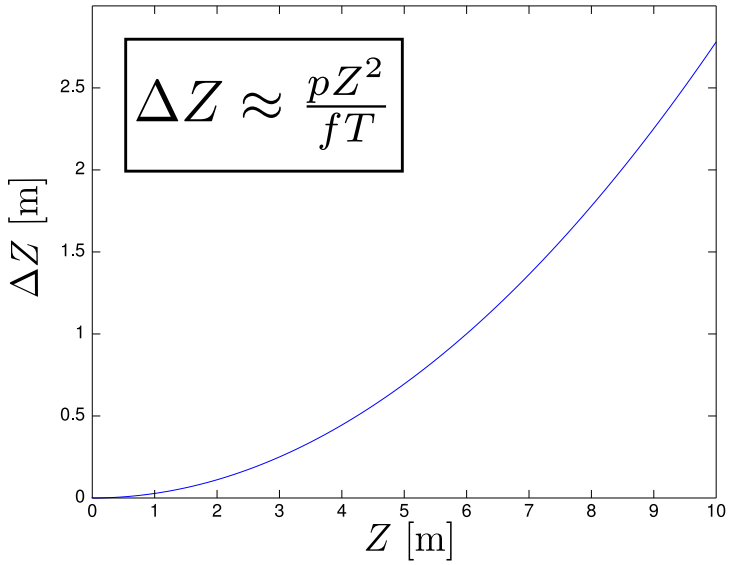


Figure 3.2: Maximum Depth Resolution, $f = 4.9mm$, $T = 66mm$, $p = 9\mu m$

However, the environment is not mapped onto our image plane one by one as assumed above. Every camera has its own typical distortion because of inaccuracy regarding the lens or e.g. the chip placement. This distortion has to be compensated in order to determine reliable disparity and depth information. In order to understand the process of undistortion, first of all the principle of central perspective will be explained in brief to extend this model later on with the distortion model used for this thesis.

3.2 Principle of central perspective

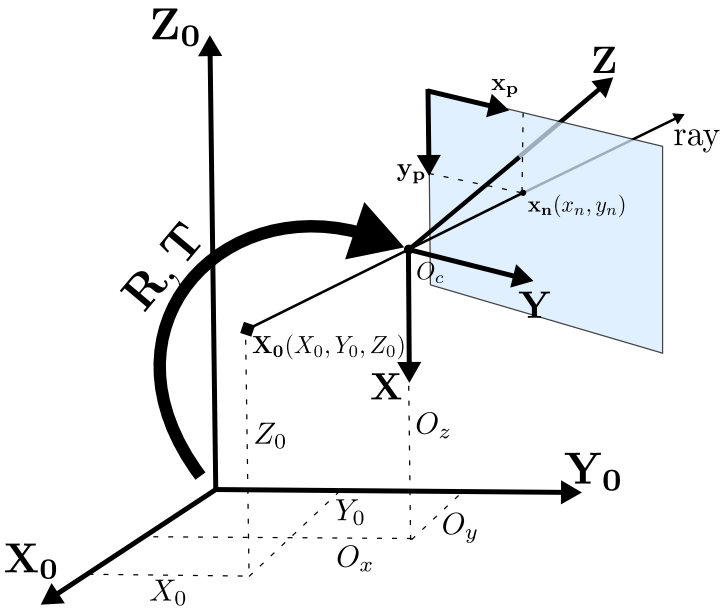


Figure 3.3: Principle of Central Perspective

The principle of central projection describes how under idealized circumstances a 3D point $\mathbf{X_0}$ given in homogeneous world coordinates can be described in camera coordinates and finally projected onto our image plane. The constellation as given in figure 3.3 defines three coordinate systems. First of all the world coordinates can be freely chosen. Usually for single camera calibration the origin will be located in a corner of a chessboard used for the calibration. The x and y axes are then aligned with the chessboard edges while the z axis is orthogonal to the board. The second coordinate system is the one of the camera. The origin of this system is called the perspective center and is marked in figure 3.3 as $\mathbf{O}_c$. The transformation from world to camera coordinates is defined by three rotations given with matrix $\mathbf{R}$ and a translation vector $\mathbf{T}$ thus a point $\mathbf{X_0}$ given in world coordinates can be expressed in camera coordinates with

$$\mathbf{X_c} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{bmatrix}$$

The third coordinate system is the resulting image plane. As shown in figure 3.3 a point $\mathbf{X}_0$ given in world coordinates is projected on $\mathbf{x_n}$ in the image plane and is located on the ray from $\mathbf{X}_0$ through the perspective center $\mathbf{O}_c$. $\mathbf{x_n}$ is described in homogen coordinates by the following equation

$$\mathbf{x_n} = \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \frac{f}{Z_c} \cdot \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \frac{1}{Z_c} \cdot \underbrace{\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{K_f}} \cdot \underbrace{\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right]}_{\mathbf{\Pi_0}} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{3.1}$$

$$= \frac{1}{Z_c} \cdot \mathbf{K_f} \cdot \mathbf{\Pi_0} \cdot \mathbf{X_c} \tag{3.2}$$

with the focal length f. The matrix $\mathbf{K_f}$ is now extended with the intrinsic parameters of our camera to get from the normalized coordinates $\mathbf{x}_n$ to the real pixel coordinates $\mathbf{x}_p$.

$$K_s = \begin{bmatrix} fs_x & s_\theta & o_x \\ 0 & fs_y & o_y \\ 0 & 0 & 1 \end{bmatrix}$$

For most cameras the pixels will never be of square size thus the focal length f needs to be scaled by s_x and s_y. Furthermore the principle point will never be the middle of the captured image. Due to manufacturing errors the point will allways be slightly shifted and we need o_x and o_y to describe this shift in relation to the optical axes. s_θ is called the skew factor which describes the angle between x and y of the sensor. Indeed, for most cameras this factor can be neglected and is set to zero. Summarizing a 3D point $\mathbf{X}_0$ in world coordinates is mapped to a point in pixel coordinates $\mathbf{x}_p$ by following equation:

$$\lambda \cdot \mathbf{x_p} = \mathbf{K_s} \cdot \mathbf{\Pi_0} \cdot \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{bmatrix}$$

3.3 Camera Distortion Model

The idealized model as described above needs to be extended to get one step closer to reality. Due to manufacturing errors there will always appear distortions caused by the lens and a misalignment between lens and camera chip. These distortions happen before the light ray hits the chip and consequently affect the normalized coordinates $\mathbf{x}_n$. Slama [29] introduced a model which is used most of the times. $\mathbf{x}_p$ describes the location of a pixel in the received and distorted image whereas $\mathbf{x}_d$ is the corrected pixel location. According to Slama the corrected pixel location $\mathbf{x}_d$ results in

$$\mathbf{x_d} = \mathbf{x_{rad}} + \mathbf{x_{tan}}$$

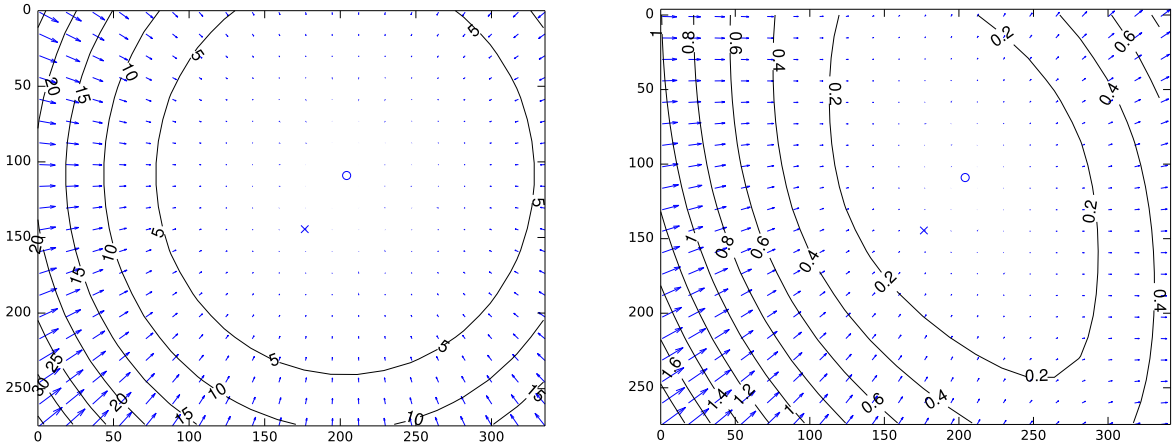
where $\mathbf{x_{rad}}$ describes the radial distortion

$$\mathbf{x_{rad}} = \begin{bmatrix} x_r \\ y_r \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_5 r^6) \cdot \mathbf{x_n}$$

with $r^2 = {x_n}^2 + {y_n}^2$ and $\mathbf{x_{tan}}$ describing the occurring tangential distortion

$$\mathbf{x_{tan}} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} 2k_3 x_n y_n + k_4(r^2 + 2x_n^2) \\ k_3(r^2 + 2y_n^2) + 2k_4 x_n y_n \end{bmatrix}$$

produced by the lens.



(a) Radial Distortion

(b) Tangential Distortion

Figure 3.4: Camera Distortion

Figure 3.4 visualizes the influence of distortion. The arrows point in the direction of the occurring distortions while the length of the arrows resembles the degree of displacement. The o inside the figures marks the principle point given by the cameras while the x shows where the point should be. The main error is produced by the radial distortion. In the paper of Heikkila [14] an algorithm is presented to determine the intrinsic and extrinsic camera parameters including the described distortion coefficients. Once we know the distortion parameters it is possible to undistort the received images. In summary this method allows to undistort each camera on its own.
Our total stereo constellation however is still defective. In section 3.1 we claimed that the cameras are perfectly row aligned what will never be the case therefore a stereo rectification needs to be applied.

3.4 Stereo Rectification

Stereo rectification is the process of remapping images according to the constellation of the cameras to each other in a way that the resulting images are row aligned. In the end we want the points in the left picture of one row to be in the same row of the right picture. This has to be the case to be able to find correlating pixels in real time. Without row alignment the search for a pixel from the left image inside the right image would be a 2D search and consequently consume a lot of time. If we reduce the search space to 1D a lot of calculations can be avoided. To understand where a pixel of the left image can actually appear in the right image the epipolar geometry will now be introduced.

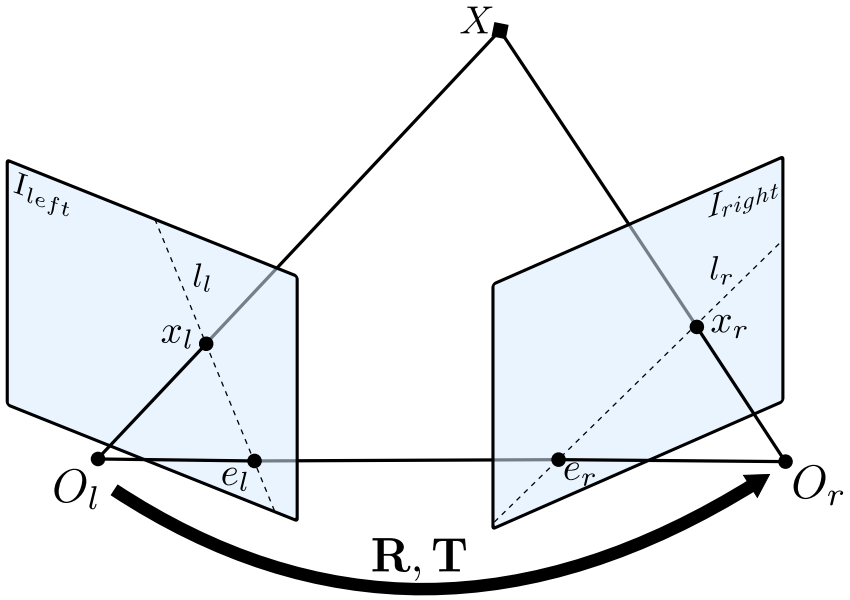


Figure 3.5: Epipolar Geometry

A point **X** is projected on two image planes I_{left} and I_{right} as shown in figure 3.5. All points on the ray from $\mathbf{O_l}$ to **X** are projected on the same point $\mathbf{x_l}$. This ray appears in the image plane of the right hand side as line because the cameras are not aligned perfectly or for example a slight angle between the cameras was chosen to increase the common field of view. The projection of this image ray in the right picture is defined by two points, one is of course the projection of **X**, which is $\mathbf{x_r}$, and the second point is $\mathbf{e_r}$, called epipole. The position of this epipole is only defined by the stereo rig constellation and results as interception point between the connection of the two centers of projection and the image planes. The line from the projected point $\mathbf{x_r}$ through the epipole $\mathbf{e_r}$ is called epipolar line l_r. This line can be calculated out of the stereo rig and the projected point $\mathbf{x_l}$ on the left image. Once we know the epipolar line for the right image we have reduced the search space for the matching point $\mathbf{x_r}$ from 2D to 1D as we know that this point must lie on the epipolar line. But not only a reduction of the search space is achieved by calculating the epipolar line. The probability of wrong matches is decreased as well, because there

are less possibilities for matching points.
The computation of the epipolar line is simple once we have calibrated the stereo rig and determined the so called fundamental matrix $\mathbf{F}$. The calculation of the fundamental matrix can be done e.g. with the eight-point algorithm which will not be explained here. The interested reader however can go into details with [23] and [13]. For the fundamental matrix $\mathbf{F}$ following equation can be formulated

$$\mathbf{x_r}^{\mathbf{T}}\mathbf{F}\mathbf{x_l} = 0$$

With this equation the resulting epipolar line l_r of the format $[a, b, c]$ and the linear equation in of the form $ax + by + c = 0$ is

$$l_r = \mathbf{F}\mathbf{x_l}$$

Now that we know where to search for our corresponding point we still have another problem. The search along a line which is probably not horizontally orientated within our image is not easy to solve. We would first have to calculate for every pixel $\mathbf{x_l}$ the line l_r with the equation above. Then we would have to calculate the possible (x, y) coordinates with the resulting linear equation and at the end use interpolation for calculating the similarity between x_l and the points on l_r.
The idea is to transform the pictures in a way that the epipolar lines are horizontally oriented within the image and thus perfectly row aligned. This simplifies the search for matching pixels. We anyway have to transform the pictures because of the distortion effects described in section 3.3 so it should be possible to merge these transformations to one rectification step at the end.
In order to get horizontal epipolar lines we have to find a transformation for each image plane which maps the epipoles to infinity. The relationship between the two perspective centers $\mathbf{O_l}$ and $\mathbf{O_r}$ is given by a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{T}$ as shown in figure 3.5[1)]. First of all the image planes have to have the same orientation so we rotate both images slightly instead of rotating one image plane by the full angles given in $\mathbf{R}$ in order to minimize the reprojection error. We split up the rotation matrix $\mathbf{R}$ into two rotation matrixes $\mathbf{r_l}$ and $\mathbf{r_r}$. The new translation vector will result in $\mathbf{t} = \mathbf{r_r}\mathbf{T}$. Now the image planes are coplanar meaning that the epipoles are mapped to infinity. However, the images are still not row aligned.
The rotation matrix $\mathbf{R}_{rect}$ which aligns the rows is a rotation matrix about the z axis

$$\mathbf{R_{rect}} = \begin{bmatrix} \mathbf{v_1}^T \\ \mathbf{v_2}^T \\ \mathbf{v_3}^T \end{bmatrix}$$

[1)] Be aware that these transformations are not the same as in the section 3.2 about the principle of central perspective. However, the $\mathbf{R}$ and $\mathbf{T}$ given here can be derived from the extrinsic parameters of each camera $\mathbf{R_l}$,$\mathbf{T_l}$ and $\mathbf{R_r}$,$\mathbf{T_r}$ with $\mathbf{R} = \mathbf{R_r}(\mathbf{R_l})^T$ and $\mathbf{T} = \mathbf{T_r} - \mathbf{R}\mathbf{T_l}$

with

$$\mathbf{v1} = \frac{\mathbf{r_r T}}{\| \mathbf{r_r T} \|} \tag{3.3}$$

$$\mathbf{v2} = \frac{1}{\sqrt{{T_x}^2 + {T_y}^2}} \begin{bmatrix} -T_y \\ T_x \\ 0 \end{bmatrix} \tag{3.4}$$

$$\mathbf{v3} = \mathbf{v1} \times \mathbf{v2} \tag{3.5}$$

The rotations which have to be applied to the views then result in $\mathbf{R_L} = \mathbf{R_{rect} r_l}$ and $\mathbf{R_R} = \mathbf{R_{rect} r_r}$. After the rotations have been applied on both images the new rotation matrix $\mathbf{R}$ results in the identity matrix $\mathbf{I_{3\times 3}}$ and the translatoric vector $\mathbf{T}$ is changed to $\mathbf{TR_R}$.
Now the new camera matrix of the final rectified pictures has to be derived in a way that the rectified images contain as much information of the distorted images as possible. First of all a common vertical focus length has to be found. To get maximum information the minimal focus length of f_{y_left} and f_{y_right} is chosen. The principal points have to be calculated again with the new common focal length. The common principal point is then set to be the mean value between both principal points. With the resulting camera matrix $\mathbf{K_{rect}}$ and the derived necessary transformations it is now possible to determine two look-up tables (one for each camera) which allow us to perform all the described transformations within one step.

3.5 Determination of Look-Up Tables

First of the cameras have to be calibrated to determine the parameters of the described model. The algorithm for determining the camera parameters according to Heikkila [14] and Bouguet's algorithm for stereo rectification are both embedded in the Camera Calibration Toolbox [6] for MATLAB. This toolbox was used for the thesis to calibrate the stereo camera system. The input of the toolbox are pictures of a chessboard taken with both cameras at the same time. A chessboard pattern is presented to the cameras in different angles in order to get various 3D points. For a couple of positions picture pairs are taken of the chessboard placed in the world. After an efficient amount of pictures has been recorded the user has to tell the toolbox the size of the squares and mark the outer squares in each picture in the same order. It is recommended to take more than 20 pictures to have enough input for the following optimization process which calculates the camera parameters. To perform a stereo calibration you first have to calibrate each camera on its own. The resulting files are then used as an input for the stereo calibration. A lot of information and tutorials about the calibration process can be found on the homepage of the Camera Calibration Toolbox [6].

Based on these results it is now possible to derive a look-up table which takes care of both, the distortion and the stereo rectification. To create an undistorted picture we need to

know which coordinate of the rectified picture belongs to which coordinate in the received distorted picture. There are two directions we can go to calculate such a mapping.
If we calculate a mapping starting from integer coordinates inside the received distorted image and calculate the corresponding coordinates inside the rectified picture I_{rect} we will get floating point coordinates for I_{rect} as a result and will end up with a lot of holes inside the rectified image, because we probably do not hit every location.
If we, however, do it the other way round and calculate for every integer coordinate of I_{rect} a source coordinate we get floating point coordinates as well, but now we can simply interpolate the resulting pixel out of the pixel values of the distorted image. In order to calculate this mapping we need to go backwards.
Assume $\mathbf{p_l}$ is a point in our left destination picture (undistorted). The first thing we have to do is undoing the projection. This is achieved by multiplying $\mathbf{p_l}$ with the inverse of $\mathbf{K_{rect}}$ and setting the third coordinate to one. After this step the rotation has to be reversed thus a multiplication from the left with ${\mathbf{R_L}}^T$ has to be applied. In the next step the distortion model is applied and finally the result is projected again with the original $\mathbf{K_{s\_left}}$. The following code shows how the rectification lookup table is gained in Matlab.

```
%% Calculate mapping left
% create pixel coordinates
[mx,my] = meshgrid(0:nx/floor(nx):(nx-1),0:ny/ny:(ny-1));
[nnx,nny]=size(mx);
px=reshape(mx',nnx*nny,1);
py=reshape(my',nnx*nny,1);
%undo projection
rays = inv(KK_left_new)*[(px - 1)';(py - 1)';ones(1,length(px))];
% Rotation: (or affine transformation):
rays2 = R_L'*rays;
x = [rays2(1,:)./rays2(3,:);rays2(2,:)./rays2(3,:)];
%distort
xd=apply_distortion(x,kc_left);
%project
px2=fc_left(1)*(xd(1,:) + alpha_c_left*xd(2,:)) + cc_left(1);
py2=fc_left(2)*xd(2,:) + cc_left(2);

mapping_left = [px py px2' py2'];
```

3.6 Calculation of the Rectified Image

As already mentioned the coordinates of the lookup table are most likely no integer values, so interpolation has to be used in order to calculate the pixel values. The most common way is to use bilinear interpolation in order to keep the complexity of the resulting algorithm low. For the bilinear interpolation we need four pixels surrounding our floating point position as shown in figure 3.6. The resulting pixel intensity then can be approximated with

$$I_{rect}(x_{dest}, y_{dest}) = \begin{bmatrix} 1-dx & dx \end{bmatrix} \begin{bmatrix} I(x_{src}, y_{src}) & I(x_{src}, y_{src}+1) \\ I(x_{src}+1, y_{src}) & I(x_{src}+1, y_{src}+1) \end{bmatrix} \begin{bmatrix} 1-dy \\ dy \end{bmatrix}$$

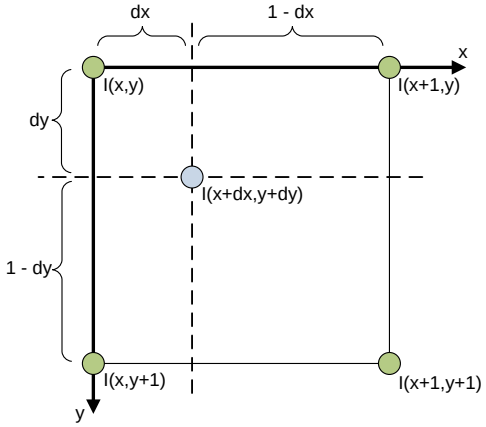


Figure 3.6: Bilinear Interpolation

A software solution based on lookup tables which have been exported from MATLAB achieved 18.9 fps on the EyeBot M6. This rate however does not contain capturing the stereo frames. It is the mere transformation of images stored inside the RAM without any communication between the CPU and the FPGA. As the rectification has to be the first step of image processing it has to be done in hardware. The hardware implementation of the rectification is described in the following.

