Design of an FPGA-Based Smart Camera and its Application Towards Object Tracking

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Engineering

in

Electronics and Computer Engineering

at Massey University, Manawatu, New Zealand

Miguel Contreras

2016

Abstract

Smart cameras and hardware image processing are not new concepts, yet despite the fact both have existed several decades, not much literature has been presented on the design and development process of hardware based smart cameras. This thesis will examine and demonstrate the principles needed to develop a smart camera on hardware, based on the experiences from developing an FPGA-based smart camera. The smart camera is applied on a Terasic DE0 FPGA development board, using Terasic's 5 megapixel GPIO camera. The algorithm operates at 120 frames per second at a resolution of 640x480 by utilising a modular streaming approach. Two case studies will be explored in order to demonstrate the development techniques established in this thesis.

The first case study will develop the global vision system for a robot soccer implementation. The algorithm will identify and calculate the positions and orientations of each robot and the ball. Like many robot soccer implementations each robot has colour patches on top to identify each robot and aid finding its orientation. The ball is comprised of a single solid colour that is completely distinct from the colour patches. Due to the presence of uneven light levels a YUV-like colour space labelled YC_1C_2 is used in order to make the colour values more light invariant. The colours are then classified using a connected components algorithm to segment the colour patches. The shapes of the classified patches are then used to identify the individual robots, and a CORDIC function is used to calculate the orientation.

The second case study will investigate an improved colour segmentation design. A new HSY colour space is developed by remapping the Cartesian coordinate system from the YC_1C_2 to a polar coordinate system. This provides improved colour segmentation results by allowing for variations in colour value caused by uneven light patterns and changing light levels.

Acknowledgments

I would like to thank my supervisors, Donald Bailey and Gourab Sen Gupta, for all their support throughout this thesis. I am grateful to Donald, for being the driving force that always pushed me to go the extra mile and achieve more. I am thankful to Sen for always being a friendly person to talk to and for his helpful advice.

I would like to thank my mum Lindy Lockett, my dad Raul Contreras, my stepdad Maurice Sackett, and my brother Ben Contreras, for their unconditional love and support throughout this thesis. As teachers, Mum and Dad always put an emphasis on education and our happiness, and worked tirelessly to ensure we had both. I am eternally grateful for my Mum, Dad, stepdad, and brother, for keeping me grounded through the good times and focused through the rest. Thank you Dad and Pop (my grandfather, Alan Lockett) for being the beacons that guided me towards, and inspired, my love of engineering.

I would also like to thank all the friends and neighbours who have supported me from the start with their time, advice, proof reading and listening ears. Special thanks to Sam Corpe, for his support and company through the many long days and late nights at Massey.

Publications

1) D. Bailey, G. Sen Gupta, and M. Contreras, "Intelligent Camera for Object Identification and Tracking," in *Robot Intelligence Technology and Applications 2012*. vol. 208, ed: Springer International Publishing, 2012, pp. 1003-1013.
2) M. Contreras, D. Bailey, and G. Sen Gupta, "FPGA Implementation of Global Vision for Robot Soccer as a Smart Camera," in *Robot Intelligence Technology and Applications 2013*. vol. 274, ed: Springer International Publishing, 2013, pp. 657-665.
3) M. Contreras, D. Bailey, and G. Sen Gupta, "Techniques for Designing an FPGA-Based Intelligent Camera for Robots," in *Robot Intelligence Technology and Applications 2014*. vol. 345, ed: Springer International Publishing, 2014, pp. 633-646.
4) M. Contreras, D. Bailey, and G. Sen Gupta, "Robot Identification using Shape Features on an FPGA-Based Smart Camera," in *29th International Conference on Image and Vision Computing New Zealand (IVCNZ '14)*, Hamilton, New Zealand, 2014, pp. 282-287.
5) D. Bailey, M. Contreras, and G. Sen Gupta, "Towards automatic colour segmentation for robot soccer," in *6th International Conference on Automation, Robotics and Applications (ICARA '15)*, Queenstown, New Zealand, 2015, pp. 478-483.
6) D. Bailey, M. Contreras, and G. Sen Gupta, "Bayer interpolation with skip mode," in *Irish Machine Vision and Image Processing (IMVIP '15)*, Dublin, Ireland, 2015, pp. 67-74.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1. Introduction

Processing speed is critical in many vision applications. This is especially true when vision is used as a sensor in a control system, as is frequently encountered in robotics, machine vision and surveillance. The processing speed (latency) limits the usefulness of the data derived by image processing, because increased delays within a feedback loop can severely impact the controllability of a system. In many cases, long latencies can cause poor performance and in some cases system instability.