4 Rectification in Hardware

A common way of implementing rectification in hardware is the usage of the LUTs as in the software approach. The calibration is once done offline and the resulting two LUTs - one for each camera - are loaded into the FPGA.
In [31] the unrectified image comes in a stream from the camera and is at first stored to a DRAM. The LUTs are placed inside a second DRAM to avoid the creation of a bottleneck. A counter steps trough all coordinates of the destination image and loads the corresponding source coordinates from the LUT. These coordinates have an integer and a fractional part in order to perform a bilinear interpolation. For every source coordinate four pixels of the source image are loaded from the DRAM. After interpolating these four pixels the resulting pixel of the rectified image is stored into the DRAM again.
Unfortunately it is not possible to implement this design on the EyeBot M6 in an efficient way. Above all we will run out of memory to save the LUTs. The size of the four LUTs, for each coordinate x and y of each side one LUT, is given by $4 * width * height * wl$ bit resulting in 693 kbyte if we use `ufix[14,5]` for the representation of the coordinates with fraction part. The Spartan 3E comes with 20 block RAMs each with a size of 18 kbit. Consequently, with all available block RAM it is only possible to store 50% of our LUTs.
Another option is to use the SRAM for both, saving two raw pictures, two rectified pictures and the two LUTs. The necessary amount of memory then would sum up to 1485 kbyte. The SRAM used in our design comes with 2197 kbyte of memory, however, the memory has to be segmented. The word width of the SRAM is 18 bit but one color pixel has only 16 bit. By ignoring 2 bits or for example using them as parity bits we already loose 244 kbytes of our SRAM. Still there would be 467 kbyte left for additional data as for example the feature points of the Harris Detector (see chapter 5). Beside the shortage of memory the SRAM will be the design's bottleneck regarding transfer speed. We already have five devices requesting data from or writting to the SRAM. The cameras have to write the raw images to the SRAM. When the pictures have been written four pixels for every undistorted pixel need to be loaded in order to perform the interpolation. The rectified images need to be written again into the SRAM to make them available for the CPU. The feature coordinates resulting from the Harris Corner Detector have to be saved as well. Finally the rectified pictures together with the feature list need to be read from the CPU. Because of this huge amount of traffic and the shortage of memory it should be avoided to additionally store the four LUTs inside the SRAM and read them continuously.
Another option presented in [17] is to reduce the traffic by not saving the images to the SRAM and instead processing them directly in stream. For this at least n lines have to be buffered with the internal block RAM where n equals the biggest y-offset between a

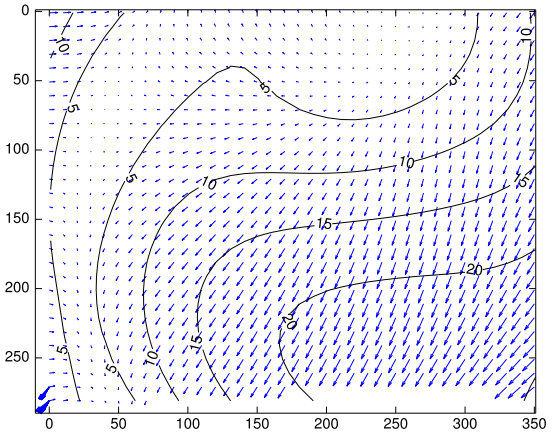
coordinate at the left and the right side of the image. For the current setup of the cameras n equals 20 lines what would exactly fit into the available block RAM but only for one camera. Moreover a lot of block RAM will be necessary for the later processing of the rectified images so this is again not an option for the EyeBot M6.
To avoid the storage of the LUTs a further alternative has been considered. Without a LUT the coordinates have to be calculated what is time consuming. However, the time needed for the calculation of the next coordinates can be used to load meanwhile the four pixels from the SRAM and interpolate the rectified pixel. If the calculation is fast enough no time is lost at all and there is no need for additional SRAM-accesses for loading the coordinates as in [31]. One drawback with this approach is that the calculations done inside the Rectification Toolbox (see appendix A) to gain the mappings are far too complex to realize them inside the FPGA. As described in section 3.3 polynomials of the sixth degree are evaluated inside the distortion model. In addition a normalization of the coordinates is needed which comes hand in hand with implementing a fixed-point division inside the FPGA. To avoid the complexity of these calculations I analyzed the possibilities to use a 2D regression polynomial to approximate the produced mappings.
Based on the MATLAB regression function, 2D polynomials of second, third and fourth degree have been evaluated. For both coordinates x and y and for each side a polynomial is regressed (here a polynomial of third degree):

$$\begin{aligned} x_{src} &= p_1 + p_2x + p_3y + p_4x^2 + p_5y^2 + p_6xy + p_7x^3 + p_8y^3 + p_9x^2y + p_{10}xy^2 \\ y_{src} &= q_1 + q_2x + q_3y + q_4x^2 + q_5y^2 + q_6xy + q_7x^3 + q_8y^3 + q_9x^2y + q_{10}xy^2 \end{aligned} \tag{4.1}$$

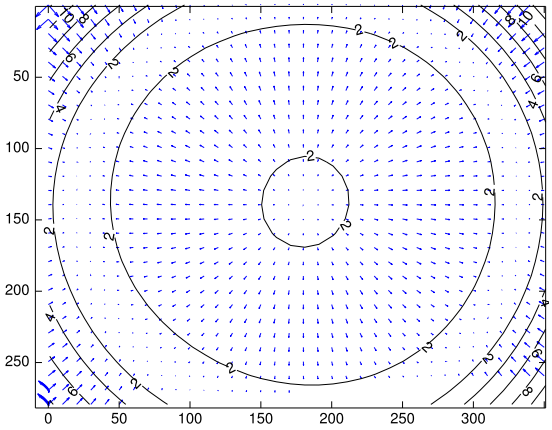
In order to judge about the quality of the regressed polynomials we need to measure the error produced by the approximation. As an error measure the mean distance in x, y and the spatial distance r between the real mapping given by the developed Rectification Toolbox and the mapping resulting from the regressed polynomials are used. Additionally the difference between the mappings has been visualized similar to figure 3.4 to locate the errors. The arrows in such a figure resemble the deviation in 2D. The length of the vector is proportional to the distance. Furthermore the figures are separated with the help of contours to be able to localize the error easier. For basic visual inspection the mappings were finally applied to the images.
As it can be concluded from table 4.1 and figure 4.1(b) a polynomial of second degree is definitely not good enough to regress the rectification formula. The radial error is still not compressed and can be observed by simply looking at resulting "rectified" pictures. In addition it can be concluded from the results that polynomials of third and fourth degree produce good results. Only at the corners we find a projection error bigger than one pixel. Not the whole rectified image can be used for that reason, but the lost area is still within acceptable range. The increase of complexity by implementing a polynomial of fourth degree instead of third degree in hardware is not justified by the small difference of the produced error.

type of polynomial	residuals			mean			var		
	x	y	r	x	y	r	x	y	r
none	12.90	20.32	25.83	5.97	9.03	11.22	22.78	55.14	47.85
O(p(x,y)) = 2	8.14	9.61	14.10	1.90	1.72	2.77	5.54	4.94	2.79
O(p(x,y)) = 3	1.40	3.51	5.22	0.25	0.22	0.36	0.11	0.09	0.08
O(p(x,y)) = 4	1.81	1.76	2.48	0.18	0.16	0.25	0.05	0.04	0.03

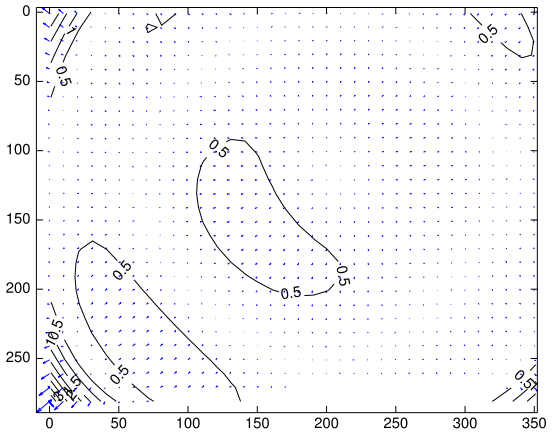
Table 4.1: Evaluation of regression polynomials



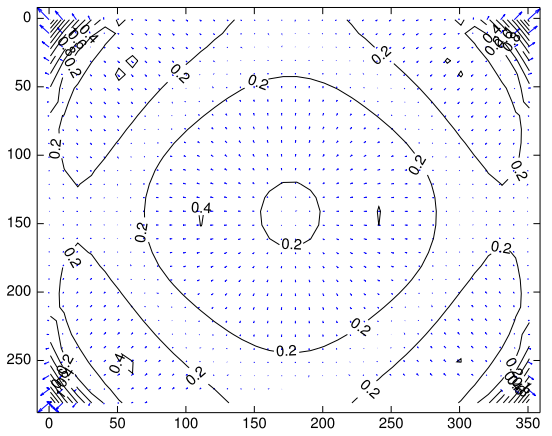
(a) No Correction



(b) Correction with polynomial of second degree



(c) Correction with polynomial of third degree



(d) Correction with polynomial of fourth degree

Figure 4.1: Remaining Error after Rectification with regressed Polynomials

Concluding from these results polynomials of third degree will be used to calculate the source coordinates within the FPGA. A function for the regression of the polynomials became therefore part of the Rectification Toolbox. Now the discrete hardware implementation of the rectification system will be described. First of all an overview is presented in order to go more and more into the implementation details afterwards.

4.1 The Rectification System - Overview

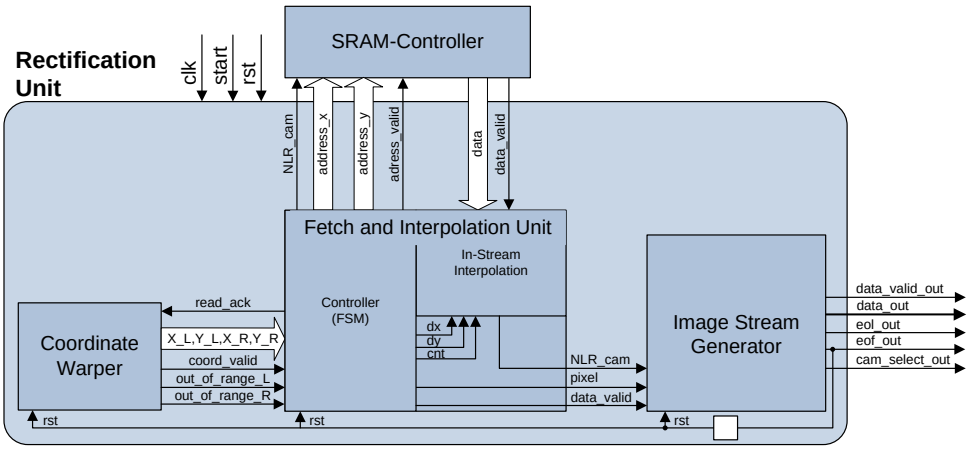


Figure 4.2: Overview of Rectification System

Figure 4.2 gives an overview of the developed rectification system. It can be divided into three subsystems: the coordinate warper, the fetch and interpolation unit and the image stream generator. A start signal notifies the rectification unit that two whole images were saved inside the SRAM and a new rectification cycle can be started again. The coordinate warper is the functional heart of the system.

4.2 Coordinate Warper

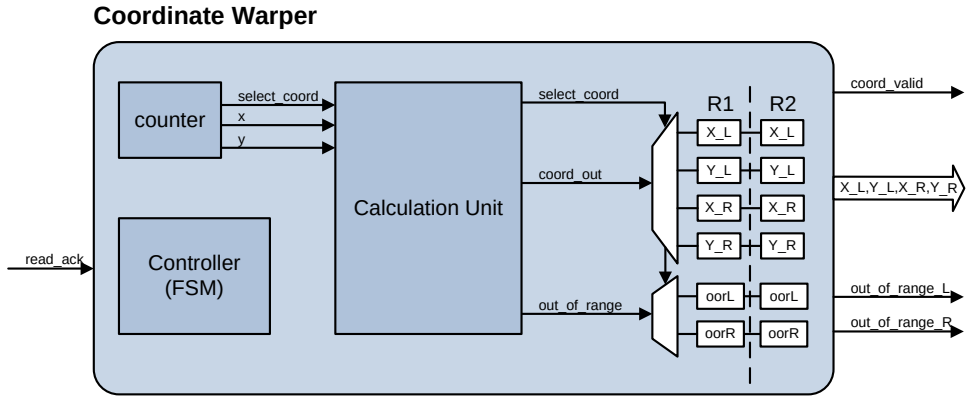


Figure 4.3: Coordinate Warper Overview

The coordinate warper consists of a calculation unit which computes four polynomials and a control unit (see figure 4.3). In order to understand how the data flow has to be controlled by the control unit we first have a look into the calculation unit and how the coordinates are generated there.

4.2.1 Calculation Unit

Inside the calculation unit four polynomials of third degree have to be evaluated. If such a polynomial calculation is implemented straight forward 20 multipliers and 9 adders are necessary for each polynomial. By rewriting the polynomial equation the number of multipliers can be reduced to 9 each.

$$x_{src} = p_1 + y(p_3 + p_6 x + y(p_5 + p_8 y + p_{10} x) + x(p_2 + x(p_4 + p_7 x + p_9 y))$$

Figure 4.4 shows this equation implemented in Simulink.

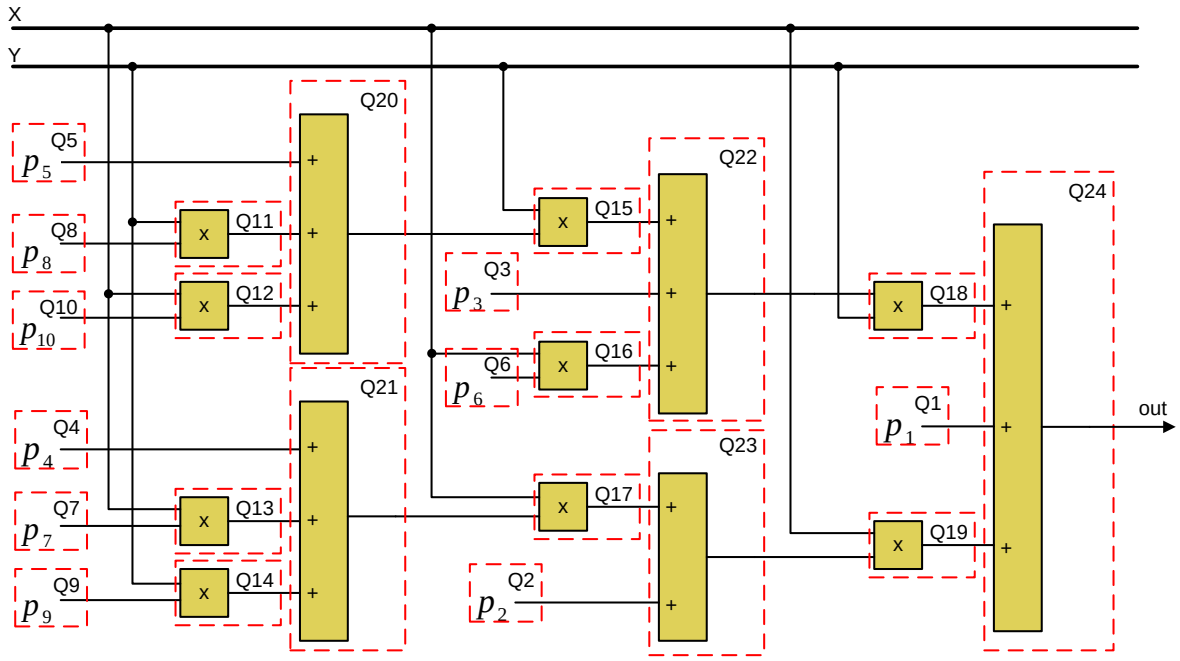


Figure 4.4: Implementation of Polynomial of third Degree

As described in chapter 2 a floating point representation cannot be used inside the FPGA and the data needs to be converted to the fixed point representation by quantization. The positions where quantization has to happen inside the FPGA are marked with red squares and numbered in figure 4.4. Altogether there are 24 positions plus 2 quantizers for the incoming coordinates x and y what is, combined with the highly non-linear behavior of the quantization error, definitely too much to be solved by hand. This is the reason why the FFC of the coordinate warper has been automated as described in section 2.5. For the polynomials left-X, left-Y, right-X and right-Y the automated floating to fixed point conversion produced the quantizer settings listed in table A.2.

Once the FFC is done the calculations can be coded. However, the calculation is too complex for being done within one clock cycle inside the FPGA. The logical paths would

be much too long and thus the maximum frequency would drop drastically. So first of all we have to figure out how many register stages we are allowed to place inside the calculation unit in order to reduce the logical paths but meanwhile keep the maximum throughput of the rectification. This maximum possible throughput is limited by the speed of the SRAM. For two coordinate pairs $(x_l, y_l), (x_r, y_r)$ we have to fetch eight pixels from the SRAM, because we need four source pixels to interpolate one destination pixel for each side. For each pixel we will need one clock cycle at best if no other device is communicating with the SRAM. Thus we will have at least eight clock cycles of time until two valid coordinate pairs have to be available at the output if we do not want to slow down the whole system with the calculations. If we place a register after each arithmetic operation inside the four polynomial solvers we would get six stages in each, which does not violate our maximum time of eight clock cycles. The implementation of one calculation unit with six registers in between returned a maximum clock rate above the 100 MHz which is as well the theoretical maximum SRAM rate. So this seems to be an appropriate solution.
However, things change if we implement the necessary four polynomial calculators. Due to the fact that we only have 20 hardware multipliers available inside the Spartan 3E the synthesize tool implements the multipliers with our available resources. Thus the consumption of resources increases rapidly as shown in table 4.2. Furthermore the maximum speed drops to 67.7 MHz because the generated multipliers are much slower than the hardware multipliers. Therefore the design has to be reconsidered.

Resource	used	out of	percent
Number of Slices	2083	4656	44
Number of Slice FFs	1218	9312	13
Number of Multipliers	20	20	100
Max Frequency	67.751MHz		

Table 4.2: Resource Consumption of four Calculation Units

At that point we get four coordinates after every six clock cycles. During the six clock cycles no valuable data lies at the output of our four calculation units. The idea is now to merge the units into one by fully pipelining the signals. With a pipelined design we can get a valid coordinate every clock cycle once the pipeline is filled, so after four clock cycles a full set of coordinates would be available. Although this sounds promising another problem occurs. The calculations are optimized with the FFC to use the smallest possible bit width for the calculations in order to minimize resource consumptions. This optimization is done for each polynomial itself. As you can tell from table A.2 the resulting bit representations are definitely not the same. The idea is now to merge all four quantizers by determining the minimum common bit width and largest necessary fraction length for each quantizer. For this process the Rectification Toolbox has been extended with the function `merge_quantizers()` which gets any number of quantizer lists and returns a single list with common minimum quantizer settings. The output of this function, the merged quantizer settings of our four calculation units, is as well

listed in table A.2. The increase of the bit width is still in an acceptable range because the inputs of the multipliers are still below 18 bit what is the maximum input width of the hardware multipliers of the Spartan 3E. Otherwise Xilinx ISE would connect two multipliers in a row resulting in a decreased maximum clock rate again. Based on this thoughts a fully pipelined and merged calculation unit was developed.

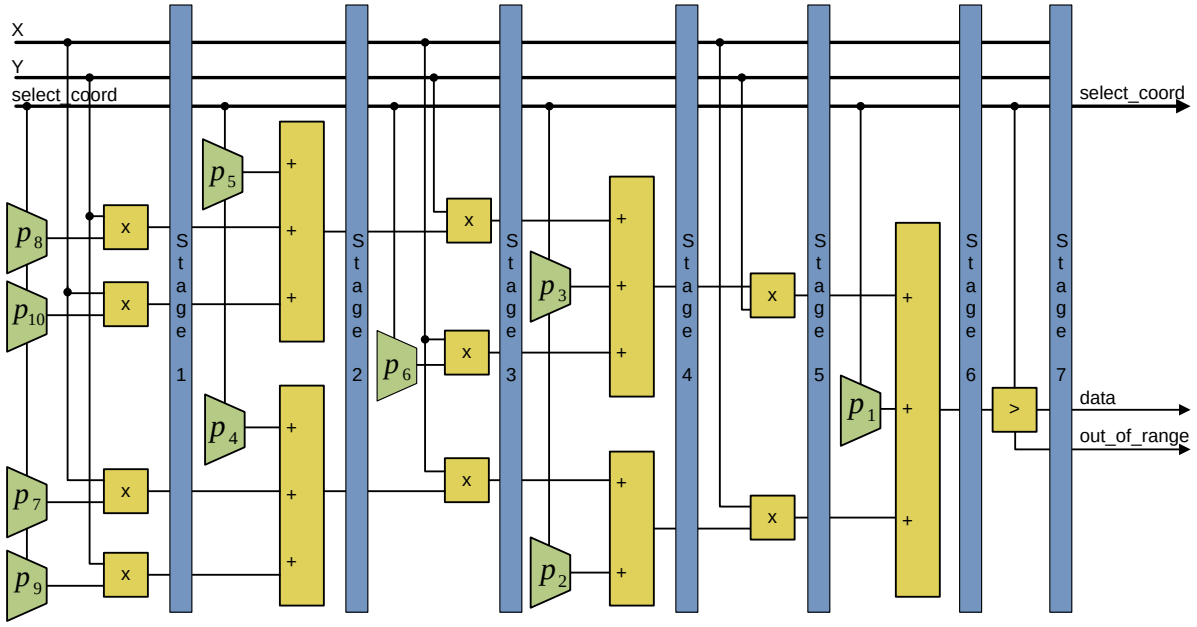


Figure 4.5: Fully pipelined Coordinate Calculation

Figure 4.5 shows the resulting merged calculation unit and the introduced registers. There are several signals which have to be pipelined and additionally created if we want to merge the four calculation units to form one. The signals which need to be pipelined are the inputs x, y and select_coord and the results of the calculation steps. Additionally we need multiplexers in every stage where coefficients are involved into an operation because they obviously differ from polynomial to polynomial.
To control these multiplexers the signal select_coord has been introduced telling every stage which of the four polynomials it is actually calculating at the moment. This signal is output with the polynomial result after it has gone through the pipeline and is encoded as shown in table 4.3. A global enable can furthermore freeze the whole pipeline if set to low and so stop the coordinate output. This is necessary if the SRAM fetches are delayed and the coordinates cannot be read fast enough.
At the output of the calculation unit and before the coordinate is reduced to the necessary accuracy for the interpolation a comparison is necessary to detect whether the coordinate lies within the image or not. This is important to be done at this point because the values could be falsified due to quantization. There are two comparisons which have to be done. The first one checks if the coordinate is positive. The second comparator checks if the coordinate is still within the image range. A multiplexer switches the in-

put for the comparator between IMG_WIDTH-2 and IMG_HEIGHT-2 controlled by the select_coord signal. Two has to be subtracted because first of all the coordinates have the range [0;IMG_WIDTH-1] or [0;IMG_HEIGHT-1] and furthermore we need to be able to load the pixel of the current location and the position + 1 in order to perform the interpolation. If the coordinate lies within the image the output signal out_of_range is set to zero respectively to one. The validity of the coordinate is certified with the third bit of the select_coord signal.

enable	$\overline{\text{LR}}$	$\overline{\text{XY}}$	coordinate
0	0	0	-
1	0	0	left X
1	0	1	left Y
1	1	0	right X
1	1	1	right Y

Table 4.3: Encoding of the select_coord signal

4.2.2 Automatic Package Generation

Due to the fact that we automatically determine the quantizer settings it would be helpful if we did not have to reprogram the calculation unit every time the camera arrangement is changed. For the reprogramming the user would have to go deep into the calculation code in order to make the changes and adopt the polynomials for the new configuration. To avoid this a MATLAB function `create_stereo_cam_package()` has been written which automatically creates a VHDL package out of the determined quantizer settings. Inside this package subtypes and constants are defined for every quantizer. Additionally we need the information about fraction and word length saved as a constant.

```
...
-- q_18=sfix[17,7]
constant q_18_wl : integer := 17;
constant q_18_fl : integer := 7;
subtype type_q_18 is signed(16 downto 0); -- sfix[17,7]
...
```

Inside the calculation unit the resulting signals of the calculation steps are based on these subtypes. In order to calculate with these signals the programmed fixed point library is used. The constants q_x_wl and q_x_fl are then used as parameter values.

```
...
signal q_18 : type_q_18;
...
q_18 <= mul_fp( q_22, q_22_fl, q_26_delay4, q_26_fl, q_18_wl, q_18_fl);
...
```

Because functions are overloaded and exist for every combination of signed and unsigned the user does not have to care about the function calls anymore. The synthesize tool selects the matching function automatically and resolves the types without creating any overhead. Additionally the package includes the coefficient settings for every combination X left, Y left, X right and Y right defined as constants.

```
-- Coefficients LEFT X
constant q_1_L_X_val : type_q_1 := "11101110011"; -- sfix[11,7]
constant q_2_L_X_val : type_q_2 := "0101110101111010"; -- sfix[16,15]
...
```

With this package and the fixed point package including all necessary functions it is possible to change the whole polynomial calculations without any deeper understanding of the code. The only thing which has to be done is to call the function `create_stereo_cam_package()` in MATLAB with the quantizer settings and include the generated VHDL package in the calculation unit with one single line of code.

```
use work.stereo_calibration_pkg.all;
```

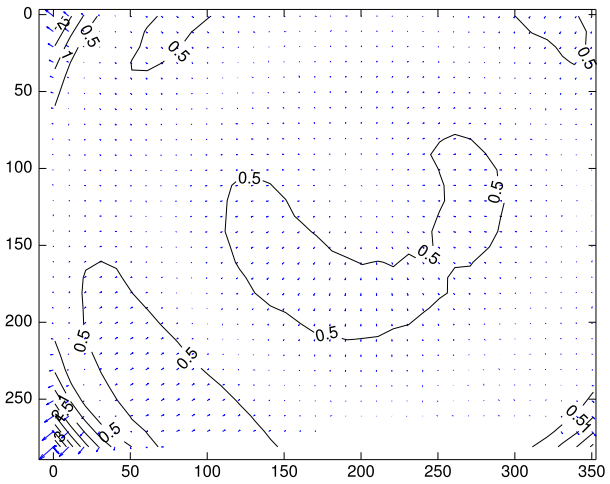


Figure 4.6: Correction with quantized polynomial of third degree

The still remaining error as already plotted for floating point accuracy in figure 4.1(c) is now plotted in figure 4.6 including the additional quantization error. As you can conclude from the small difference the automated FFC of the Rectification Toolbox did a good job. For this pipelined and highly adaptable calculation unit a control unit has been programmed which will be described next.

4.2.3 Coordinate Warper Control Unit

In order to put the calculation unit into operation we first need to generate the input signals. There are two counters running for the x and y values representing the position

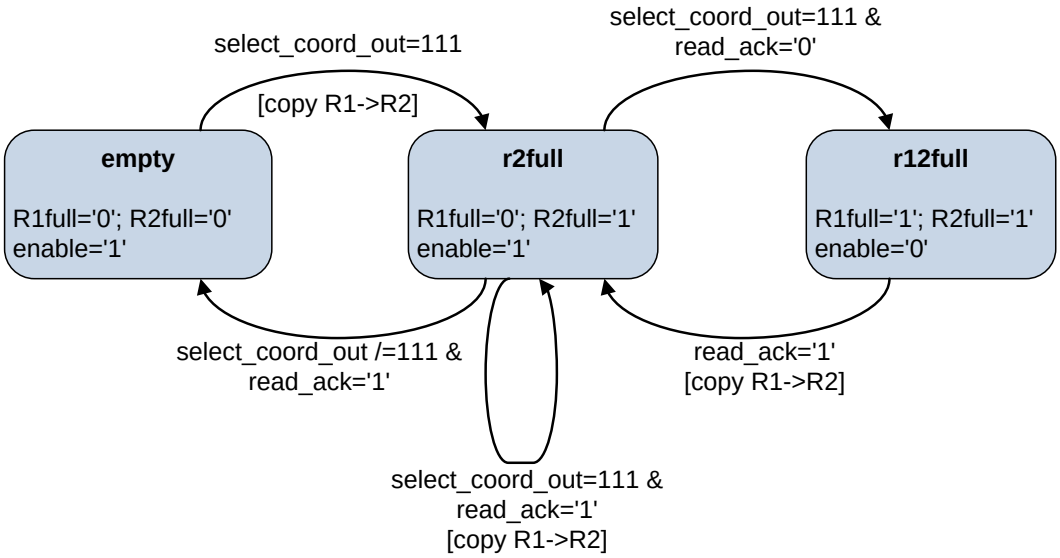


Figure 4.7: FSM inside the Coordinate Warper

in our rectified image. The x value is two bits wider than the necessary nine bit (for an image width of 352 pixel). The lower two bits are used for the select_coord signal described above. The x-counter is increased with every clock cycle but only if the pipeline is enabled. y is incremented if the upper nine bits of x representing the pixel location are equal to the defined IMG_WIDTH.
At the output of the calculation unit the calculated coordinates are multiplexed into the according registers (see figure 4.3). This multiplexing is controlled by the select_coord_out signal which also notifies if the coordinate is valid. The out_of_range signal has to be multiplexed as well but is divided only into left or right frame. Once all four registers X-L, Y-L, X-R, Y-R are filled with valid data the attached SRAM fetch and interpolation unit (FAI) is notified with the signal coord_valid. As soon as the coordinates are read the FAI pulls the read_ack signal high for one clock cycle in order to notify the coordinate warper that the output registers are available again for writing. The FAI starts now to process these two coordinates and interpolates two pixels, one for the left and the other one for the right coordinate. Meanwhile the Coordinate Warper already calculates new coordinates and hopefully finishes before the FAI can request new ones.
Our pipeline is theoretically capable of producing one valid coordinate part every clock cycle. Thus a valid set of coordinate pairs (P_l, P_r) can be calculated every four clock cycles. Yet, in the timing diagram in figure 4.8 it can be recognized that with a simple one-register output buffer the enable signal for the calculation unit has to be pulled to low as soon as a valid set of (P_l, P_r) is inside the output register. We definitely cannot rely on the FAI to be able to request coordinates as soon as they are available because the FAI unit highly depends on SRAM request times.
We have to ensure that the data in the registers is not overwritten so we have to disable the pipeline. This disabling of the calculation unit results in a pipeline stall of two clock cycles if the coordinates are read with maximum available speed. To be able to make use

of the pipeline's maximum speed double buffering has been introduced. Valid data is read from the second level of registers R2 while the pipeline is allowed to write into the first level R1. With this double buffering technique the pipeline enable signal can stay high and it is possible for the FAI to read coordinate pairs every four clock cycles as you can see in figure 4.8.

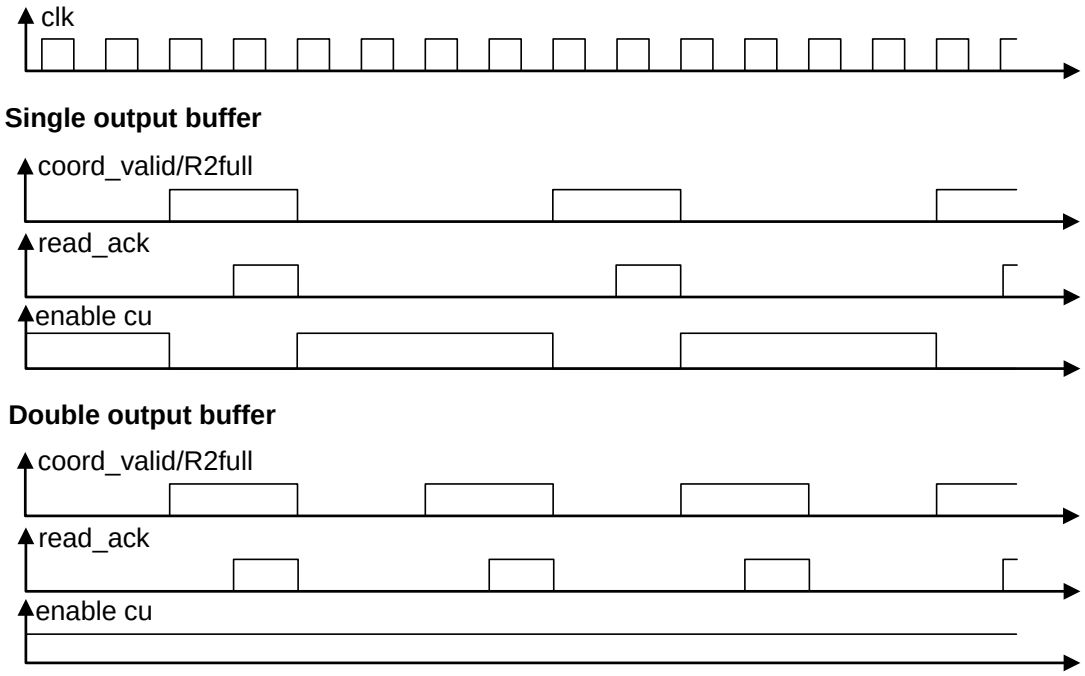


Figure 4.8: Comparison of Single and Double Buffering the Calculation Output

To control the output flow and the double buffering the state machine shown in figure 4.7 has been developed. There are three different states representing the filling status of the double buffer registers R1 and R2. The status is saved in the signals r1full and r2full. If r2full is high valid data is available, consequently this signal represents as well the coord_valid signal. If r1full and r2full are both high the double buffer is completely filled and no further data must be output by the calculation unit. Therefore the negated r1full signal resembles the pipeline enable signal.
The filling status of the coordinate registers is captured with the help of the select_coord_out signal. If this signal equals binary 111 we know that the coordinate Y-R, which is the last of the four, lies at the output and the register level one is completely filled. Thus the data can be copied to R2 and coord_valid switches to high. This, however, can only happen if we are in the state empty or R2full and receive a read_ack at the same time. In both cases we will end up in the state R2full. If R2 is already full and read_ack is low the data has to stay in R1 and the pipeline is disabled until we receive a read_ack from the SRAM Fetch and Interpolation Unit. With an incoming read_ack the data of R1 is copied to R2 and the pipeline can be enabled again.

4.3 SRAM Fetch and Interpolation Unit

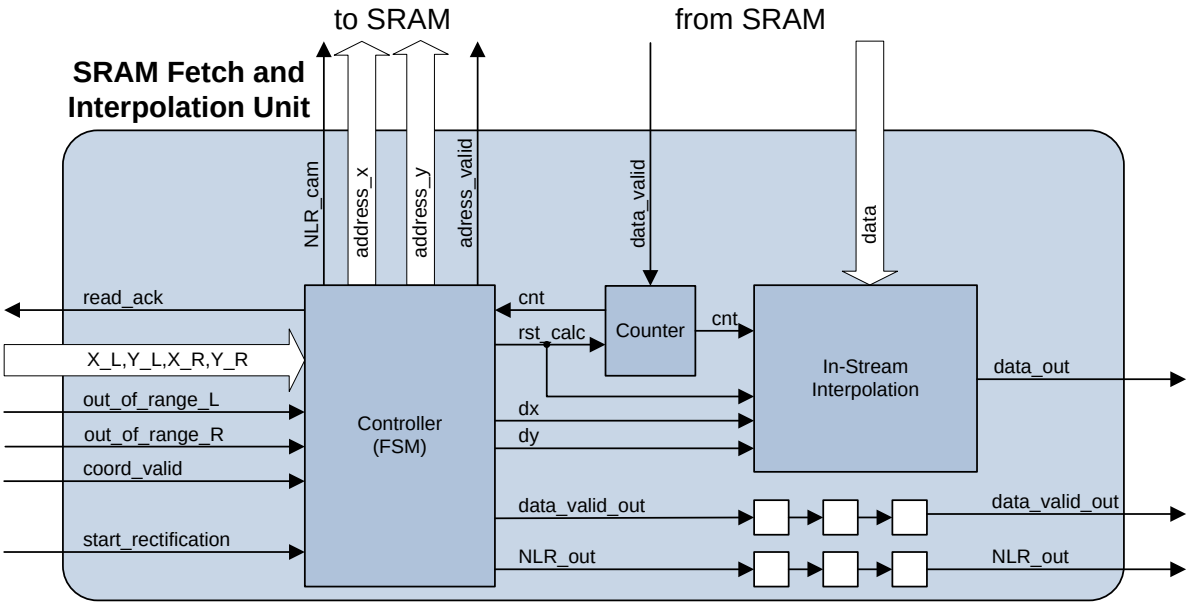


Figure 4.9: Fetch and Interpolation Unit Overview

The SRAM Fetch and Interpolation Unit (FAI) is the heart of the Rectification System. It reads the coordinates from the Coordinate Warper, requests if necessary four pixels for each coordinate P_l and P_r from the SRAM and interpolates the received pixels. Figure 4.10 shows the state machine controlling the whole system.

4.3.1 Controller

When the system starts up it waits until it receives a start_rectification signal. This signal indicates e.g. the fact that two images which can be rectified are inside the SRAM and therefore causes the FAI to start working. First of all we need to wait for valid coordinates from the Coordinate Warper which are signaled with the coord_valid signal. Once the signal is high the values are stored in registers, a read_ack is sent and the fetching of the necessary pixels from the SRAM is initiated.

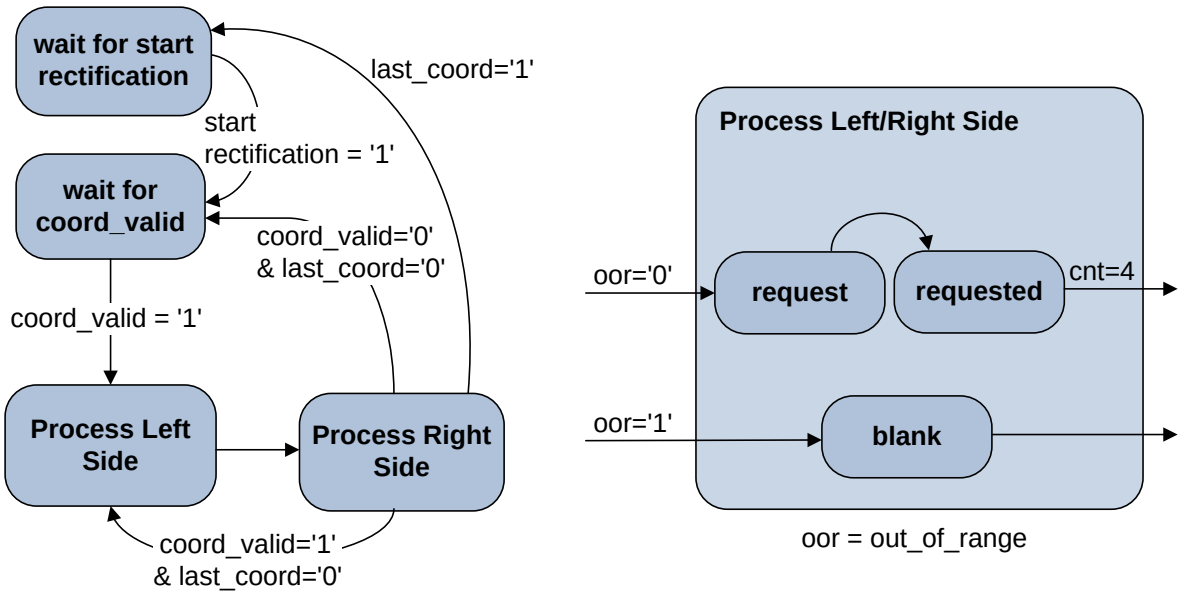


Figure 4.10: FSM of Fetch and Interpolation Unit

In order to calculate one new pixel four pixels have to be loaded from the SRAM and interpolated but only if the four pixels are all inside the image so the out_of_range signal is low. In the request state (left or right) of the FSM as shown in figure 4.10 the coordinate (x, y) and a the signal $\overline{L}R\_cam$ representing the selection left or right are transfered to the SRAM controller with a high address_valid signal. Additionally the counter which is responsible for counting the incoming pixels from the SRAM is set to zero. The SRAM sends four pixels with the coordinates (x, y), $(x + 1, y)$, $(x, y + 1)$ and $(x + 1, y + 1)$ to the FAI as soon as possible. Valid incoming data is signaled by a high SRAM data_valid signal which is as well used to increase our counter. Once the counter value has reached four we know that all necessary pixels for the current coordinate are received from the SRAM and we can request the next pixels.
If the out_of_range signal is high no pixels need to be requested and a black pixel is output. When pixels of the left and right side are calculated for the current coordinate the next state depends on the coord_valid signal. If it is high already the state machine reads immediately the coordinates, sets read_ack to high and jumps directly into the process left side state. If coord_valid is still low the rectification system has to wait for the next valid coordinate.
The incoming pixels have to be interpolated as already mentioned before the data_valid_out signal of the FAI unit can be set. This is done directly inside the pixel stream coming from the SRAM after a coordinate has been requested. Therefore the SRAM Controller will be programmed in a way that it sends the four pixels directly in a row [10].

4.3.2 In-Stream Interpolation

For bilinear interpolation as explained in chapter 3.6 we need to calculate the following equation.

$$\begin{aligned} I_{rect}(x_{dest}, y_{dest}) = & \, dxdyI\left(x_{src}+1, y_{src}+1\right) \\ & + dx(1-dy)I\left(x_{src}, y_{src}+1\right) \\ & + dx(1-dy)I\left(x_{src}+1, y_{src}\right) \\ & + (1-dx)(1-dy)I\left(x_{src}, y_{src}\right) \end{aligned}$$

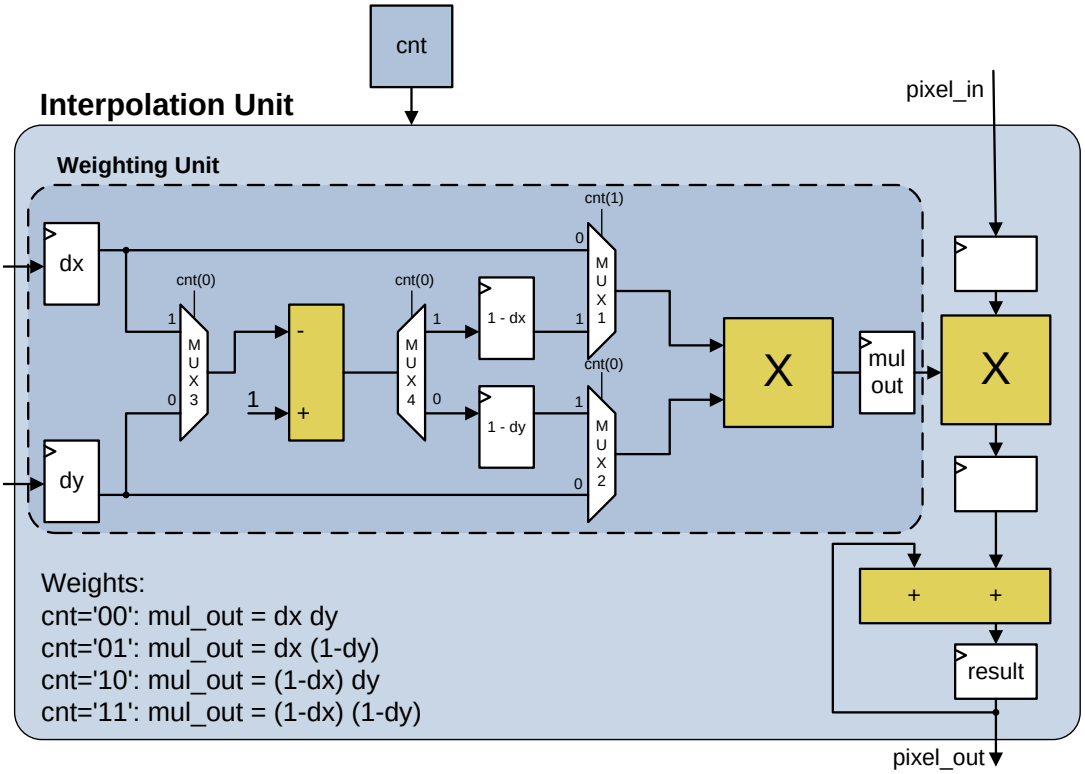


Figure 4.11: Interpolation Unit

In order to interpolate now we first need to calculate the weighting factors out of the fraction part (dx,dy) of the current coordinate. The weighting factors we need are $dxdy$, $dx(1-dy)$, $dy(1-dy)$ and $(1-dx)(1-dy)$. The factors are calculated in hardware one after the other to use as few resources as possible (see figure 4.11). There are four multiplexers which are controlled by the actual pixel count. Multiplexer 3 and 4 take care that the $(1-dx)$ and $(1-dy)$ terms are calculated successively and put into the corresponding registers. Multiplexer 1 and 2 select from the registers to generate the right input for the multiplier. The order of incoming pixels is chosen in a way that the pixel $P\left(x_{src}+1, y_{src}+1\right)$ is send at first by the SRAM because this one needs to be multiplied with $dxdy$ which can be calculated within one clock cycle (the first one). The order of

the following pixels has to be the same as shown in figure 4.11.
The output of the weighting unit has to be registered because otherwise the calculation paths would get too long and the maximum frequency would drop again. To be able to multiply the registered and thus delayed weighting coefficients with the corresponding pixel the incoming pixels are delayed as well. These registers only produce latency and no loss in time.
The registered pixels and coefficients are then put into a multiply and accumulate unit where the pixel values are weighted by multiplying with the corresponding weighting factor. Afterwards the weighted pixels are accumulated. Once all four pixels are summed the value stored in the output register contains our valid interpolated pixel. After the fourth pixel has been received the interpolation unit needs two additional clock cycles because of the delays in between. Therefore the signals NLR_data_out and data_valid_out controlled by the FSM need to be delayed as well. The multiply and accumulate unit is reset by the described FSM after the data has been sent to the Image Stream Converter. The Image Stream Converter is a simple module that creates with the help of counters a so called Image Stream. The usage and details about the Image Stream will be described in section 6.1.