The complexities of many vision algorithms often require higher performance computing to manage the high data rates from cameras. This usually entails a processor running with high clock speed just to maintain the high pixel throughput. As a result, this usually involves high power consumption, high temperatures, and sometimes expensive equipment. Furthermore, the increased processing load incurred by the image processing reduces the time available for other processing tasks, such as control functions or strategy. In some cases, a high clocked processor can be used to increase processing power; however, this generally compounds the problems of power, heat, and expense. Another solution that has been explored is utilising parallelism [1]. By breaking the processing into smaller problems, and distributing them among multiple separate processors, the processing can often be completed much faster than conventional image processing. This would allow multiple low powered processors to work together to produce a comparable result to a single high powered processor.

Many form factors and architectures have been explored for parallel image processing, such as RISC processors [2, 3], graphics processing units (GPUs) [4], and digital signal processors (DSPs) [5, 6]. Because many of the options can be implemented on a single chip, parallelism can be utilised in a relatively small form factor. These solutions often solve many of the problems (size, weight, power consumption, heat generation) that are generally found in many robotics and machine vision applications. This is especially beneficial for mobile robotic projects as system size and weight are often limited and the system is usually battery operated.

However, even with the advantages of parallel processors, they do not necessarily fix the processing overhead problem encountered when deriving the control functions. In order to overcome this barrier, processing is increasingly being moved to within the camera. Consequently, the last decade or so has seen the advent of so-called “smart cameras”, where the camera is no longer solely responsible for capturing the images, but performing some of the processing, and communication of the processed images or results [5]. Following this reasoning a smart camera could be developed using the same parallel hardware systems described. This would provide the same advantages of fast and robust processing, in a relatively small form factor, and allow a separate processor sufficient time for control function or strategy processing.

1.2. What is a Smart Camera?

Put simply, a smart camera is a self-contained vision system that applies image processing procedures on video streams or single images in order to extract application-specific data. From this point the smart camera can then communicate the results for further processing or make decisions and regulate a control system. Therefore, as a generalisation, smart cameras capture images or video streams and output processed data. Often smart cameras are implemented using low power embedded systems, thus creating a physically small and low powered device. By using embedded elements, the system is more robust as it generally doesn't need to rely on an operating system, like conventional computers.

In recent years, as embedded technology has become more advanced, more and more processing is being moved into the camera. This could be as simple as applying a filter to an image, or as complex as connecting to the internet and uploading the images online. At this point it is important to further define the term "smart camera" as it applies in this thesis. The term does not define a device that is only capable of fixed tasks such as image modification or video compression. A smart camera must also be capable of extracting data from the images which can then be used for further analysis or used to make control decisions.

As cameras increase in resolution and frame rate, the image quality and accuracy for dynamic scenes also increases. However, this also increases the computational overhead required for processing. At high resolutions or frame rates, a general-purpose processor struggles to keep up with the processing demands of image processing applications. However, there are many embedded devices that are better suited to the large computational loads of image processing. Digital signal processors are commonly used for implementing smart cameras [5-7]. The architecture of DSPs has been optimized for signal processing in a number of ways. The use of a Harvard architecture doubles the memory bandwidth by separating instruction and data memories. The CPU also has single cycle multiply and accumulate functions which speeds up filtering and other related operations.

Low-level vision processing operates independently on individual pixels, enabling the associated parallelism to be readily exploited [8]. This has led to parallel processors which work simultaneously on multiple parts of an image. One example of this is described in [9]. They developed a dedicated chip that operates a bank of 320 processors in parallel on the pixels within one or more rows using a SIMD (single instruction multiple data) architecture. The input has a serial to parallel conversion to convert the incoming pixel stream to make the image rows available to the processor in parallel. A similar unit transforms the parallel processed data back to a serial stream for output.