4.4 Summary

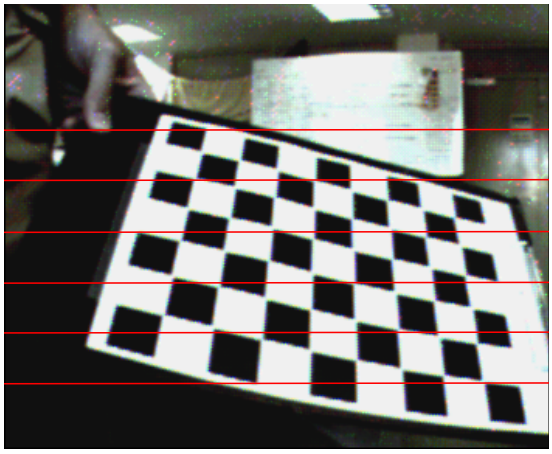
The fact that a rectification inside a Spartan 3E is highly limited by the available memory, multipliers and LUTs of the FPGA made it necessary to share available resources whereever possible if any further image processing has to be done after the rectification inside the FPGA. The described rectifier is the result of economical programming and finally allocates just a fraction of the primarily estimated resources. The total consumption can be seen in table 4.4. The resource consumption regarding used LUTs is the result of the Post PAR (Place & Route) which gives the resource consumption sorted by modules.

Unit	MUL	in %	LUTs	in %
Calculation Unit	9	45	262	2.81
Coordinate Warper	0	0	99	1.06
FAI	2	10	23	0.25
Total	11	55	384	4.12

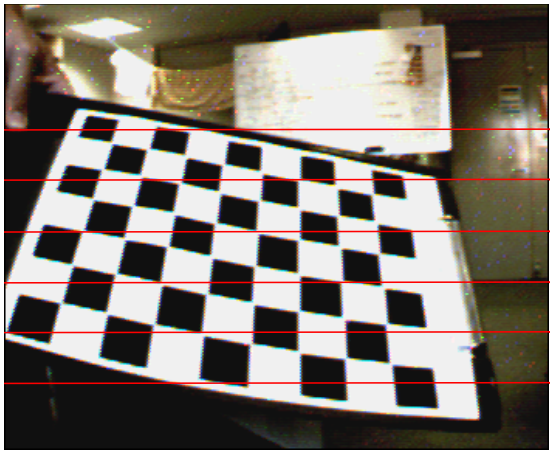
Table 4.4: Resource Consumption of the Rectification Unit

In comparison the design of [17] would allocate already 17.9 % of LUTs and 100% of block RAM for a single camera in stream rectification on our Spartan 3E. A similar amount of occupied LUTs would result out of the implementation presented in [31]. A performance measurement of the given system unfortunately can not be presented at this point because the SRAM interface was not ready to use when this thesis was handed over. Therefore a Stereo Bitmap Package has been written which allows to load bitmaps into Modelsim and use these as input of the rectification algorithm. A simple model of the SRAM Controller

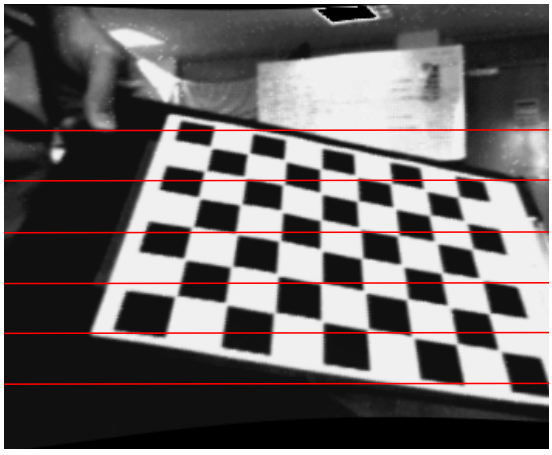
has been designed which simulates random delays resembling the time until the rectification unit wins the arbitration. As already mentioned, five devices have to access the SRAM thus an arbitration unit has been planned in the thesis of [10] which guarantees the candidate with highest priority access to the SRAM. A detailed description of a frame rate estimation is given in chapter 7.
Figure 4.12 presents rectified images in comparison to the unrectified versions. The distortion can be seen especially at the disformation of the chessboard close to the corner which is compensated in the rectified picture. To visualize the misalignment and its compensation red horizontal lines are inserted into the images. It is easy to recognize in the rectified images that they are row aligned and ready for further image processing.



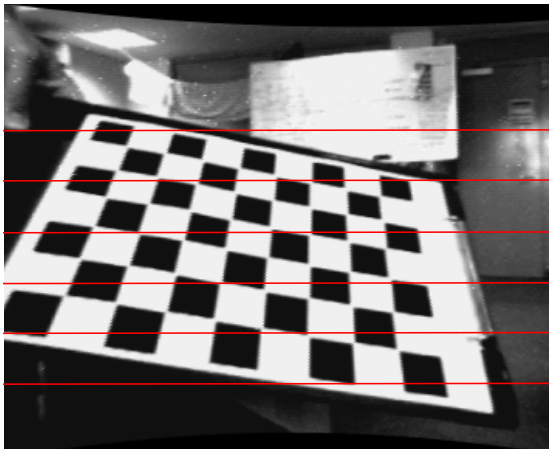
(a) Original Image from the Left Camera



(b) Original Image from the Right Camera



(c) Rectified Image from the Left Camera



(d) Rectified Image from the Right Camera

Figure 4.12: Images before and after the Rectification

5 The Harris Corner Detector

5.1 Disparity Calculation

Once our images are rectified we need to figure out which pixel p of one image belongs to pixel p' in the other image in order to calculate the depth of this point. Due to rectification we already know that the corresponding pixel p' has to be in the same row. To calculate the world coordinates of p we need to determine the disparity between p and p' (see chapter 3) thus we have to locate the pixel p' inside the row. Once we have determined the disparity in pixels we can create a so called disparity map from which we can conclude the world coordinates of the mapped pixels. According to [21] we can distinguish two kinds of disparity maps:

- **Dense disparity** maps result out of an area-based algorithm
- **Sparse disparity** maps as a result of a feature based algorithm

A dense disparity map can be obtained by area-based algorithms. According to the ordering constraint we can assume that the neighborhood of a pixel is constant over time and space. Therefore a window W around our point p is taken and the pixel intensities (grayscale or rgb) are correlated with a window W' placed in the second image (in the same row in case of rectified images). The ordering constraint tells us now that we will get the highest correlation if window W' is placed above the correlating point p'. Thus we have to move the window W' and determine the correlation between W and W' for every position in the row in order to find the position with the highest correlation. Possible correlation functions will be discussed in section 7.3. The problem of this algorithm is quite obvious: it will consume a lot of CPU time depending on the image size, the search radius, the correlation function and especially on the amount of points which have to be found in the second picture. This kind of brute force approach of course can be improved e.g. by dynamic programming, but the computation will still consume too much time as shown in chapter 7.
A hardware accelerated dense disparty map calculation requires much more resources than available inside the FPGA. The system presented in [9] allocates three times the amount of the logic cells we have in our Spartan 3E. Additionally the amount of block RAM exceeds the available 20 by more than 100 percent. Concluding from this and other available papers [24] the idea of implementing a dense disparity system on the Spartan 3E has been dropped and another solution has been considered.
A good way to improve the area-based method is the reduction of search space by gaining knowledge about the two pictures. One method to drastically reduce the search space is

to find features in both images and try to match only those features. This feature based method will result in a sparse disparity map meaning that we only can determine the world coordinates for the correlated features. This might appear to be a big drawback. However, if we have a look at the available resources of our EyeBot M6 as described in chapter 1.1 and done estimations, it is obvious that we have to accept a compromise.

A feature is defined as a distinctive point in an image which is likely to be found in the other image. At this point the question arises what such a feature might look like. Imagine two pictures of the same scene lying in front of you and your task is to pick points in one picture which you can easily recognize in the other one. You probably will not choose a wall or another plane area because there is no possibility to tell where a specific point on the wall of one picture can be found in the other picture. There will be the nearly the same problem if you choose a point on an edge to find in the second image. You will recognize the edge itself in the other picture but you will not be able to tell where exactly on this edge the point you picked in the first image is located. With regularly repeated patterns you will end up with the same problem. These difficulties are known as white wall and aperture problem.
The solution is quite simple: you have to pick a corner with unique properties which you can describe and compare with other found corners of the second image and finally match them because of their unique description or area-based correlation results. There are plenty of algorithms for corner detection, varying in complexity and quality. Based on evaluation results of the paper [8] the Harris Corner Detector [12] (as well known as Plessy Point Detector) which is one of the most popular algorithms for corner detection has been chosen. Following reasons speak for the Harris Corner Detector [16]:

- The operator is simple and suitable for automatic feature detection.
- The detected points are well proportioned and valid.
- The quantity of detected points can be determined by the users according to their requirements.
- The detected points are invariant to scale and rotation, and the operator is stable.

However, the main disadvantage of the algorithm is that the accuracy can only reach one pixel. In the following the Harris Corner Detector will be explained to afterwards step into the details of the hardware implementation of this algorithm.

5.2 The Harris Corner Detector

The Harris Corner Detector is based on the thoughts of Movarec [25] which are explained briefly. Movarec starts with a window centered at the pixel $p(x, y)$ and moves this window in the neighborhood of p. The changes are measured with the help of the auto-correlation function

$$f(x,y) = \sum_{x_k, y_k \in W} \Big(I(x_k, y_k) - I(x_k + \Delta x, y_k + \Delta y \Big)^2 \tag{5.1}$$

where (x_k, y_k) are points within the window W centered at point $p(x, y)$. Movarec claims that the changes behave the following way:

- Small changes will appear in all directions for a constant intensity in the neighborhood, representing e.g. a wall, floor etc. (1)
- Small changes in only one direction can be found for an edge whereas the direction of nearly no changes resembles the direction of the edge (2)
- Big changes in all directions will be observed for a corner (3)

Graphic 5.1 illustrates these three cases.

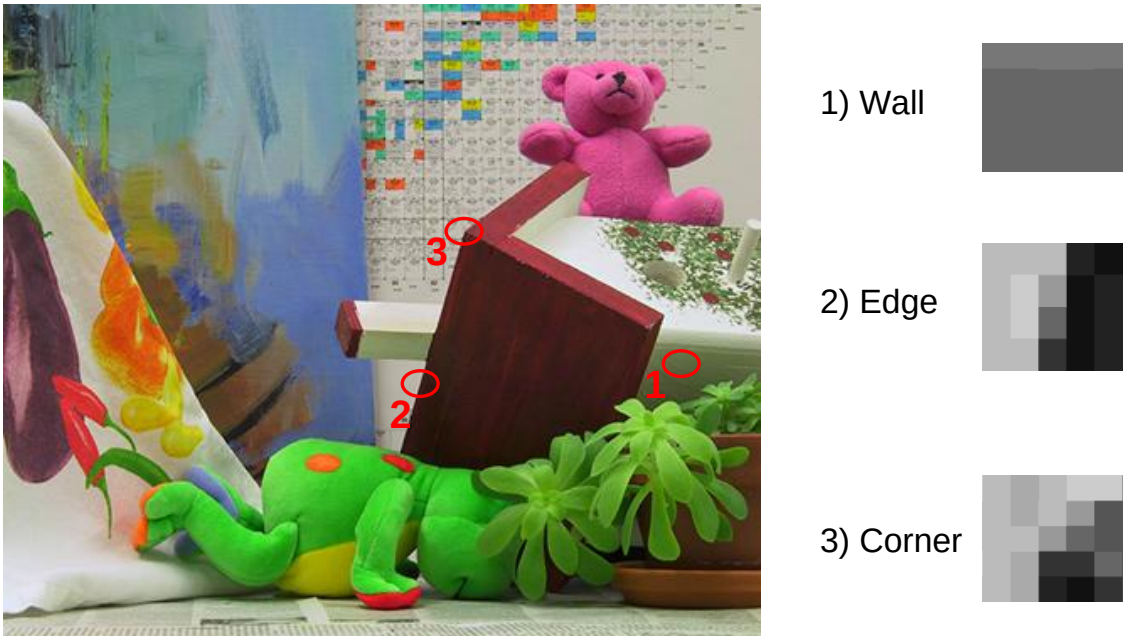


Figure 5.1: Examples for Wall, Edge and Corner [28]

This algorithm is computationally intensive because the window must be moved in all directions, and for every move the changes have to be calculated. To avoid the integration over shifts in all directions Harris introduced the auto-correlation matrix, which is derived from equation 5.1 in the following way:

Using the Taylor expansion $I(x_k + \Delta x, y_k + \Delta y)$ can be rewritten as

$$I(x_k + \Delta x, y_k + \Delta y) \approx I(x_k, y_k) + \begin{pmatrix} I_x(x_k, y_k) & I_y(x_k, y_k) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}, \tag{5.2}$$

where I_x and I_y are the derivatives of the image in the corresponding directions. If we now combine 5.1 and 5.2 we obtain

$$\begin{aligned} f(x,y) = & \sum_{x_k, y_k \in W} \left(- \begin{pmatrix} I_x(x_k, y_k) & I_y(x_k, y_k) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \right)^2 \\ = & \begin{pmatrix} \Delta x & \Delta y \end{pmatrix} \begin{bmatrix} \sum_{x_k, y_k \in W} I_x^2 & \sum_{x_k, y_k \in W} I_y I_y \\ \sum_{x_k, y_k \in W} I_x I_y & \sum_{x_k, y_k \in W} I_y^2 \end{bmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \\ = & \begin{pmatrix} \Delta x & \Delta y \end{pmatrix} \mathbf{G}'(x,y) \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}, \end{aligned}$$

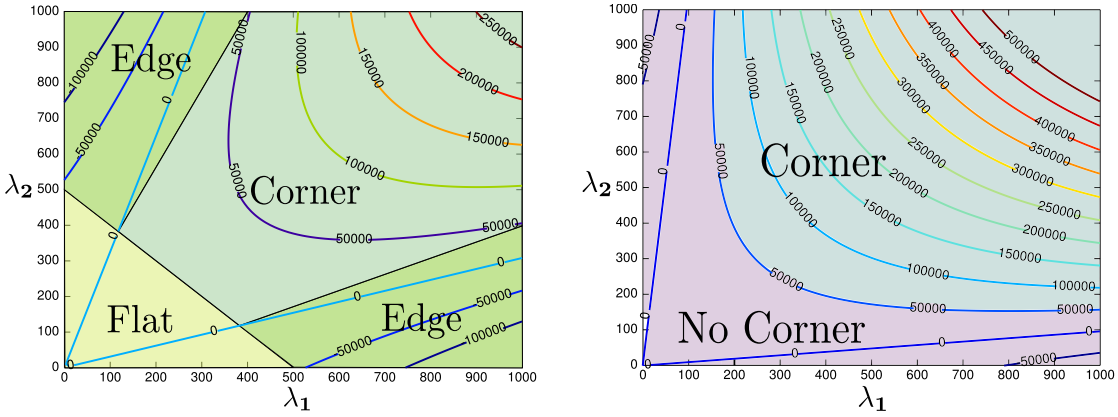
where $\mathbf{G}'(x, y)$ is the auto-correlation matrix. To reduce the localization error of the corner Harris furthermore recommended the usage of a Gaussian window instead of simply summing the values within the window W. That results in the weighted auto-correlation matrix

$$\mathbf{G}(x,y) = \begin{bmatrix} \sum_{x_k, y_k \in W} w_k I_x^2 & \sum_{x_k, y_k \in W} w_k I_y I_y \\ \sum_{x_k, y_k \in W} w_k I_x I_y & \sum_{x_k, y_k \in W} w_k I_y^2 \end{bmatrix}$$

The eigenvalues λ_1 and λ_2 of this matrix describe the changes inside our window similar to the moving window of Movarec:

- Small eigenvalues of $G(x, y)$ correspond with little changes in all directions and consequently with a plain texture. (1)
- One big eigenvalue tells us that we found an edge within the window. (2)
- A corner is described by two big eigenvalues. (3)

According to these assumptions the eigenvalue space can be ideally divided into a flat, edge and corner area as shown in figure 5.2(a). However, the direct calculation of the eigenvalues λ_1 and λ_2 is computationally intensive and therefore Harris used the following formula to derive the so-called cornerness



(a) Ideal Division of the Cornerness Space; $k = 0.18$

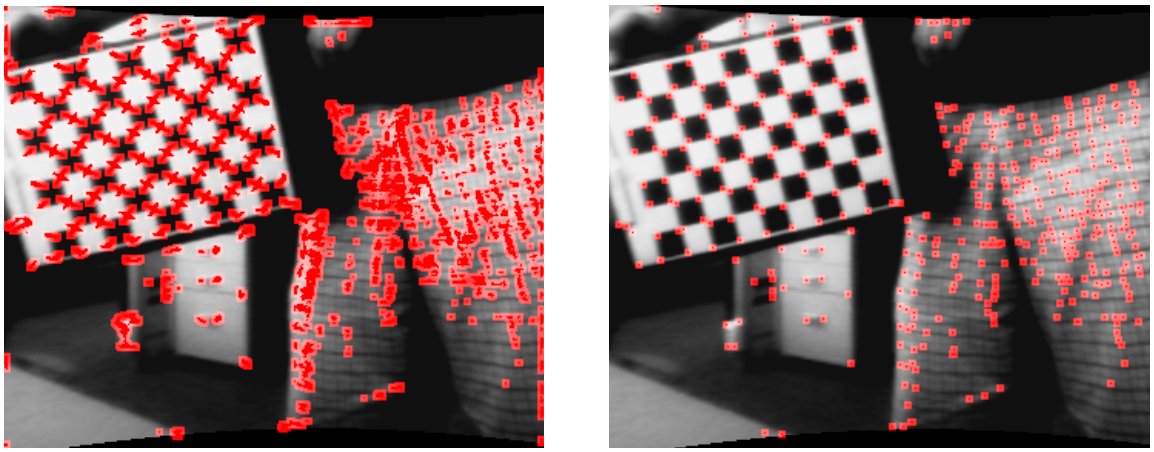
(b) Division of the Cornerness Space by Thresholding; $k = 0.08$

Figure 5.2: Harris Equicornerness Lines

$$\begin{aligned} c(x,y) &= Det(G(x,y)) - k \cdot trace^2(G(x,y)) \\ &= \lambda_1\lambda_2 - k \cdot \Big(\lambda_1 + \lambda_2\Big)^2 \\ &= \lambda_1\lambda_2\Big(1 - 2k\Big) - k\Big({\lambda_1}^2 + {\lambda_2}^2\Big) \end{aligned} \tag{5.3}$$

The Harris Cornerness $c(x, y)$ is plotted in figure 5.2(b) with the resulting equicornerness lines in the shape of parabolics. These lines approximate the ideal division of the eigenvalue space (marked areas).
The shape of the equicornerness lines can be changed with the parameter k in equation 5.3. The first term of the cornerness equation which is the product of the eigenvalues, is big if both eigenvalues are of medium magnitude or if one of the eigenvalues is really big. That will be the case if the window is above an edge. To avoid now that this case is declared as a corner the squares of the eigenvalues, each weighted by k, are subtracted. This ensures that a single but very large eigenvalue is not detected as a corner though it is an edge. The bigger the k the more the eigenvalues have to be of the same range and the less edges will be rated as corners. The influence of k on our equicornerness lines can as well be observed in figures 5.2(a) and 5.2(b). The bigger k is the more the parabolics get squeezed. To determine now if the cornerness of a point inside our picture is actually a corner or not we need to apply a threshold.

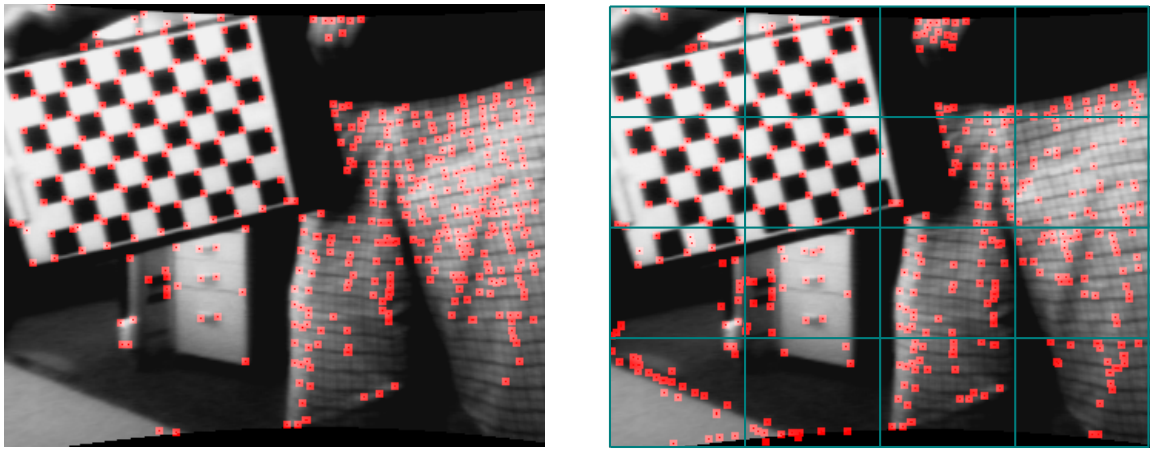
5.3 Thresholding



(a) Non-Maximum Suppression of Size 1×1 (b) Non-Maximum Suppression of Size 5×5

Figure 5.3: Harris Results with Global Thresholding

Thresholding the cornerness at the end of the algorithm corresponds to the selection of one of the equicornerness lines to divide the area into a corner and non-corner space (see figure 5.2(b)). A bigger threshold requires bigger eigenvalues and therefore bigger changes within our window. Figure 5.3(a) shows the result of simple thresholding. You can see that this thresholding method will give us a lot of feature-points for a single corner because there are always a lot of pixels with a high cornerness arranged around the actual corner. This makes it hard to exactly localize the corner.
The solution considered most of the times is a non-maximum suppression after the cornerness calculation. To perform such a suppression a window of a specific size is moved over the picture. If the center value of the window is the maximum within the whole window the filter response is exactly this value. Otherwise the filter response will be zero. As a result only the location with the highest cornerness value within the window which will correspond best to the real corner location will survive the filtering process. A common size of the filter is 3×3 or 5×5. After filtering the remaining local maxima are compared to a threshold and the location is marked as a feature or not. The results of a 5×5 non-maximum suppression can be seen in figure 5.4(a).
We will choose our threshold in a way that we get a specific amount of features for every picture. The problem with the application of a global threshold is the fact that features will not be distributed smoothly over the picture. If you have a look at figure 5.3(a) you will recognize that with a global threshold we get a lot of features around the chessboard and the person's trousers on the right side but the other areas are lacking of features. That is because we adapt our threshold to limit the total amount of features and thus only the strongest features will survive. In order to get smoothly distributed features we split up the thresholding process by dividing the image into blocks and thresholding each block on its own. By doing so we can guarantee that we find a certain number of features for every block. Figure 5.4(b) visualizes the result of this segmented threshold process.



(a) Pre-Thresholding 1×1, Non-Maximum Suppression 5×5

(b) Pre-Thresholding 4×4, Non-Maximum Suppression 5×5

Figure 5.4: Harris Results with Pre-Thresholding

5.4 Summary

In summary we need to perform the following steps for Harris Corner Detection:

- Differentiate the input image in direction x and y
- Calculate the Gaussian window over the neighborhood of the current pixel
- Calculate the cornerness according to equation 5.3
- Apply non-maximum suppression
- Segmentate and threshold the data

As a result we will get a set of features for each camera which can be matched in order to determine the disparities.

Function	EyeBot M6 [fps]
Sobel Unseparated	48.45
Sobel Separated	55.6
Window 5x5	26.3
Window Separated	48.7
Cornerness Calculation	58.9
Non-Maximum Suppression 3x3	56.47
Non-Maximum Suppression 5x5	29.37
Harris with NM 5x5	7.37

Table 5.1: Software Performance of the Harris Corner Detector

The Harris Corner Detection has been implemented in software on the EyeBot M6. The resulting frame rates of the single processes and of the whole Harris are listed in table 5.1. It is conspicuous that the separation of the Sobel does not have the expected effect of approximately doubling the frame rate as it can be observed for the window function. One guess is that the maximum frame rate is limited by the speed of the SDRAM accesses which are necessary to load the image data. The maximum achieved frame rate resulted in 7.39 fps for a Harris with 5×5 non-maximum suppression. This frame rate however does not include the transfer of the rectified pictures from the FPGA into the SDRAM of the CPU which would be necessary. The achieved frame rates of course can be increased a lot by optimized programming but a high amount of CPU time will still be spent for the image processing. Therefore the Harris Detector has been implemented in hardware. The implementation details are described in the following chapter.

6 Harris in Hardware

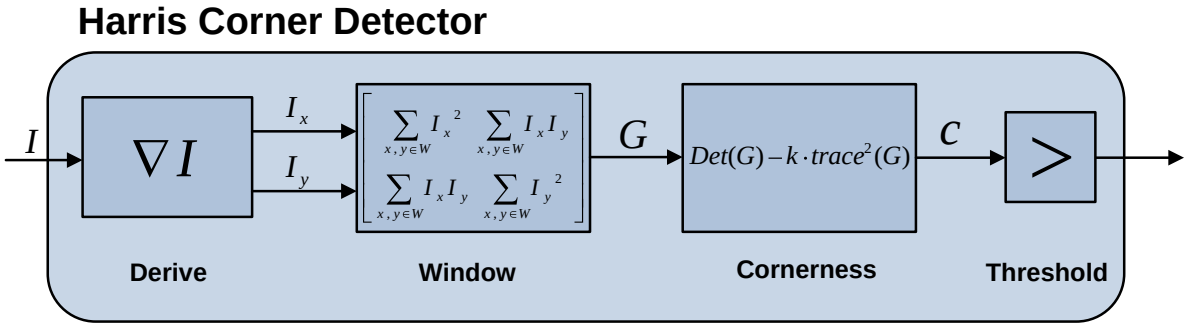


Figure 6.1: Modules of the Harris Corner Detector

Figure 6.1 gives an overview of the implemented Harris Detector. The system is divided into four subsystems according to the steps described in chapter 5. During my whole project one important aspect was to achieve a high portability and reusability. Three of the four modules (sobel, windowing and the non-maximum suppression) are basic image operations and are therefore widely in use. The reusability of the developed entities is seen to be very important for further research. Therefore a global interface and protocol has been developed which connects all of the modules shown above with each other.

6.1 Module Interface

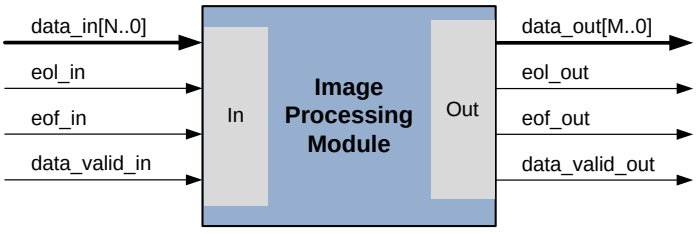


Figure 6.2: Interface of the Image Processing Modules

Four signals are used to control the flow between modules and are used for both, the input and the output of the modules: data, data_valid, end_of_line and end_of_frame (see figure 6.2). All signals are defined as active high and are reckoned to be valid at the rising edge

of clock. First and most important of all the data signal. Its width varies depending on the operation which has to be performed. The data at the input and output respectively is valid when the data_valid signal is high. A low data_valid at the input of a module notifies the module that the data and the other signals are not valid.
There are two signals which describe the current location inside the image. The end_of_line signal (eol) which is sent with the last data word of an image line notifies that with the next word a new line is started. The end_of_frame signal (eof) which goes high with the last data word of the frame notifies the module that the next incoming pixel will be of the next frame. The end_of_frame and end_of_line signal both have to be high at the same time as shown in the timing diagram 6.3. In order to initiate a reset and bring the modules into a defined state after the start-up of the FPGA a valid and high eof in combination with a low eol has to be sent. Once the reset signal is detected it is forwarded immediately to the following modules to guarantee that all modules are initialized before the first valid data arrives. This protocol will be referred to in the following as image stream.

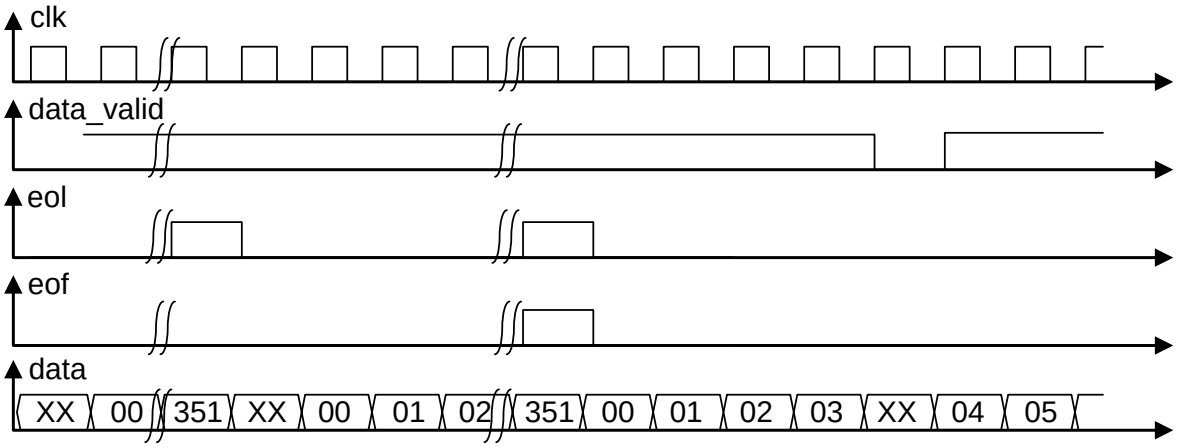


Figure 6.3: Image Stream Signal Timing

Now that the interface of our modules is defined we will have a look at the modules themselves. The description of the implemented Harris Corner Detector is split in two parts. The first part describes the mono camera case whereas the second part describes how resources can be saved by merging two mono Harris modules to a stereo Harris.

6.2 Mono Harris

The first subsystem of our Harris Detector performs the derivation of the image data. This derivation is calculated with a 2D convolution as described in the following.

6.2.1 Convolution in Hardware

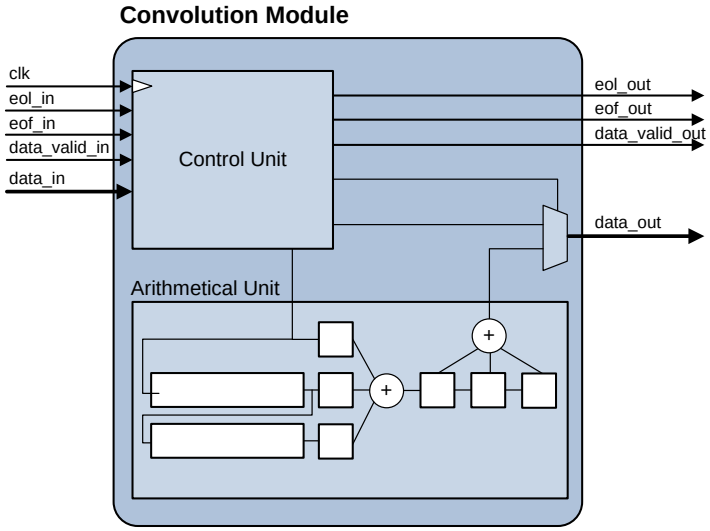
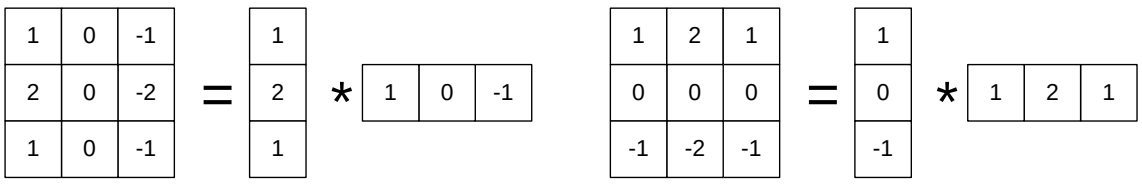


Figure 6.4: Convolution module overview

A 2D-convolution of an image I with a filter kernel h is described with the following formula:

$$I'(x,y) = I(x,y) * h(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(m,n)h(x-m, y-n),$$

where $M \times N$ describes the size of the kernel. A typical kernel is the Sobel kernel as shown in figure 6.5.



(a) Separated Sobel in x Direction

(b) Separated Sobel in y Direction

Figure 6.5: Sobel Kernel

With this kernel the necessary derivation of I in x or y direction are calculated resulting in I_x and I_y. The 2D convolution is of the complexity $O(M * N)$ per pixel in regard to multiplications and additions thus the calculation in software is quite time consuming. In hardware however the convolution can be performed in stream and parallel as described in the following.

Arithmetical Unit

For the derivative of I and therefore the calculation of a pixel $I_x(x, y)$ resulting out of the convolution with a Sobel kernel (see figure 6.5) nine values of the image are needed which lie all inside the 3×3 window centered at the position (x, y). To gain these pixels at the same time, the incoming pixels need to be delayed. Figure 6.6 illustrates how the convolution can be performed within our image stream.
However, many important image filtering kernels as the Sobel are separable. For these

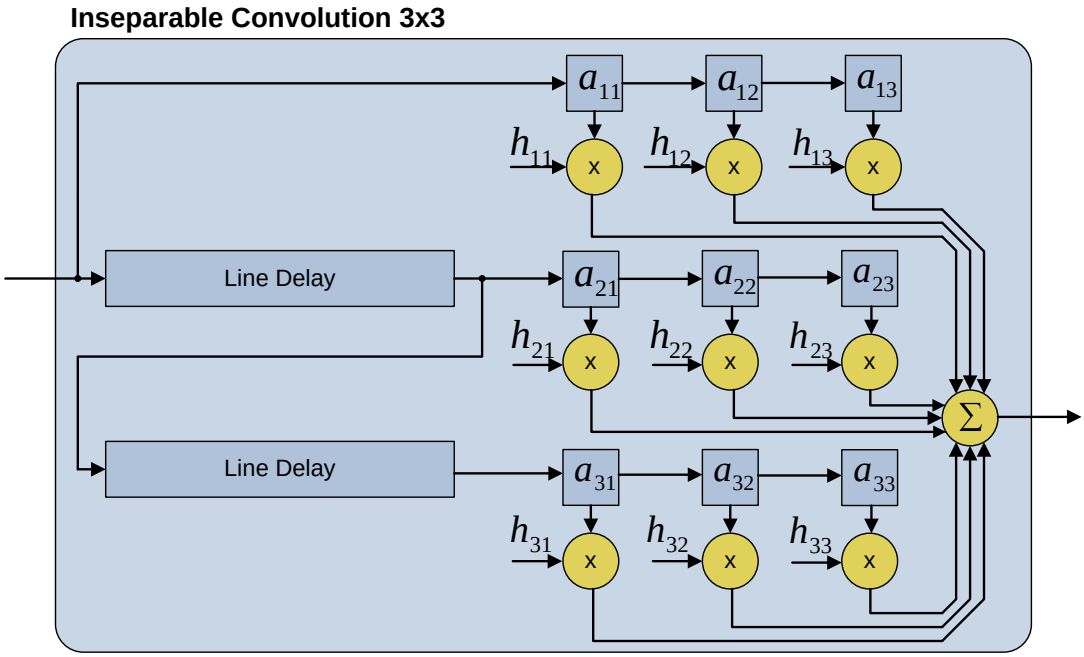


Figure 6.6: Implementation of an inseparable 3×3 Filter

kernels formula 6.2.1 can be rewritten the following way.

$$I'(x, y) = I(x, y) * h(x, y) \tag{6.1}$$
$$= I(x, y) * f(x, y) * g(x, y) \tag{6.2}$$
$$= \sum_{m=0}^{M-1} \left(\sum_{n=0}^{N-1} I(x - n, y - m) \cdot f(n) \right) \cdot g(m) \tag{6.3}$$

By doing so the complexity can be decreased from $O(M * N)$ to $O(M + N)$ per pixel. The used Sobel and the later on implemented Gaussian window function are both separable and therefore implemented as in figure 6.7.

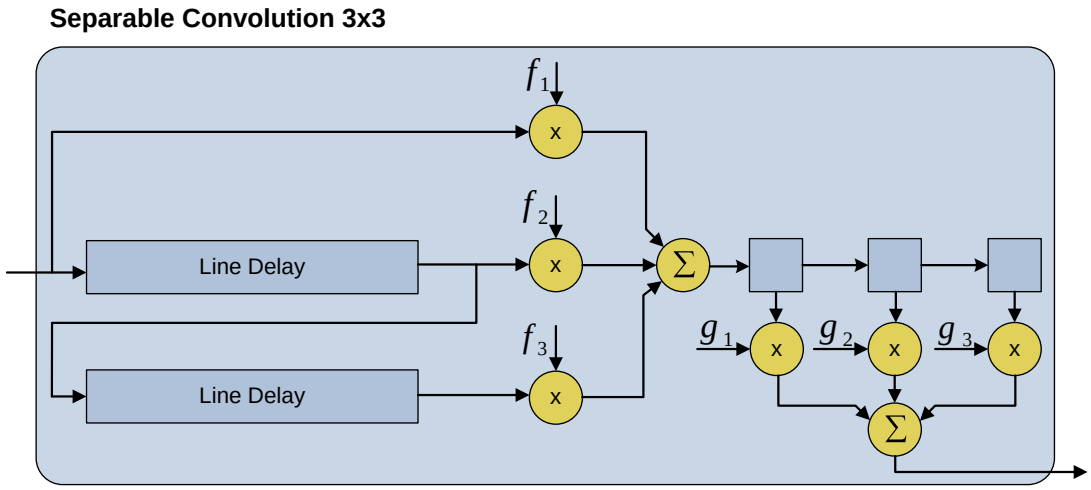


Figure 6.7: Implementation of a separable 3×3 Filter

One can tell the advantage of using separable filters directly from the figures. Table 6.1 depicts the decrease of resources needed in numbers for implemented kernels. It has to be noted that these figures include all the control logic for border treatment and producing the control signals as well (see section 6.2.1).

Module	LUTs (Unseparated)	LUTs (Separated)
Sobel	314	259
Window	1241	843

Table 6.1: Resource Consumption of separated and unseparated Convolution

Based on the implementation as shown in figure 6.7 the operation mode is explained in the following.
A line delay delays the pixels for a whole line of the image, whereas the registers delay the pixel just by one time step. The incoming pixels are shifted into the first line delay one by one. As soon as the first one is full the output is enabled and the next line delay is filled. At every entry point of the line delays and at the output of the last line delay multipliers are connected. The current incoming pixel, the pixel which is delayed by one line and the pixel which is delayed by two lines (for a kernel size of 3×3) are then multiplied with the appropriate coefficients of the 1D kernel f and are afterwards accumulated. This construct so far calculates the inner sum of equation 6.3. In order to calculate the outer sum the values are registered after the first accumulation and one by one shifted with every new incoming pixel. The registered values are then multiplied with the coefficients of kernel g and are finally summed up to the result of the convolution.

Line Delays

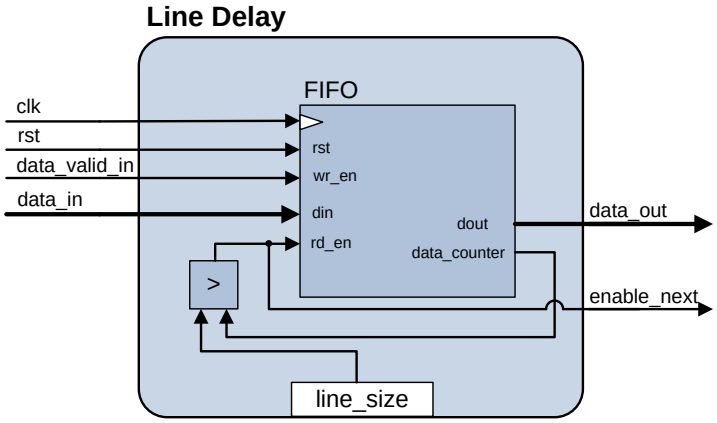


Figure 6.8: Line Delay Module

The line delays have been realized with the internal block RAM of the FPGA. One block RAM is of the size 18K which can only be divided in a specific way. With a image resolution of 352×288 the resulting FIFO has to be at least of the size 352. The physical depth of the FIFO can be only to the power of 2, therefore we need a depth of 512 words for our block RAM. One word is consequently at most 36 bit wide and we will need for each line delay one of the 20 block RAMs.
Xilinx ISE gives the user the possibility to automatically generate FIFOs which access the block RAM with the FIFO-Generator. This FIFOs are the main-element of the line delays. As you can see in figure 6.8, signal data_valid is used for enabling the line delays whereas the internal data-counter is used for enabling the data output and enabling the next module. The filling status signals offered by the Xilinx FIFO Generator are not used due to a strange reset and overflow behavior. Instead the internal counter is compared to a fixed value, the line width. Once the line delay is full and additional data is coming in, it acts like a shift register and the pixel which entered the FIFO at first leaves the line delay with a high enable_next signal. The FIFO is implemented as a First Word Fall Trough FIFO meaning, that the data already lies at the output before the read signal rises at the input of the FIFO. Without doing so we would have to introduce an additional delay of one clock cycle. After an end_of_frame signal is received the data inside the FIFOs is no longer needed because valid data can not be calculated anymore (see section 6.2.1). Therefore the eof signal is used as a synchronous reset signal. It has to be noted that for the generation of the FIFOs the FIFO Generator v4.4 was used after a bug in v4.3 was found during my development, which does not allow to reset the First Word Fall Through FIFOs completely.

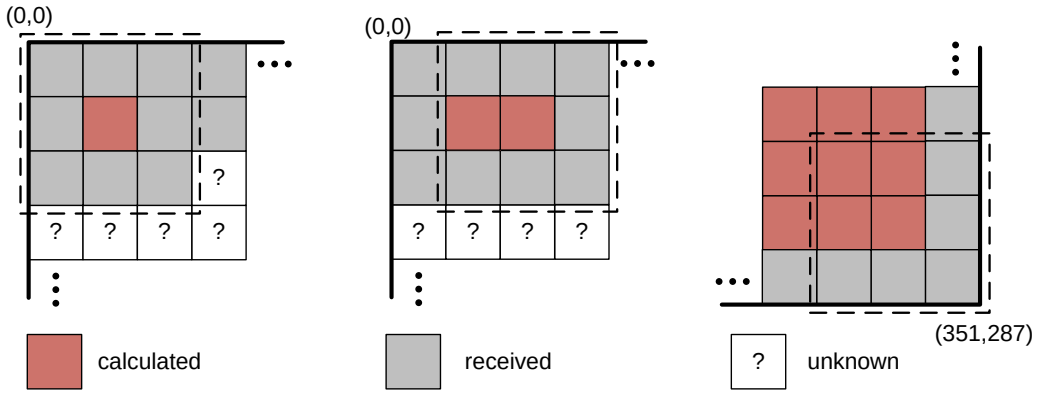


Figure 6.9: Movement of the Window inside the Image

Border Treatment

A problem of the convolution is that valid data can only be produced if the filter window lies above valid values as shown in figure 6.9. According to this graphic the data at the output of our $n \times n$ convolution unit becomes valid after $n - 1$ lines and n pixels have entered the unit. This however is only true if the filtering window does not touch the borders of our image. There are three different possibilities of how to treat the borders of an image.
One method which is often applied in software is to mirror the borders. With this method it is possible to calculate values at the border of the image. However, the quality of these values is doubtful and furthermore it does not seem to make sense to put additional effort into performing these calculations. If you have a look at the given distortions of the cameras you will recognize that the error is higher the closer we get to the borders of a picture (see chapter 3). Because of this the mirror method of the borders was discarded.
Another possibility is to ignore the borders completely and crop the image. The data_valid_signal would then have to switch to low at the borders which could be done quite easily. Yet, one main aspect of the programmed modules is a high reusability and portability. If the images are cropped for every convolution we would have to recalculate the image coordinates at the end of the Harris Corner Detector. Additionally the segmentation of the image for block based thresholding as described in section 3 will become more difficult because the image size can vary depending on the kind and number of filters used before.
The third way of treating the borders is to output the same image size as the incoming one with random data at the borders. This results in a high portability and reusability with a reasonably additional effort compared to the cropping. We will only need a counter to set data_valid high for the right amount of clock cycles after a eof has been received and which works independently from the input control of the convolution unit. Thus we have to add a control unit which generates the signals data_valid, eol and eof as we need them to the arithmetical unit as shown in figure 6.7. These outgoing image stream signals

are created inside the control unit out of the incoming signals. Depending on the number of the received incoming eols we will pass through a delayed version of this signal or not. Furthermore we start outputting data signaled with a set data_valid after a specific amount of pixels has entered our arithmetical unit. This filling status is monitored by the line_delay_enable_next signals which notify that the line delay is full. For example for a 3×3 convolution we need to have output one line and one pixel of data before the actual valid data at position $(2, 2)$ is calculated and lies at the output. How the signals are generated in detail is not further described here. The developed control unit can be seen in figure 6.4.
The derivation of the image with a Sobel kernel (separable) and the window function are both based on this convolution unit. The applied 3×3 Sobel as shown in figure 6.5 can be implemented without any multipliers because the kernel elements are either one or two what results in simple shifting and adding of the data. The calculation of I_x and I_y can be done with a single set of line delays. Only the calculation itself and the pixel delays after the line delays have to be implemented twice because of the different kernels. In order to keep the resource consumption small quantization was applied during the calculations. Thus at the output of the Sobel unit we get values in the format `sfix[9,-3]`.