Despite the advantages of a hardware-based smart camera there are some disadvantages. Designing complex image processing algorithms for embedded systems can be rather complicated and time-consuming. Furthermore, parallel processing increases the difficulty of development by requiring the temporal component of the algorithm to be explicitly defined, whereas serial processing handles this implicitly.

Embedded systems generally suffer from limited memory, especially for storing operating code. This usually means that each smart camera needs to be tailored to complete a specific role. While this increases the robustness for performing its intended task, this tends to limit the adaptability of the smart camera. Even though it is possible to reprogram a smart camera to fulfil a different role, this requires a new algorithm to be developed, which takes more development time. These problems can sometimes outweigh the benefits of hardware-based smart cameras in favour of other software-based solutions.

1.3. Why use an FPGA?

A Field-Programmable Gate Array (FPGA) is a type of integrated circuit (IC) comprised of hundreds of reconfigurable internal logic gates. Unlike the rigid designs of many integrated circuits, FPGAs are highly configurable, allowing their internal logic to be completely rewritten. This results in the FPGA acting like a virtual circuit board, effectively transforming the programmers coding into virtual wire circuits. This allows a single FPGA to take on the function of many different IC simply by uploading a new program. Another advantage FPGAs have over ICs like microcontrollers and processers is their parallel operation. An FPGA is capable of processing vast amounts of data in parallel, and across multiple clock domains.

Parallel processing has widely been used as a way of increasing overall processing power by performing multiple operations simultaneously. This can be done by employing multiple processors, such as computer clusters or RISC processors, or within a single processor using parallel architectures such as SIMD. Implementing parallel processing using non-parallel hardware such as microcontrollers and multiple processors can provide higher processing speeds compared to serial approaches.

Unlike other parallel processors that utilise limited parallel capabilities, the FPGA offers a truly scalable and customizable environment. FPGAs are not limited by internal hardware structures, like CPUs and microcontrollers. For example, a RISC processor or CPU is limited by the number of cores that can split up the work. An FPGA however, can create as many cores as needed, and is only limited by the amount of on-chip logic resources. This allows the FPGA to adapt to the requirements of the design, rather than the design changing to suit the capabilities and limitations of the hardware. In practice, this allows the FPGA to be far more flexible than other processors. In theory, this allows for a greater number of operations to be performed at once resulting in faster processing, and for video processing, at high frame rates.

The FPGA is well suited for performing repetitive processes over a long period of time. This makes it ideal for image processing tasks, as image processing often consists of performing repetitive tasks on high volumes of data. Due to the parallel environment within the FPGA, more data is able to be processed every clock cycle. This can potentially lead to multiple filters or processes to be processed simultaneously.

The idea of using FPGAs as a platform for a smart camera is not a new one. In recent years, FPGA technology has matured to the stage where it is practical for image processing tasks. FPGAs have been used in a wide range of ways within smart cameras. One has been through

the use of a SIMD architecture to exploit spatial parallelism [10]. This is achieved using a set of identical processing elements (PEs) in parallel. Another common architecture for image processing is the window processor [11]. Here a series of dedicated PEs are used to perform identical operations to pixels within a window, which are then reduced by another PE to give a single window output. This kind of architecture exploits functional parallelism [12], where the same function is applied to all of the pixels within a window. This effectively implements a collection of general-purpose processors operating in parallel within the FPGA. At the other end of the spectrum, dedicated logic can be developed for the particular application, and implemented on the FPGA. Leeser et al. [13] demonstrate two cases where an FPGA is used as a smart camera to accelerate processing. Each case was developed to implement optimised image processing techniques in order to process live video feeds. In each case the FPGA was found to be roughly 20 times faster than a conventional 1.5 GHz computer and allowed real-time and low-latency processing of video feeds when previously this was not possible. Dedicated logic approaches have the potential to use fewer resources, compared to more general-purpose processor approaches. Though, they tend to be more specialised and designed for specific operations. However, blocks of logic, such as memory management, camera interfacing, and other basic operations can be reused from one design to the next.

With parallel computing, as with general computing, there are commonly two approaches, using a generalised architecture capable of completing many different functions, or developing dedicated logic for individual functions. General purpose architecture tends to use more resources and is commonly more complex to design, due to its all-purpose nature. Alternatively, while dedicated processes usually take relatively fewer resources in comparison, they generally require greater development time as each process must be developed separately. Furthermore, dedicated processes offer less flexibility to design changes because of their customised nature.