6.2.2 The Window Function

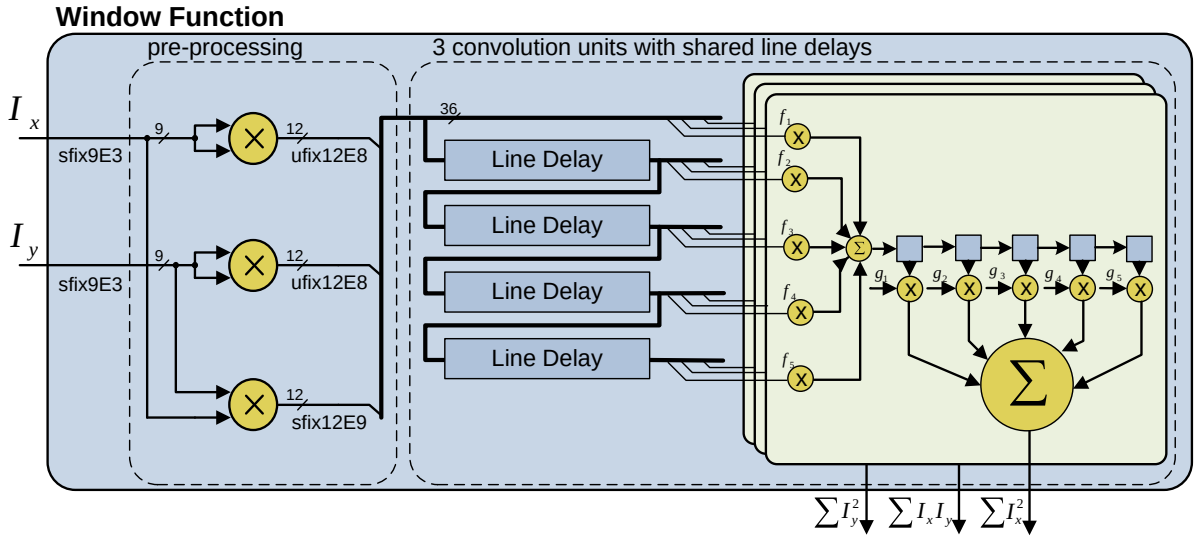


Figure 6.10: Window Function

In the next step the autocorrelation matrix

$$\mathbf{G} = \begin{bmatrix} \sum_{x \in W} I_x^2 & \sum_{x \in W} I_y I_y \\ \sum_{x \in W} I_x I_y & \sum_{x \in W} I_y^2 \end{bmatrix} \in \mathbb{R}^{2 \times 2}$$

has to be calculated out of the results of Sobel. For calculating the sums at first a simple summing convolution kernel was used. Later on a weighting of the elements has been introduced to increase the corner location accuracy. The farer a pixel is away from the center of the summing window the less influence it should have. The following kernel gave, compared to a real gaussian, the best results.

$$\frac{1}{128}\begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 16 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix} = \frac{1}{128}\begin{bmatrix} 1 \\ 2 \\ 4 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \end{bmatrix} \tag{6.4}$$

A real gaussian was not used to save the remaining multipliers of the Spartan 3E. Due to the fact that the coefficients are all of the power of 2 the design does not need much more resources than the simple summing, because the multiplications can be performed again by simply shifting the data.
Before we convolve we need to calculate the squares of the incoming data. To reduce the number of multipliers the incoming data from the Sobel module is multiplied before it enters the delays. After the calculation of I_x^2, I_xI_y and I_y^2 this data is quantized. The resulting data is theoretically 20 bit (+1bit for the sign of I_xI_Y) wide. Due to the quantization after the sobel to 9bit signed the resulting width of our multiplication is reduced to 16bit (+1). Therefore, we would need 8 block RAMs to delay three 16bit streams for a window of the size 5×5. Two results of the multiplications can be put together into one block RAM and and the third result would occupy four block RAMs on its own. Altogether we would then already need 10 out of 20 block RAMs just for the calculation of the derivative and the windowing. One option would be to execute the multiplication after delaying the data, but then we would need $5 * 3$ multipliers at the output of the delay module, what is again definitely too much for our FPGA. By sticking to the multiplication at the entry point of our delay module and reducing the data width of the multiplier results to 2×12 bit unsigned data and 12 bit signed the data can be packed into a 36bit word and put into one delay module together. This method seems to be the better option than using 4 additional block RAMs or implementing 12 additional multipliers. However, the loss of accuracy is the price we have to pay. The reduction has been simulated in Simulink and the influence on the resulting feature locations was not noticeable.
Because the pseudo Gaussian kernel 6.4 is separable the summing at the output of the registers is implemented, according to figure 6.7, just with two more line delays and 16 additional registers. Before the delayed data is actually convolved it has to be unpacked again. The resulting window function with the used bit width can be seen in figure 6.10.

6.2.3 Calculating the Cornerness

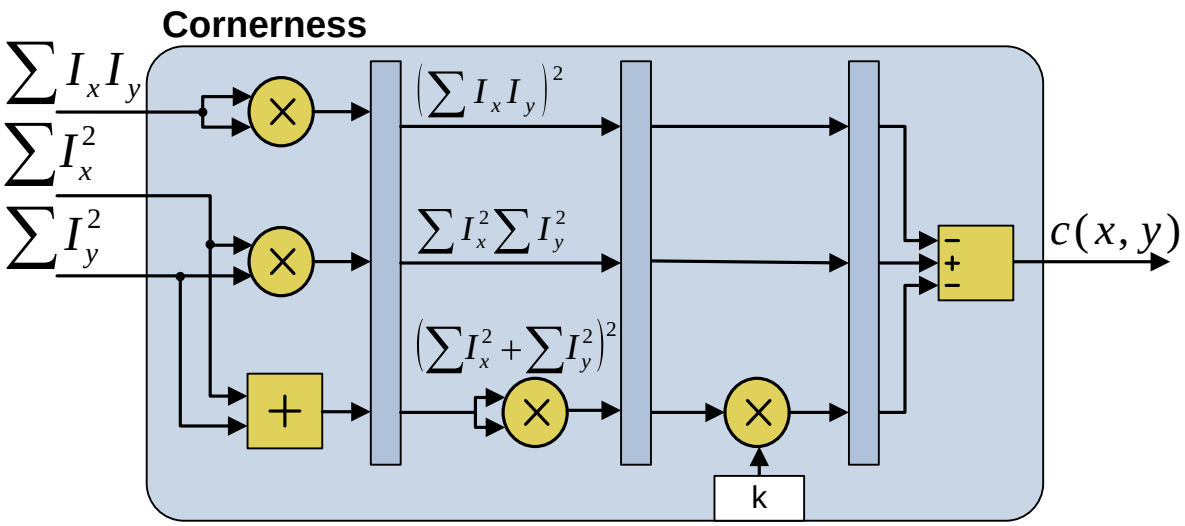


Figure 6.11: Cornerness Calculation

After the calculation of the autocorrelation matrix we need to determine the cornerness as described in chapter 5. The following equation has to be solved in hardware:

$$c(x,y) = \sum_{x_k,y_k \in W} I_x^2 \sum_{x_k,y_k \in W} I_y^2 - \left(\sum_{x_k,y_k \in W} I_y I_y \right)^2 - k \left(\sum_{x_k,y_k \in W} I_x^2 + \sum_{x_k,y_k \in W} I_y^2 \right)^2$$

To keep the maximum path length of the hardware implementation short the calculation was fully pipelined as shown in figure 6.11. The data paths are furthermore quantized to ensure that the inputs of the multipliers do not exceed the maximum bit width of our hardware multipliers which is 18bit. The picked quantizers are a result of bit-true simulation in MATLAB and Simulink with various pictures. Based on histograms and overflow logging the quantizers were changed manually until a good compromise between accuracy and resource consumption was found. The automated floating to fixed point conversion as described in chapter 2.5 has not been applied on this problem due to the high simulation times. Additionally it turned out to be difficult to evaluate the quality of the results. This feedback is however necessary for the automated quantization. Therefore the bit widths were determined manually and the final cornerness result is of the type `sfix[20,-16]`.

6.2.4 Thresholding

The process of thresholding is separated in two steps, the pre-thresholding and the non-maximum suppression as it can be seen in figure 6.12. The pre-thresholding compares

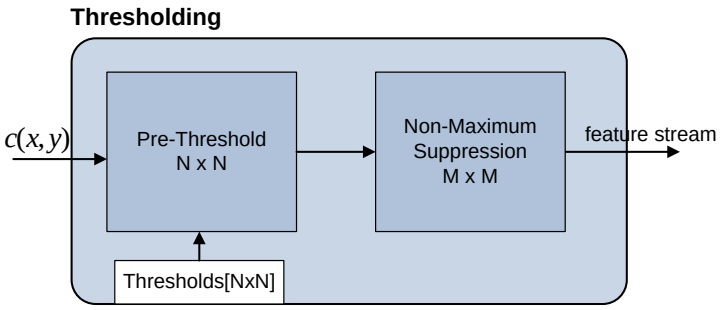


Figure 6.12: Thresholding Overview

the cornerness values to the appropriate thresholds stored in registers and acts as a pass-through if the value is bigger or outputs zero if not. As described in section 5.3 a segmentation of the image gives better results. Thus a segmentation of the image by 4×4 with a resulting block size of 88×72 has been implemented in hardware. The 16 thresholds for the blocks can be written by the CPU using the peripheral bus [10] and thereby the number of features can be regulated with a delay of one image. The implementation of the segmented threshold unit has been solved straight forward with pixel and eol counters controlling a huge multiplexer in order to switch between the different thresholds saved inside registers.
After the pre-thresholding only the values which are bigger than the threshold are not zero. As explained in section 5.3 a non-maximum suppression is now necessary to avoid that more than one feature is returned for a corner. The pre-thresholded cornerness stream is transformed to a binary feature stream with this unit telling whether the current pixel is a feature or not. The easiest way to implement such a suppression is the use of a convolution delay of an inseparable filter kernel as described in section 6.2.1. Instead of the multipliers and accumulators we use comparators which compare the surrounding cornerness values with the value of the center. If the center cornerness is the maximum inside the window and is bigger than zero we have found a corner and our feature stream output is set to one, otherwise to zero. However, this straight forward implementation already needs 8 comparators with a width of 20 bit each for a 3×3 non-maximum suppression.
The non-maximum suppression can be separated similar to the convolution kernels discussed above. The resulting implementation is shown in figure 6.13. After the data has been delayed by the line delays the maximum value in y direction has to be determined. This maximum value is then delayed with the information whether the maximum was the center value or not. If the maximum is not the center value we already know that the current filter answer and likewise our feature signal will be zero. If the center value is the maximum in y direction we have to compare this value with the other maxima in x direction as shown in figure 6.13. If the center value is as well the maximum in x direction we found a local maximum inside our window and the feature signal is set to one. With the separation of the non-maximum suppression we save resources as shown in table 6.2.

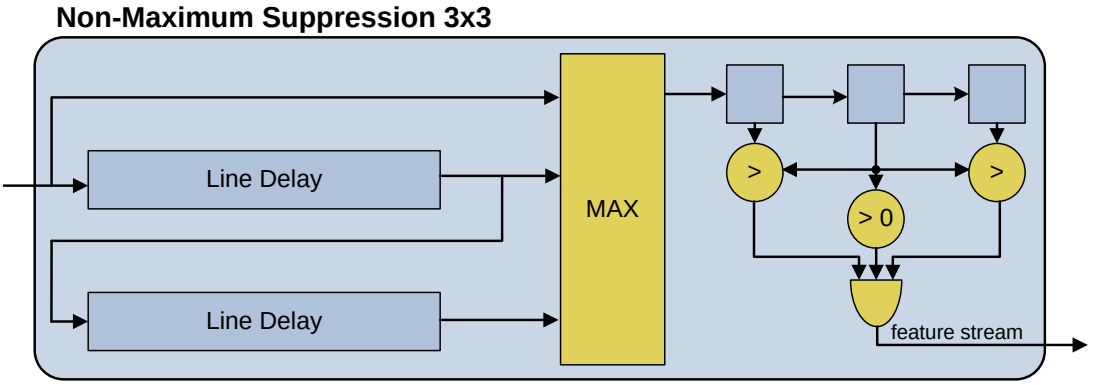


Figure 6.13: Separated Non-Maximum Suppression

Module	LUTs (Unseparated)	LUTs (Separated)
Non-Maximum 3x3	419	381
Non-Maximum 5x5	1011	706

Table 6.2: Resource Consumption of separated and unseparated Non-Maximum Suppression

6.2.5 Summary

The described implementation is capable of an in-stream Harris Corner Detection. The maximum frequency given by the synthesize tool is 65 MHz no matter which kind of non-maximum suppression is used at the end. All different combinations of the Harris have been tested in hardware with a clock rate of 50 MHz and worked successfully. Due to the fact that the CPU reads are too slow to get the data in stream from the FPGA and there was no SRAM controller at the time when the testing was done the cameras were neglected because there was no possibility to store the results of the Harris Corner Detector. Instead two simple FIFOs were used to validate the design. One FIFO was filled by the CPU with test images while the other one was emptied as soon as data was available. The resource consumption of this design is given in table 6.3. Note that the slice consumption also includes the interface structure built up for testing purpose but can be neglected compared to the consumption caused by the Harris. The number of block RAMs does not include the two IO FIFOs in order to avoid confusion.

As you can tell from table 6.3 the current design will in any case need seven of the twenty available multipliers. If we now double the design in order to perform Stereo Harris Corner Detection we will run out of multipliers because the rectification described in chapter 4 already allocates eleven multipliers. Additionally a lack of block RAMs will occur if we want to be able to perform 3×3 or 5×5 non-maximum suppression in stereo. Two of the twenty available block RAMs will be used to cross the clock domain between camera and FPGA and additionally two block RAMs are needed to run burst transfers between CPU and FPGA. Thus we will end up with 20 or 24 necessary block RAMs.

Module (Mono)	LUTs	block RAM	MUL
Sobel	259	2	0
Window	843	4	3
Cornerness	132	0	5
Pre-Threshold 1x1	20	0	0
Pre-Threshold 4x4	197	0	0
Non-Maximum 1x1	7	0	0
Non-Maximum 3x3	381	2	0
Non-Maximum 5x5	706	4	0

Table 6.3: Mono Harris Resource Consumption

Yet, the rectification unit is not capable of outputting a pixel every clock whereas the current Harris is programmed to be able to process pixels at full speed. The rectification unit which can be clocked with 100MHz has to load four pixels out of the SRAM to produce one pixel at the output. If we assume a delay of two clock cycles we will end up with a break of at least 6 clock cycles at 100 MHz between two pixels at the input of the Harris. The Harris Corner Detector will run at 50 MHz what results in at least 3 clock cycles of time for every pixel. The idea is now to use the break of three clock cycles and share resources inside the Harris. Based on these thoughts two of the above described Harris Corner Detectors have been merged to a Stereo Harris.

6.3 Stereo Harris

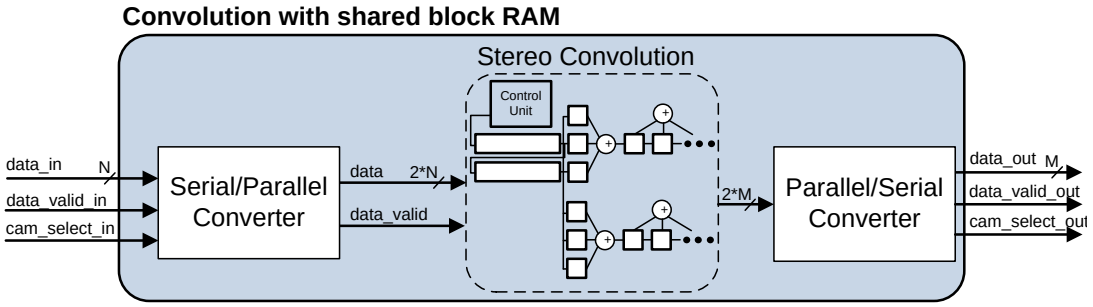


Figure 6.14: Stereo Convolution with shared Line Delays

The interface of the modules as described in section 6.1 first of all had to be extended to allow the module to distinguish from which side, left or right, the pixel is coming. The resulting signal cam_select_in is defined to be low for a pixel from the left side and high if the pixel is from the right side. All modules are extended with such an input and output. The first element which is taken into consideration is the Sobel. The Sobel uses two block RAMs for line delays but only 8 bit of the possible 36 bit word width are used at the moment. The idea is now to delay the left and right pixels at the same time

within one word. Due to the fact that the pixels arrive not at the same time a serial to parallel converter was developed which delays the left pixel and outputs the left and right pixel within one word as soon as the right pixel arrives. This double word then can be delayed with the block RAM and will be separated at the output in order to perform the Sobel calculations for each side. At the output of the Sobel the still parallel data stream is transformed back into the serial format. The resulting stereo convolution module is shown in figure 6.14.

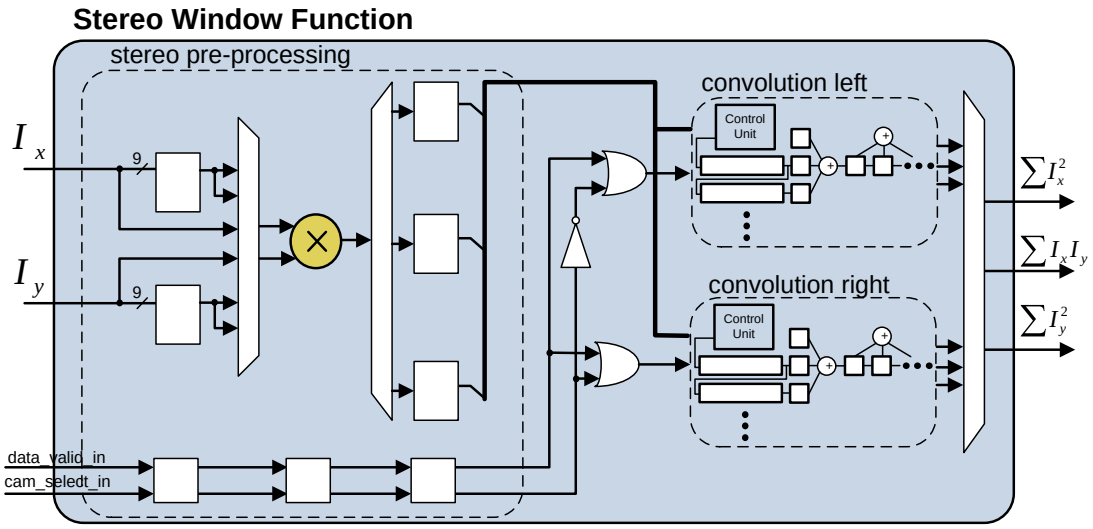


Figure 6.15: Stereo Window Function

The next module is the window function. At the input of this function the incoming I_x and I_y have to be multiplied to gain $I_x I_y$, I_x^2 and I_y^2. The original Mono Harris does this within one clock cycle and consequently uses three multipliers. The idea is now to use one multiplier for all three multiplications what will exactly take the available three clock cycles between incoming pixels. The resulting stereo window function can be seen in figure 6.15. The data_valid_in signal is delayed together with the other control signals for the necessary clock cycles. Data_valid_in and its delayed signals are used to switch the multiplexers at the in and output of the multiplier. Once all registers at the output are filled the data is passed to one of the two convolution units depending on the delayed cam_select_in signal. For the window function the block RAM cannot be shared due to the fact that the full bit width of 36 bit is already used.

After the windowing the cornerness has to be calculated. The Mono Harris Corner Detector uses four multipliers to calculate the cornerness in stream. This number can be reduced if we take into consideration that we have three clock cycles of time before the next pixel enters the cornerness unit. The stereo cornerness as shown in figure 6.16 uses only two multipliers instead of eight which would be necessary if we doubled the Mono Harris.

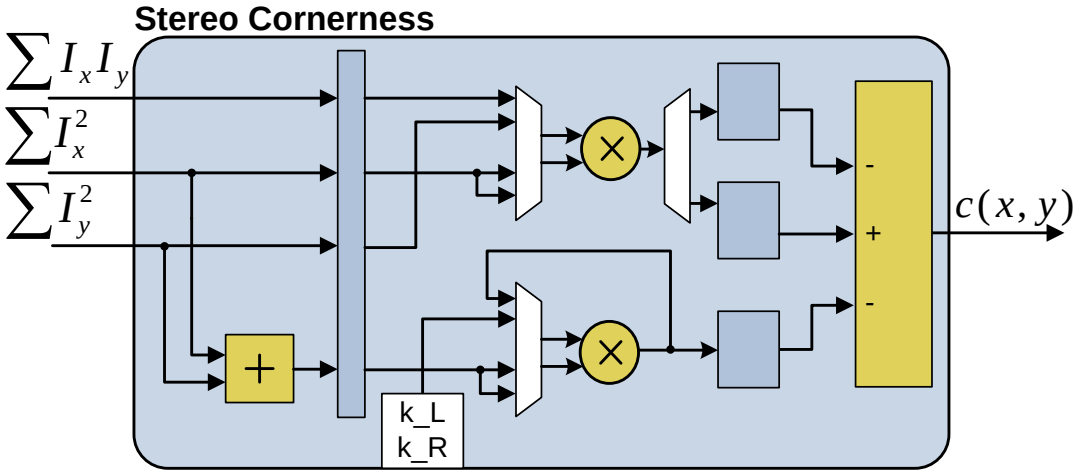


Figure 6.16: Stereo Cornerness

At the end of our Harris thresholding has to be performed. The comparators of the pre-thresholding units can be shared for the stereo case by simply switching between the thresholds of the left and right side. The final non-maximum suppression is implemented like the stereo Sobel with shared block RAM (see figure 6.14).