1.4. Issues with Smart Camera Design

There has been a lot of research outlining the design and use of smart cameras. However, there is very little literature on design techniques for the development of smart cameras. Furthermore, while some literature is starting to emerge documenting image processing techniques for FPGAs, there is still no specific literature on developing smart cameras. This thesis aims to fill the gap where the knowledge and techniques needed to create smart cameras are not well documented. The goal is to reduce the learning curve for smart camera design, making them more practical for future projects.

Despite all of the advantages that a smart camera can offer to projects requiring robust image processing applications, they are not commonly employed. This is due to greater complexity and longer development time, especially for an FPGA-based approach, compared to more conventional software methods. This thesis will look at a few of the more common issues that can hinder the development process. Tested techniques will then be outlined and demonstrated through the use of case studies, to show practical examples of the techniques in use.

There are four major issues that can hinder FPGA-based smart camera development:

- Design of parallel systems is complex
- Timing constraints must be explicitly handled due to parallelism
- Limited hardware debugging resources at run-time
- Long compile times for hardware algorithms

Designing a hardware algorithm is not straightforward. Unlike software algorithms which follow a serial command structure, FPGAs allow for parallel algorithms to be executed. Parallel algorithms allow for faster processing, as FPGA functions can be calculated simultaneously. Because of this, the design can become complex, especially as the algorithm becomes more intricate.

Timing in particular becomes an issue, as different operations have different latencies. This is not as much of an issue in software as timing is implicit in the sequence of commands. However, hardware must handle the timing explicitly to ensure information is processed correctly. In many cases, errors can occur when implementing the correct logic but at the wrong instance.

Debugging is not as simple for hardware algorithm development as it is for software algorithms. Although compilers check for syntax errors, finding logic and timing errors is much more difficult in hardware. There are two main difficulties here. First, in software, only one thing happens at a time. So, the changes resulting from a single clock cycle are small. In hardware, many things happen simultaneously. A lot can change in a single clock cycle, making it significantly more difficult to identify errors and track down their root causes. Many development environments provide a software simulation package meant to emulate the functionality of the FPGAs resources. These provide a systematic overview of the functionality as the parallel image processing algorithm is executed. Even so, finding the cause of unwanted or unanticipated changes in data can be time-consuming. Second, when interfacing the algorithm with real-time hardware, it is necessary for the algorithm to process data at real-time rates. For simulation, it is also necessary to simulate the operation of hardware external to the FPGA. Therefore, additional simulation models must be developed, debugged, and tested before even simulating the algorithm. For a smart camera, the simulation model for the camera will need to stream pixel data from a file. Any errors in this model (for example timing errors) can invalidate perfectly working algorithms. Due to the complexity of most image processing algorithms, simulation can be a very laborious and time-consuming method of debugging. Furthermore, once the algorithm is encoded onto the FPGA there are limited mechanisms for debugging errors.

Code compilation is an inevitable part of any programming and image processing project. Generally, the more complex the algorithm is, the longer it takes to compile. This is true for both software and hardware compilers. However, it takes longer to compile for parallel approaches, as more optimisation and timing passes must be processed. FPGAs have an added ‘place and route’ phase where logic is mapped to specific resource. The very large number of possible mappings makes the process poorly defined and difficult to optimise, especially when

approaching the capacity of the FPGA. This drastically increases the compilation time, and therefore the development time. This is especially true when parts of the algorithm are dependent on the surrounding physical environment, and must be adjusted frequently.

During the development of the smart camera for the case studies, many techniques were explored in an attempt to minimise these issues. This thesis will outline some of the more important and successful principles.

- It is faster to test theories in a software environment than in hardware
- A modular algorithm design can make programming and testing simpler
- It is important to clearly define the interfaces between modules
- Built-in functions in an image processing suite (such as Matlab) can both help and slow down algorithm development
- Some form of communication with the smart camera is critical
- A parameterised hardware design can help make it easier and faster to make changes
- On development boards, there can be some resources available to help debug during run-time

Designing an image processing algorithm in hardware is very difficult. The development tools are not designed to easily show the effects of different filters or techniques compared to a software development environment, such as Matlab. Even if the filter is relatively trivial, it is better to first test its behaviour in a software environment, to investigate how it affects the algorithm as a whole. Using a software environment can save a lot of development time as it allows for easy interaction and fast prototyping of different algorithms.

Numerous image processing environments also come with built-in functions and filters that can further decrease development time. Great care should be taken when utilising these built-in functions and filters. Even though they may return quick and desirable results, they may be too complex to run efficiently on an FPGA. Once an algorithm has been prototyped in software it is possible to use the software environment to modify the design to closer resemble the functionality of the FPGA. Even though the parallelism will not be duplicated, the algorithm can be adapted to loosely mimic its functionality. This can simplify converting the algorithm to a hardware language, and helps reduce its complexity as a whole.

Both a modular and parameterised algorithm design can be very useful when developing and testing in both software and hardware. A modular design allows each filter or block to be tested individually and separates the algorithm into smaller simpler blocks. With each module compartmentalised it is important to clearly define and minimize the transfer of information between each module. This makes design, debugging, and testing much simpler as each module becomes localised. Similarly, making each block parameterised can simplify changes to the interaction of the blocks without changing their functionality. However, parameterisation can add some complexity to the design of each block. Even with the added complexity, it still benefits the development process by eliminating the possibility of accidentally introducing errors.

Communication between the FPGA and a computer is important. Many off-the-shelf FPGA development boards come with a variety of communication ports, such as USB, RS232, and general purpose IO. These interfaces are essential for transferring captured images from the sensor to the computer in order to begin the development of an algorithm. They can also be used to send commands to the smart camera to execute different functions, such as adjusting thresholds. This allows for settings to be changed during run-time without the need to recompile, thus reducing the total time spent compiling the algorithm. Results and processed images can also be transferred from the FPGA in order to test the functionality of the algorithm, to measure performance and to debug errors. A solid form of communication can be one of the most useful ways to test and develop the smart camera during run-time.

There are several other resources available to help debug an FPGA at run-time other than a solid form of communication. Many off-the-shelf FPGAs come with resources such as LEDs, switches, and buttons that allow for interaction with the smart camera. These can be used in many different ways, from displaying information, to enabling and disabling filters. This can help identify faults without the need for extensive simulation examinations.

Many of these techniques are also useful for designing more generic FPGA implementations. When developing any project on an FPGA development board, extra work needs to be done to develop base drivers to utilise different functions and resources on the board. While the development of these drivers is equally as critical as the developing of the image processing algorithm, the techniques and processes required are covered in other literature and is considered outside the scope of this thesis. This thesis will assume that the reader has already developed or is able to develop the base drivers needed to utilise the techniques described, allowing this thesis to concentrate on the image processing aspects only.

1.5. Goals

The aim of this thesis is to define and demonstrate techniques to aid the development of FPGA-based smart cameras. The techniques demonstrated in this thesis are meant as a guide to help others new to the field to quickly and efficiently develop their own projects. Appropriate use of these techniques should simplify the development process as well as reduce the development time. Furthermore, they should make modifications or preparation for future projects simpler by providing a logical and structured system to work from.

Two case studies will be investigated in order to demonstrate the usage and effectiveness of the development methods in a wide variety of situations. The first case study will demonstrate how the methods can be used to aid the development of a new project. The second case study will demonstrate how these techniques simplify the modification of a functioning smart camera implementation.

1.6. Overview

Chapter 2 will discuss much of the methodology for designing FPGA-based smart cameras. This chapter will focus on the techniques and design principles needed to develop an FPGA-based

smart camera. This will lay the groundwork for good design practices in order to shorten development time. These techniques will then be demonstrated in the two following case studies.

Chapter 3 contains the first case study, which will implement a global vision system for robot soccer. Robot soccer has been used as a test bench for advances in image processing and robotics for many years. From an image processing stand point it requires the acquisition and tracking of objects in a dynamically changing environment. This will require the smart camera to implement a number of filters to prepare the image, algorithms to extract features from the image, and make control decisions based on the information. This creates a perfect example of a basic smart camera, while allowing for relatively simple and time tested algorithms to be implemented in order to shorten development time. While the smart camera will be using many basic principles and algorithms for designing the image processing section, some new and novel approaches will also be developed and documented. This case study will be used to demonstrate the techniques needed to develop a new smart camera from initial designs.