The final resource consumption of the Stereo Harris Corner Detector is listed in table 6.4. Compared to the Mono Harris the logic consumption nearly doubled what seems to be no improvement at all. However, the number of block RAMs used was reduced from 16 to 12 by sharing them in the modules Sobel and non-maximum suppression 3x3. The number of used multipliers has been drastically reduced from 16 to 3 because of the fact that the results do not have to be available after every clock cycle anymore. Consequently the Stereo Harris fits into the FPGA together with the Rectification Unit.

Module (Stereo)	LUTs	block RAM	MUL
Sobel	374	2	0
Window	1818	8	1
Cornerness	266	0	2
Pre-Thresholding 1x1	34	0	0
Pre-Thresholding 4x4	477	0	0
Non-Maximum 1x1	6	0	0
Non-Maximum 3x3	538	2	0

Table 6.4: Stereo Harris Resource Consumption

7 Image Processing System

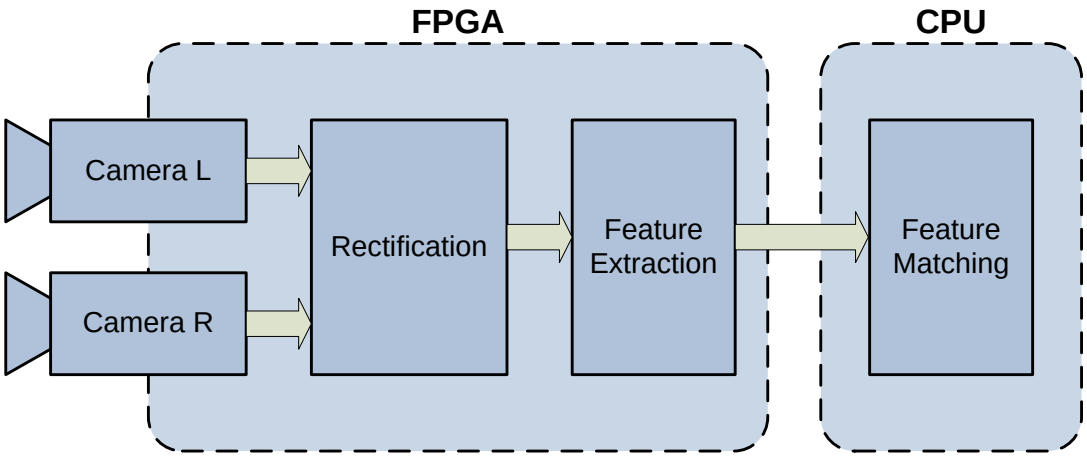


Figure 7.1: Image Processing System - Overview

In the following chapter the complete image processing system designed during this thesis will be described. Figure 7.1 gives an overview of the system. The first three steps of the image processing are done inside the FPGA. Two of the three processes, the rectification and the Harris Detector, have already been described in the previous chapters. In the following will be explained how the third step, the frame grabbing works and how these three processes can be finally combined in hardware to form a complete system.

7.1 Hardware Based Image Processing

Now that all modules of the FPGA related image processing are described the structure of the resulting complete system is defined. Figure 7.2 gives an overview of the image processing system in hardware. With the help of this figure the data flow will be described below. As already mentioned, the SRAM controller was not ready when this thesis has been handed in. However, the controller has been planned and documented in [10] and speed estimations can therefore be given. The numbers given in the following are based on these estimations.
The cameras are each clocked with their own oscillator and therefore are not synchronous to the FPGA clock. The clock domain crossing is achieved with the help of dual port block RAMs. Before the data enters the FIFOs the signals of the camera are transformed

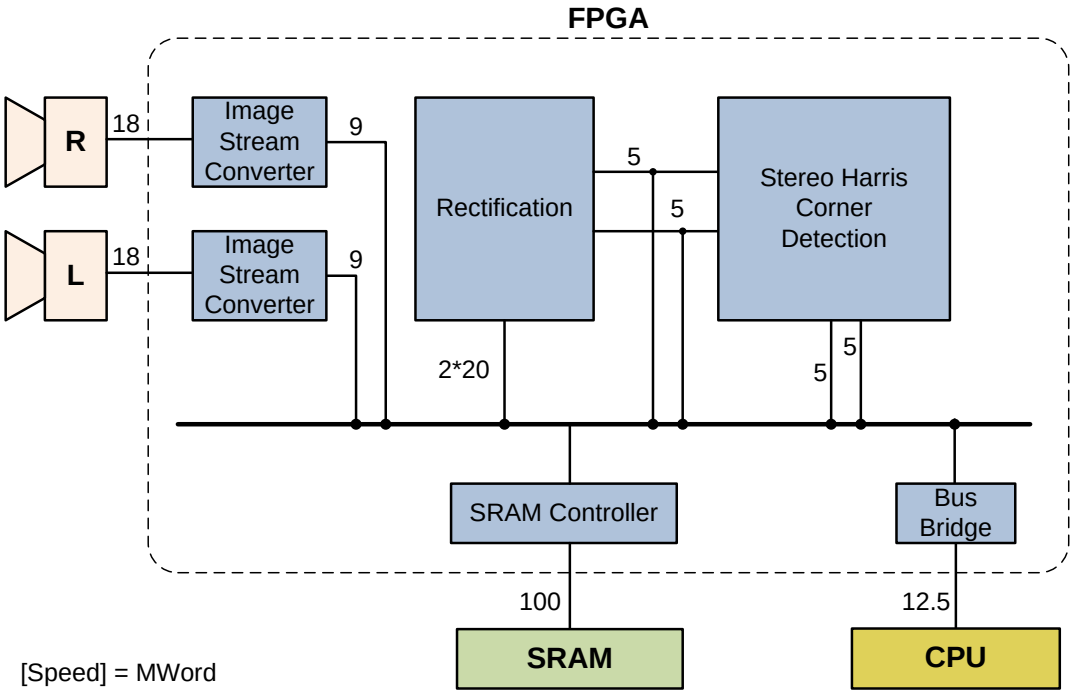


Figure 7.2: Image Processing System

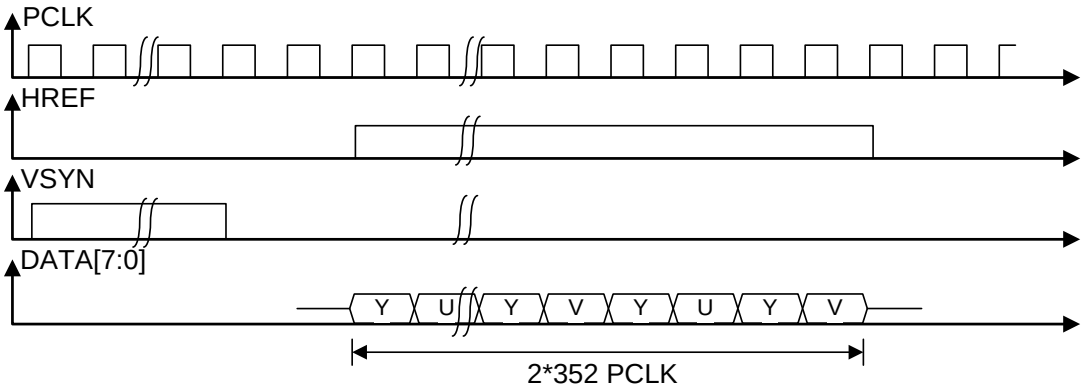


Figure 7.3: Camera Signal Timing

into the image stream format. The image stream format requires the creation of the signals data_valid, end_of_line and end_of_frame which can be generated directly out of the camera signals. Additionally the data format given by the cameras with 8bit 4:2:2 YUV has to be aligned into 16bit words in order to use the bit width of the SRAM. Because of this alignment of the incoming YUV data the camera speed of 18M is halved to 9 Mword/s. Without any counters and therefore with a small logic footprint the camera signals as given in figure 7.3 are transformed into the image stream format (see chapter 6.1) by tricky combinations. The image stream is written to the SRAM via the given SRAM bus. For further details about the bus and the SRAM controller see [10]. This

camera interface has been tested in combination with a huge FIFO to validate the functionality. Some problems occurred during testing. The fact that this only happend at high frequencies and the fact that the connection to the FPGA was heavily constraint to force the synthesize tool to build paths as short as possible indicates electrical problems on the PCB. Further investigations have to be done to get a stable camera interface.
Once the pictures from the left and right camera are stored inside the SRAM the rectification unit can start working with the maximum possible 100 MHz. The stored Y-values are now requested by the rectification unit alternating between the left and right picture. For every request of four pixels a delay of at least 6 clock cycles is produced by the SRAM controller. If the SRAM is running with 100 MHz the pixels are coming from the SRAM with a maximum rate of 2 * 20 Mword/s. The four pixels are interpolated as described in 4.3.2. The output rate of the rectification unit is consequently one fourth of the incoming data rate. This 2*5 Mword/s stream is stored into the SRAM again and is as well piped into the Stereo Harris Module. These SRAM storing accesses will happen at the same time as the rectification unit reads pixels from the SRAM. To avoid that the processes block each other a buffer FIFO should be used. This buffer collects the rectified image data and allows burst writes into the SRAM. This will avoid alternating write and read accesses which would generate a congestion. The Harris has to be clocked with 50 MHz and can only accept a pixel every third clock cycle because of the resource sharing described in 6.3. This however is enough because the data is only coming every fifth clock cycle in the 50 MHz domain. As described in chapter 5.3 the results of the Harris Corner Detector depend on the set threshold(s) and a variable k. Depending on the used Harris version a specific amount of registers is created which can be both read and written from the CPU via the so called peripheral bus [10]. For a pre-thresholding of the size 4×4 we will need 34 registers, 2*4*4 for the thresholds of both images and additional two registers to store k_l and k_r.
The feature list output at the end of the Harris is again stored into the SRAM. From there they can be read by the CPU with a maximum speed of 12.5 Mword/s with the help of a bus bridge (for more details see [10]).
Two different operating modes are possible for the design:

- If all processes described above run sequentially all processing times simply can be summed. The time to transfer the images into the SRAM equals 11.26 ms. The transfer is assumed to work in parallel because the rate of incoming data with 2*9 Mword/s is fairly small compared to the maximum SRAM rate of 100 Mword/s. The read accesses of the rectification unit will work with a maximum rate of 2*20 Mword/s what results in a time of 20.28 ms. The write accesses from the rectification unit output into the SRAM can be done at a speed of 100 Mword/s if the above mentioned buffer is introduced. The transfer of the two pictures into the SRAM will take 2.02 ms. The storing of the feature lists can be neglected because of the small amount of data. The final transfer of the two rectified images and the feature lists will take 16.22 ms at a speed of approximately 12.5 Mword/s. The resulting frame rate of the whole image processing is given with 20.01 fps. This frame rate does not include any overhead caused by arbitration times of the SRAM controller.

- If double buffering is applied, meaning that every device has two areas inside the SRAM where it can write to, it might be possible to improve the frame rate. In this case none of the units has to pause and wait until the other units are finished as in the first case. This will however create a bigger competition regarding SRAM accesses. The consequences can not be estimated with implementation details available at the time I handed in.

The Harris Corner Detector only needs a sequential image stream to output the desired features whereas the rectification unit needs random accesses inside the image. These random accesses and the fact that I did not have the possibility to store whole images inside the FPGA made it impossible to test the rectification unit in hardware. Yet, the functionality of the Harris Corner Detector has been verified with the creation of sequential data in the form of an image stream. With the infrastructure developed in [10] it was possible to interface the hardware from the CPU and test the Harris Corner Detector.

7.2 Harris Integration

The Harris Corner Detector in hardware has been evaluated with a FIFO-based approach. In order to transfer data into the FPGA a FIFO is connected to the burst bus [10]. Data and the necessary control signals of the image stream are line by line written from the CPU into this FIFO. After a specific number of lines is transfered the Harris starts outputting data line by line as well. This output is stored into another FIFO together with the image stream signals and can then be read by the CPU. The CPU reads have to happen after every written line to avoid an overflow inside the outgoing FIFO. Additionally registers, which are connected to the threshold and k inputs of the Harris, can be written by the CPU over the peripheral bus (see [10]). With this HW/SW solution it was possible to validate the functionality of the Mono and Stereo Harris.

The evaluation of the Harris Corner Detector regarding accuracy turned out to be difficult. It is not possible to work with real images because an edge or corner is never a direct change of pixel intensities and consequently the location of an edge or corner is hard to determine. Additionally different algorithms with different parameters will return different feature points and it is not possible to tell which features are the best. This is the reason why pictures were artificially created as shown in figure 7.4. The possible corner locations can now be measured by hand with subpixel accuracy and are then used as reference $\hat{x}$. The quality-measurement is based on the formula

$$\sigma_s = \frac{1}{N}\sum_{i=0}^{N-1} \|\hat{x} - x\|^2$$

which is the mean square distance between the ideal position $\hat{x}$ and the one measured. The problem with the usage of artificial pictures is the fact that the parameter values k and the threshold of the Harris Corner Detector have to be changed to values not used for

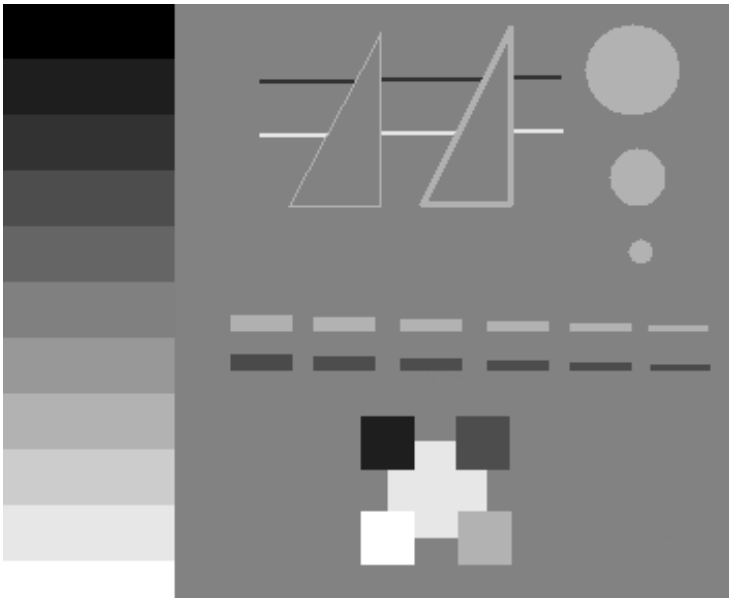


Figure 7.4: Benithaler Testpattern

real images because the edges and corners in such artificial pictures are defined sharply and the pixel intensities change immediately what is never the case in real images. The results are not conclusive but can give an idea of the algorithm's quality. By changing the parameters it was possible that at the end all features of the image were detected with maximum possible accuracy. The same results were achieved with the software solution. This was seen as a prove that the Harris Corner Detector in hardware is working accurately. The next step has been the testing of a software controlled adaptive thresholding.

As aforementioned the number of features found has to be limited to store them inside a fixed space of the SRAM. To avoid that features of the lower right part of a picture are discarded because of a lack of memory, the threshold(s) of the Harris Detector can be changed adaptively. If too many features are found the threshold(s) are increased and vice versa. The influence of the changed threshold(s) will affect the next frame and reduce/increase the number of features assuming that the image is similar to the one before. The validity of this assumption highly depends on the achieved framerate and the appearing speed of object/robot movements. Therefore no information can be given at this point how the adaptive thresholding will work in reality because the real framerate is not known yet and the robot movements are hard to estimate. A software solution has been programmed and tested which writes the threshold values to the FPGA and utilizes an I-controller to adapt the threshold(s). Depending on the settings stable feature numbers were gained after a few frames of the same picture. However, this I-control mechanism is very basic and should be improved when the system is able to handle live feeds from the cameras. The programmed HW/SW solution proved the functionality of the Harris and the hardware interface but can not give any information about runtimes because the at the end limiting component, the SRAM, is not involved in this design at all.

Assuming that the hardware part is working the next logical step after the transfer of the feature lists and the rectified images is the matching of the features in software in order to get the desired depth informations of the environment.

7.3 Software based Image Processing

7.3.1 Feature Matching

To evaluate the quality of feature matching algorithms feature lists need to be created. Hardly any differences between the results of the software and the above presented hardware solution have been found and therefore the hardware solution has not been used for further evaluations. Instead the results of the easy adaptable and platform independent Harris in C were utilized.
Two interesting aspects have to be evaluated for the feature matching: speed and accuracy. To judge about the accuracy of a match the true depth information is needed. There exist a lot of test patterns which come together with disparity maps created by universities. The probably most famous test data has been generated by the stereo vision research group of the Middlebury College [28]. These test pictures were used for the following evaluations to gain comparable results at the end.
The possibility to use OpenCV [3] on the EyeBot M6 has been evaluated at first because this vision library already comes with all algorithms we would need to match features in software. However, the resulting performance of the OpenCV algorithms on the EyeBot M6 was quite poor compared to pure C implementations. The low execution speed is caused by the fact that OpenCV calculations are mostly based on floating point accuracy. The EyeBot M6 however does not have a floating point unit and consequently the floating point operations have to be emulated. Table 7.1 gives a small overview about the measured execution times of OpenCV algorithms. The ratio of approximately 30 between the lab server named R2D2 (Intel Core2 Duo E8200, 2.66 GHz) and EyeBot M6 is still quite good for the Sobel but as soon as floating point accuracy is used by OpenCV as for the cvHarris this ratio is increased to approximately 230. This is the reason why the matching algorithms have been programmed in C and OpenCV has only been used for file I/O functionality during the evaluation process.

Function	R2D2 [fps]	EyeBot M6 [fps]	ratio
C-Sobel	1807	54.6339	33.0747
cvSobel	2292	72.5	31.6
C-Harris	190	7.3690	25.7838
cvHarris	198	0.8522	232.2300
Convolution 32F	737	3.1523	233.8005

Table 7.1: Performance of OpenCV in Comparison to C Implementations

Before the programmed algorithms can be evaluated in regards of speed and accuracy

metrics need to be defined which let us judge about the quality. Two measures, adapted from [28], where used based on the given groundtruth data:

- The Root-Mean-Squared (RMS) Error between the computed disparity map d_c and the groundtruth map d_g

$$R = \left(\frac{1}{N} \sum_{(x,y)} |d_c(x,y) - d_g(x,y)|^2 \right)^{\frac{1}{2}},$$

  with N as the number of total pixels.

- The Percentage of bad matching pixels

$$\frac{1}{N} \sum_{(x,y)} (|d_c(x,y) - d_g(x,y)| > \sigma_d),$$

  where σ_d is the disparity in pixels the computed disparity d_c has to differ from the given groundtruth disparity d_g to be counted as a bad pixel. The value σ_d was set to one for all following evaluations.

An evaluation environment has been programmed in C++ in combination with OpenCV to gather the necessary data. This environment embeds the matching algorithms programmend in C. The first function evaluated is the matching cost function.

7.3.2 Matching Cost Function

The easiest way to judge whether features are similar is the evaluation of the matching cost function. Pixel intensities i and j within a window centered at the feature point locations in both images are correlated. Depending on the result of the function the decision is made if a matching pair has been found or not. There are a lot of matching cost functions but the probably most popular three are

- Sum of Absolute Differences (SAD)

$$SAD = \sum_{i,j \in W} |i - j|,$$

  with a window W centered at the feature locations (x_l, y_l) and (x_r, y_r).

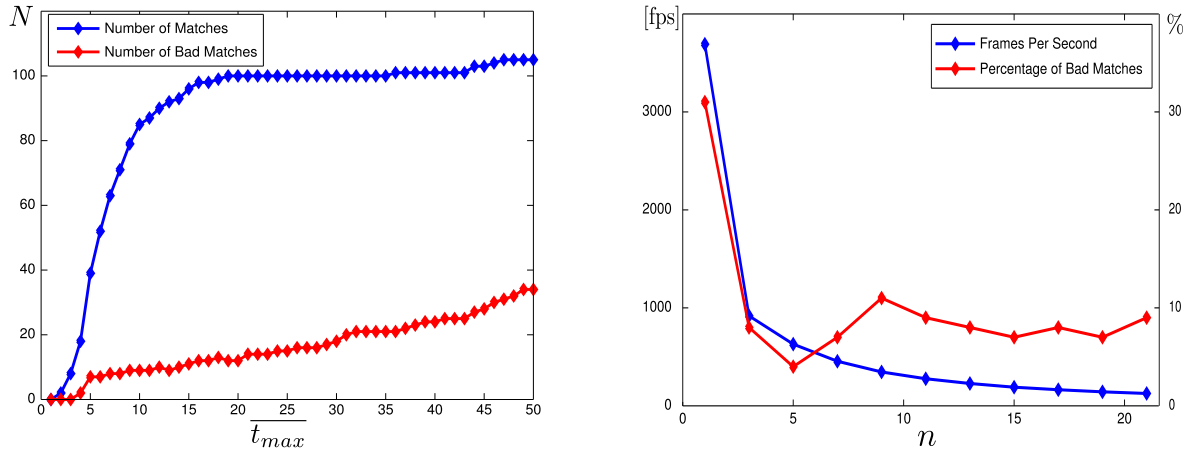
- Sum of Squared Differences (SSD)

$$SSD = \sum_{i,j \in W} (i - j)^2$$

- Normalized Cross Correlation (NCC)

$$NCC = \frac{1}{n-1} \sum_{i,j \in W} \frac{(i - \bar{i})(j - \bar{j})}{\sigma_i \sigma_j},$$

with $\bar{i}$ and $\bar{j}$ as the meanvalue of pixel intensities inside the windows. σ_i and σ_j as the standard deviations.



(a) Variation of the Threshold

(b) Variation of the Window Size

Figure 7.5: Evaluation of correlation Parameters

SAD and SSD have to be minimized whereas NCC needs to be maximized. The work of [28] has shown that there is only a little difference between SAD and SSD regarding accuracy of the results but SSD has a higher computational effort. Furthermore the much higher complexity of a Normalized Cross Correlation does not justify the slightly better results. In [15] the usage of a simple SAD is suggested for real-time systems due to low calculation times and a comparable good performance. Based on the results presented in both papers SAD was chosen as cost function for this thesis.

7.3.3 Correlation Parameters

The results of the feature correlation with a SAD cost function depend a lot on the chosen window size and the threshold. First of all a good threshold has to be found to guarantee that only features with a high similarity are matched but at the same time corresponding features are not discarded due to e.g. a slight variance in brightness. The threshold t_{max} has to be chosen in dependence on the window size like $t_{max} = window\_width * window\_height * \overline{t_{max}}$, with a constant factor $\overline{t_{max}}$. Figure 7.5(a) plots the behavior of the quality of matches depending on the value $\overline{t_{max}}$. The higher the constant $\overline{t_{max}}$ the more features are found but at the same time the number of bad matches is increased as well. The best ratio between the total number of matches and the number of bad matches has been achieved with a $\overline{t_{max}}$ in the range of $[9, 13]$ for the test patterns. This range varies only slightly from image to image and therefore a fixed $\overline{t_{max}}$ with a value of 11 was chosen for the evaluation of the window size.
The size of the window highly influences the calculation time. Figure 7.5(b) shows how

the number of bad pixels can be decreased with a bigger window size and how the frame rate is affected by the raising computational effort. The bad pixel count is only decreased to a specific point. Due to the fact that we have chosen corners as features, many of these corners will lie on edges of objects and therefore at places where disparity discontinuities will appear. This also includes that the background of the object is more likely to be different between the left and right image. If we now increase our window more and more of neighbor pixels will be considered inside the SAD cost function. These neighbor pixels can be either part of the object or of the background. If the corner is directly at the edge of an object the ratio between background and foreground pixels will be nearly the same. Therefore the decision can be easily falsified because back and foreground compensate each other and a small noise can consequently have a big influence. Most of the times it happens that a true match is discarded because of different backgrounds. This is the reason why the enlargement of the window improves the matching quality only to a specific point. The speed however is increased continuously and therefore a window size of 7 or 9 seems to be appropriate for the EyeBot M6. These results are only based on the test patterns. Further evaluation is necessary when the EyeBot is operated in its real environment.

7.3.4 Splitted Correlation

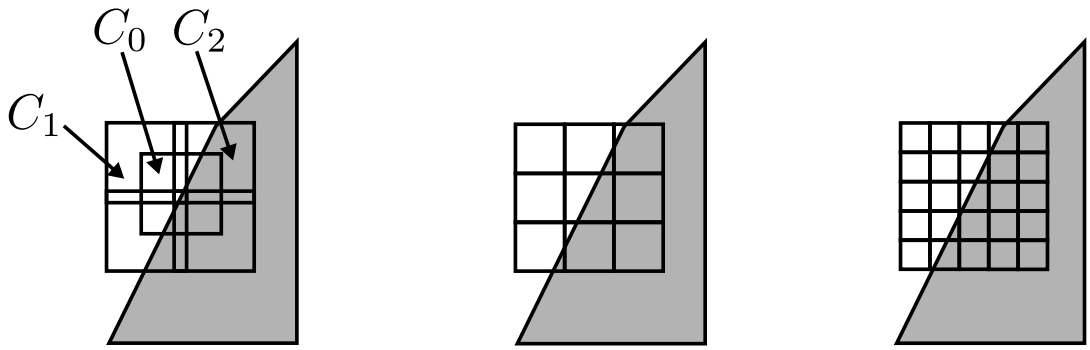


Figure 7.6: Splitted Correlation Windows [15]

A different approach for dealing with the described problem has been developed by Hirschmüller [15]. This approach splits the correlation window into smaller parts as shown in figure 7.6. For every small window the SAD is calculated between left and right image. The resulting correlation values are named as shown in the figure with C_0, C_1, etc. Hirschüller now suggests to take only the best correlation values in consideration. The resulting total correlation value for the five window approach in figure 7.6 will therefore result in

$$C = C_0 + C_2 + C_4.$$

The center value C_0 always has to be part of the summation. This selective correlation helps to use only the windows which are really part of the object but not of the back-

ground which is likely to be different. The solution has been programmed and evaluated in C. The best results have been achieved with a window size of 5×7 and a constellation of eight windows surrounding a center window. The number of good matches has been increased by 19.3 % in average while the number of bad matches stayed the same. However, the algorithm consumes more time than the single window approach of section 7.3.3. The resulting correlation values have to be sorted and the best ones have to be accumulated. The sorting is done with the help of the quicksort algorithm. A more computational intense task than the sorting is the additional area which has to be correlated. The nine window constellation achieved 31 fps in average on the EyeBot M6. This frame rate can vary a lot between the test pictures because it depends on how many features are found in the same line and have to be correlated with each other. Precise time values are not given in this case because they highly depend on the final settings which again have to be optimized for the environment the EyeBot M6 will be operated in.

7.3.5 Dense Disparity Mapping

In comparison to the solution which has been implemented for this thesis a dense disparity algorithm tries to gain the depth information for all points inside the image. The stereo vision research group of the Middlebury College [28] evaluated algorithms which produce dense disparity maps. They offer C++ code for testing and evaluation purposes of the different algorithms. This C++ code has been compiled for the EyeBot M6, too. The correlation with SAD and a 9×9 window takes 5.1 s for the famous tsubuka picture. With optimized programming it might be possible to achieve more than one frame per second on the EyeBot M6 but the calculation of dense disparity maps in real-time seems to be hardly possible. Additionally the accuracy of the resulting dense disparity maps in general have higher bad pixel counts than the above presented algorithms. The pure matching solution achieved in comparison depending on the complexity 30 to 600 frames per second. The given rates are, without image transfers and any other control tasks, later on needed on the EyeBot. The pre-processing which has the highest computational effort, the edge detection and rectification, has been implemented in hardware and works independently from the CPU, whereas the dense disparity mapping hardly can be implemented on the given FPGA. Therefore the calculation of a sparse disparity map can be seen as the best solution for the EyeBot M6.

8 Conclusion

This thesis has shown that even a small FPGA as the Spartan 3E 500 has sufficient resources for general image processing applications. The basic step which has to be done before any stereo vision processing makes sense is the rectification. The stereo rectification and image undistortion have been combined to one mapping process. This mapping is approximated by 2D polynomials which have to be evaluated inside the FPGA. The appropriate polynomials are regressed fully automated in MATLAB for every kind of camera setting. As soon as the stereo camera calibration as described in [6] has been performed, the resulting calibration file has just to be loaded into the developed Rectification Toolbox and a optimized VHDL package will be generated which only has to be included into the project. The synthesize tool will create a rectification unit according to the information provided in this package. The user does not have to do any VHDL coding in order to gain a fully functional and optimized rectification unit. The rectification unit allocates only 5 % of the FPGA.
To keep the high adaptability provided by the rectification unit a interface for the following image processing modules has been developed. This image stream format ensures a high reusability of the programmed modules because they can be easily interconnected with each other.
In order to gain the stereo information out of the given stereo rig a Harris Corner Detector has been implemented to reduce the search space from 1D given by the rectification to approximately 500 points. This solution has been chosen because the calculation of a dense disparity map does not fit into our FPGA and the software solution is way too slow. The implemented Mono Harris is capable of processing images at the full camera speed of 60 fps. However, two of the Mono Harris Detectors did not fit inside the FPGA. Therefore two Mono Harris have been merged to a Stereo Harris to share resources whereever possible. The optimized Stereo Harris operates at reduced rates which are adapted to the maximum speed given by the rectification. A optimization made it possible that the Harris Corner Detection can be performed for the stereo constellation without running out of block RAM and multipliers inside the FPGA.
It was not possible to present a complete system in the previous chapter. The described rectification unit requires the images to be stored in the SRAM. However, SRAM accesses were not possible at the time this thesis has been handed in. With a working SRAM and the controller which has already been planned and presented in [10] the developed hardware units can be combined as described in chapter 7. The software which is necessary to find matching features has already been implemented.

9 Future Work

The first problem which has to be solved is the interface between FPGA and SRAM. Once the communication works reliable an SRAM Controller has to be programmed [10]. For the described units priorities have to be allocated and a global control unit has to be developed. The easiest way will be to ignore the cameras at the beginning and write test patterns directly from the CPU into the SRAM. It is recommended to first start with a sequential flow inside the FPGA as described in chapter 7. Once the first rectified images and feature lists have been successfully transfered back to the CPU it should be investigated in which way the proposed doublebuffering can improve the performance of the system. The doublebuffering will result in a higher competition on the bus and therefore will increase the arbitration times. At the moment it can hardly be estimated how this increase behaves compared to the time gained by partially parallel running processes.
Another unsolved task is the instability of the camera interface. All investigations which have been done indicate a electrical problem on the PCB. These problems have to be solved with the new revision of the EyeBot M6. For further details see [10]. Once the cameras work reliable they can be integrated in the design with the help of the developed camera to image stream converter. The resulting image stream in the clock domain of the FPGA then has to be written into the SRAM. As soon as two images are available the rectification unit can be started.
When the hardware based image processing part is ready a camera calibration has to be performed and a rectification unit for given stereo rig has to be synthesized. The resulting rectified image streams in combination with the feature lists have then to be used for the optimization of the available parameters. The thresholds for the Harris Detector are in the current software controlled with a simple I-controller and can therefore be improved to achieve faster adaptions regarding the feature distributions over the image. In addition the parameters of the matching software have to be evaluated to find a good setting which guarantees high accuracy and a small computational effort.
With the gained 3D information of the environment basic control tasks like obstacle avoidance can be programmed for the robot. The image processing takes only a small amount of the CPU time because of the hardware acceleration so more complex algorithms can be taken in consideration as well. The Simultaneous Localization And Mapping (SLAM) algorithm is only one example. This algorithm would allow the EyeBot to create full maps of his environment out of the gained feature locations in space [22]. The presented stereo vision system gives a huge variety of possibilities for further research and experiments.

A Rectification Toolbox

The Rectification Toolbox includes all scripts programmed for this thesis in order to automatically generate quantizer settings out of a stereo calibration. The following step by step instruction explains how to create such a calibration and gain an optimized rectification unit.
First of all the cameras have to be calibrated. To take the appropriate pictures a bit stream and a program called cheese can be found in the source directory. Program the FPGA of the EyeBot M6.12 with the bitstream and run the cheese program. For the calibration process a chess pattern is now presented both cameras in different angles. Such a pattern can be found on the homepage Camera Calibration Toolbox [6]. Take now at least 20 pictures of the chessboard by pushing the cheese button. Take care that the chessboard can be seen with both cameras. By pushing the button two pictures are taken at ones and the actual picture count shown on the LCD is incremented. After you have enough pictures, just copy the pictures which are stored at the same location from where you ran the cheese program to your PC.
The next step is the calibration of the cameras, first each on their own and afterwards a stereo calibration. This is done in MATLAB with the help of the Camera Calibration Toolbox [6]. Very good tutorials can be found on the homepage thus there is no further need to explain it in more detail here. Just follow the instructions of the Stereo Calibration Tutorial.
Once you created the outputfile `stereo_calib_cam.mat` you can run the automated floating to fixed point conversion. The main file is the `run_all` script. Inside here all necessary configurations are done. First of all you have to specify the path to you calibration file, the desired maximum error `EM_MAX`, and L and H for the coarse_quantization as described in chapter 2. Additionally specify the number of points the algorithm should run with. This number highly influences the runtime of the optimization thus chose it carefully. The script then calls the function `calc_regression_polynomials()` which returns the coefficients for the left and the right side and additionally the perfect mappings for both sides. With the help of the returned coefficients of the four polynomials the function `calc_best_quantizers()` is called for each of the polynomials which starts the optimization process for the specified coordinate X or Y and side left or right. As a result you get four vectors with the settings of 26 quantizers each. These quantizer settings are now merged into one list by calling the function `merge_quantizers()`. After they have been merged a VHDL package is created with the `create_stereo_cam_package()` function. This file has finally to be included into the coordinate warper vhd-file to synthesize a rectification unit for the current camera settings. Table A.1 gives a short overview about all MATLAB files included by the Rectification Toolbox.

file/function name	description
`run_all`	runs the whole process to determine optimized quantizer settings; the parameters are specified inside the script
`calc_best_quantizers`	implements all functions described in chapter 2: the range estimation, coarse optimization and the finetuning; returns list of 26 quantizers for the specified coordinate
`calc_error`	returns the pixel error between two mappings; used to compare floating with fixed point solution; to change the error measure edit this file
`calc_fixpoint`	calculates the fixed point solution for the points and quantizers given as parameter value; returns the resulting mapping
`calc_floating`	calculates the mapping based on floating point accuracy; returns the resulting mapping with fixed point accuracy
`calc_ranges`	returns mins and maxs for all 26 quantizers based on floating point position; this function is used by the range estimation function
`calc_regression_polynomial`	calculates the LUTs out of the stereo calibration file given as parameter value; this LUTs are then regressed with 2D polynomials and the resulting coefficients are returned together with the LUTs
`compare_mappings`	useful to visualize the mapping errors; just give the function two mappings and the image size as parameter values and it calculates mean square error, residuals and the variance of the error; additionally a 3D plot is created which visualizes the error
`compare_mappings_2D`	same as compare_mappings but visualization is done in 2D with quivers as used for this thesis
`compare_vhdl_with_fixpoint`	useful script to validate the VHDL-module; the csv-files created by the testbench rectification_unit_tb.vhd are loaded and compared with quantizers which have to be specified inside the script
`create_c_lut`	creates four header-files with the LUTs used in the written c-version of the rectification
`create_stereo_cam_package`	creates the VHDL package which has to be finally included into the coordinate warper
`merge_quantizers`	merges quantizer lists given as parameter values to one single quantizer list
`imwarp`	rectifies an image with the specified method formula or map; formula means that the polynomials are evaluated inside the function and the resulting rectified image is returned together with the LUT; map rectifies an image based on the handed over LUT

Table A.1: Functions of the Rectification Toolbox

Quantizer	X left	Y left	X right	Y right	merged
q1	ufix[11,7]	ufix[7,6]	ufix[9,6]	ufix[6,7]	ufix[11,7]
q2	ufix[14,14]	sfix[13,15]	ufix[12,12]	sfix[14,15]	sfix[16,15]
q3	sfix[14,15]	ufix[14,14]	sfix[13,15]	ufix[13,13]	sfix[16,15]
q4	ufix[9,18]	ufix[11,22]	ufix[13,22]	ufix[10,21]	ufix[13,22]
q5	ufix[11,22]	ufix[12,22]	ufix[11,21]	ufix[9,18]	ufix[13,22]
q6	ufix[12,22]	ufix[12,21]	ufix[12,22]	ufix[11,20]	ufix[13,22]
q7	sfix[13,30]	sfix[6,27]	sfix[13,30]	ufix[5,30]	sfix[14,30]
q8	ufix[7,31]	sfix[13,30]	sfix[5,28]	sfix[11,28]	sfix[15,31]
q9	sfix[1,25]	sfix[13,30]	sfix[6,28]	sfix[13,30]	sfix[13,30]
q10	sfix[13,30]	sfix[9,30]	sfix[12,29]	sfix[6,30]	sfix[13,30]
q11	ufix[5,21]	sfix[13,22]	sfix[7,21]	sfix[12,21]	sfix[14,22]
q12	sfix[13,22]	sfix[9,22]	sfix[12,21]	sfix[6,22]	sfix[13,22]
q13	sfix[12,21]	sfix[9,22]	sfix[13,22]	ufix[5,22]	sfix[14,22]
q14	sfix[6,21]	sfix[13,22]	sfix[8,22]	sfix[13,22]	sfix[13,22]
q15	sfix[12,14]	ufix[11,14]	sfix[12,14]	ufix[11,14]	sfix[13,14]
q16	ufix[12,14]	ufix[13,14]	ufix[13,15]	ufix[13,14]	ufix[14,15]
q17	ufix[12,14]	sfix[13,15]	ufix[12,14]	sfix[14,15]	sfix[15,15]
q18	sfix[12,7]	ufix[14,5]	sfix[11,6]	ufix[14,5]	sfix[17,7]
q19	ufix[14,5]	sfix[13,6]	ufix[14,5]	sfix[15,7]	sfix[17,7]
q20	sfix[11,21]	ufix[11,21]	sfix[11,21]	ufix[13,22]	sfix[14,22]
q21	ufix[12,21]	sfix[12,22]	ufix[13,22]	sfix[12,22]	sfix[14,22]
q22	sfix[13,15]	ufix[14,13]	sfix[12,14]	ufix[14,13]	sfix[17,15]
q23	ufix[13,13]	sfix[13,14]	ufix[14,14]	sfix[14,14]	sfix[15,14]
q24	ufix[14,5]	sfix[15,5]	ufix[14,5]	sfix[15,5]	sfix[16,5]
q25	ufix[9,0]	ufix[9,0]	ufix[9,0]	ufix[8,-1]	ufix[9,0]
q26	ufix[7,-2]	ufix[9,0]	ufix[9,0]	ufix[9,0]	ufix[9,0]

Table A.2: Quantizer Settings produced with Rectification Toolbox

Bibliography

[1] *MATLAB*. http://www.mathworks.com/. 6, 13

[2] *ModelSim*. http://www.model.com/. 6

[3] *Open Computer Vision Library*. http://sourceforge.net/projects/opencvlibrary/. 11, 74

[4] *AccelDSP Synthesis Tool*, 2006. http://www.xilinx.com/ise/dsp_design_prod/acceldsp/index.htm. 12

[5] A. Verri, V. Torre: *Absolute depth estimate in stereopsis*. Vol. 5:426–431, 1985. 20

[6] Bouguet, Jean-Yves: *Camera Calibration Toolbox for Matlab*. http://www.vision.caltech.edu/bouguetj/calib_doc/. 26, 79, 83

[7] Coors, Martin, Holger Keding, Olaf Lüthje and Heinrich Meyr: *Design and DSP implementation of fixed-point systems*. EURASIP J. Appl. Signal Process., 2002(1):908–925, 2002. 12

[8] Cordelia Schmid, Roger Mohr, Christian Bauckhage: *Evaluation of Interest Point Detectors*. International Journal of Computer Vision, 37(2):151–172, 2000. 46

[9] Diaz, Javier: *Fine grain pipeline systems for real-time motion and stereo-vision computation*. International Journal of High Performance Systems Architecture, 1:60–68(9), 19 April 2007. 45

[10] Geier, Martin: *Design and Implementation of an FPGA-based Image Processing Framework for the EyeBot M6*. Perth, University of Western Australia, 2009. 2, 41, 44, 63, 69, 70, 71, 72, 79, 81

[11] Gumstix, inc. http://www.gumstix.com/. 2

[12] Harris, Chris and Stephens: *A Combined Corner and Edge Detector*. In *The Fourth Alvey Vision Conference*, pages 147–151, 1988. 46

[13] Hartley, R. I.: *In defense of the eight-point algorithm*. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 19(6):580–593, 1997. 25

[14] Heikkila, Janne and Olli Silven: *A Four-step Camera Calibration Procedure with Implicit Image Correction*. In *CVPR '97: Proceedings of the 1997 Conference*

on Computer Vision and Pattern Recognition (CVPR '97), page 1106, Washington, DC, USA, 1997. IEEE Computer Society. 23, 26

[15] Hirschmuller, Heiko: *Improvements in Real-Time Correlation-Based Stereo Vision*. In *IEEE Workshop on Stereo and Multi-Baseline Vision at IEEE Conference on Computer Vision and Pattern Recognition*, pages 141–148, Kauai, Hawaii, USA, 9 - 10 December 2001. viii, 76, 77

[16] Jiao, W., Y.L. Fang and G. He: *An Integrated Feature Based Method for Sub-Pixel Image Matching*. page B1: 1157 ff, 2008. 46

[17] Jorg, Stefan, Jorg Langwald and Mathias Nickl: *FPGA based Real-Time Visual Servoing*. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 1*, pages 749–753, Washington, DC, USA, 2004. IEEE Computer Society. 29, 43

[18] Keding, Holger, Markus Willems, Martin Coors and Heinrich Meyr: *FRIDGE: A Fixed-Point Design And Simulation Environment*. In *proc. Design Automation and Test in Europe*, pages 429–435, 1998. 12

[19] Kim, Seehyun, Ki il Kum and Wonyong Sung: *Fixed-point optimization utility for C and C++ based digital signal processing programs*. In *IEEE Trans. Circuits and Systems II*, pages 1455–1464, 1998. 12

[20] Knowles, J. and E. Olcayto: *Coefficient Accuracy and Digital Filter Response*. Circuit Theory, IEEE Transactions on, 15(1):31–41, Mar 1968. 11

[21] Lane, R. A. and N. A. Thacker: *Stereo vision research: An algorithmic survey*, 1996. 45

[22] Leonard, J.J. and H.F. Durrant-Whyte: *Mobile robot localization by tracking geometric beacons*. Robotics and Automation, IEEE Transactions on, 7(3):376–382, Jun 1991. 81

[23] Longuet-Higgins, H. C.: *A computer algorithm for reconstructing a scene from two projections*. Nature, 293(5828):133–135, 1981. 25

[24] Masrani, Divyang K. and W. James MacLean: *Expanding Disparity Range in an FPGA Stereo System While Keeping Resource Utilization Low*. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*, page 132, Washington, DC, USA, 2005. IEEE Computer Society. 45

[25] Moravec, Hans: *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover*. In *tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University & doctoral dissertation, Stanford University*. September 1980. 46

[26] OmniVision. `http://www.ovt.com/`. 2

[27] Roy, Sanghamitra, Debjit Sinha and Prith Banerjee: *An algorithm for trading off quantization error with hardware resources for MATLAB based FPGA*

design. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 256–256, New York, NY, USA, 2004. ACM. 13, 14, 15, 17

[28] Scharstein, Daniel and Richard Szeliski: *A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms*. Int. J. Comput. Vision, 47(1-3):7–42, 2002. vii, 47, 74, 75, 76, 78

[29] Slama, Chester C., Theurer Charles. Henriksen Soren W.: *Manual of photogrammetry*. 22

[30] T. Grötker, E. Multhaup and O.Mauss: *Evaluation of HW/SW Tradeoffs Using Behavioral Synthesis*. ICSPAT, 1996, 1996. 12

[31] Vancea, C. and S. Nedevschi: *LUT-based Image Rectification Module Implemented in FPGA*. Intelligent Computer Communication and Processing, 2007 IEEE International Conference on, pages 147–154, Sept. 2007. 29, 30, 43

[32] Xilinx: *Spartan-3E FPGA Family - Data Sheet*, 2006. `http://direct.xilinx.com/bvdocs/publications/ds312.pdf`. 2