The second case study will be outlined in Chapter 4. This will investigate an improved colour segmentation approach that can be used with the robot soccer smart camera developed in the first case study. This case study will demonstrate how making modifications to a smart camera, designed using the techniques outlined in this thesis, can be relatively simple, without having to redesign the whole algorithm.

The main goal of this thesis is to demonstrate good techniques for designing FPGA-based smart cameras. However, because of the new nature of FPGA-based smart cameras, the two case studies both investigate new and novel approaches to solving problems. The final chapter will discuss these results from the case studies, as well as identify areas for future research and final conclusions.

Chapter 2 Smart Camera Design

2.1. Introduction

Designing an algorithm for an FPGA is not straightforward. Unlike software algorithms, which follow a serial sequence, parallel algorithms can execute multiple commands at once. The advantage is that parallel algorithms allow for faster processing as many functions can be calculated simultaneously. However, this can make the design very complex, especially as the algorithm becomes more intricate. The complexity of the design process can be made somewhat easier by designing modules for reusability in order to reduce redesigning similar algorithms and recompiling for testing. Furthermore, with the limited debugging resources available, a robust development strategy is needed to aid development and reduce development time.

This chapter will outline the techniques needed in order to design, implement, and test an FPGA-based smart camera. However, many of these techniques are not limited to smart camera development and may be used for many other FPGA designs.

2.2. Current Research

Despite all of the advantages that a FPGA-based smart camera can offer to projects requiring image processing applications, they are not commonly employed. This is due to the greater complexity and longer development time of a hardware implementation compared to more conventional software approaches. While there has been a lot of research outlining the use of FPGAs utilising image processing algorithms to create smart cameras, there is very little literature explicitly demonstrating the design and development techniques for FPGA-based smart cameras. However there are resources that are useful for outlying key aspects for FPGA algorithm development [14, 15] and implementing image processing on embedded systems [12, 16].

Kilts [14] outlines many basic and advanced techniques for designing an optimized architecture for any FPGA implementation. These methodologies are backed up with very detailed real-world examples in order to instruct the reader and reduce the steep learning curve inherent with FPGA designs. Kilts describes, in detail, the importance of data pipelines for passing information between different modules within the FPGA. Pipelines allow the algorithm to take full advantage of the FPGAs capabilities and increase overall throughput of the design. Kilts [14] also briefly discusses some possible generic development structures including testing schemes. These development structures suggest using software applications to design as well as provide limited testing capabilities in order to make up for the hardware environment's lack of debugging functionality.

Another useful resource for learning FPGA algorithm development is written by Mealy and Tappero [15]. While this publication is primarily aimed at teaching the basics of the VHDL language, a hardware descriptive language, it does outline examples and design principles for FPGAs. Because the nature of the text is aimed at teach new users the basics of an FPGA

language many of the techniques and examples tend to be over simplistic. While this is advantageous for readers new to FPGA development, it may seem obvious to other more experienced developers.

Caarls [16] writes in depth about the technicalities of designing image processing on embedded systems to create smart cameras. His thesis covers many techniques needed to design embedded image processing algorithms, while also providing simplistic examples in various languages and for multiple platforms. Though, all of the languages focus on C-based subset languages for both initial image processing design and implementation. The most referenced hardware includes microprocessors, various parallel SIMD processors such as Xetel and IMAP-CE, and even graphics processing units (GPUs). While the FPGA is mentioned in various sections, it is considered out of the scope of the study because of their "difficulty for programming and requirements for strict timing and low-level optimised hardware descriptive languages". Caarls chooses to initially develop all image processing algorithms using low-level C-based languages. Caarls mentions that "Image processing applications are often prototyped in higher-level languages, such as MATLAB." Although due to the simplicity and abstract nature of the developed algorithms, higher-level programming languages are not necessary. Therefore, this study doesn't document many of the techniques needed to help develop the initial algorithm and then test the results after implementing them in hardware. Caarls [16] delivers an in-depth study into basic and more advanced techniques for embedded processing, however it fails to address some of the key features for novices to this field. While it is a useful document that has its place for demonstrating different design techniques for smart camera development, it fails to fully consider the potentials and advantages of the FPGA platform.

One of the main requirements of a smart camera is its ability to apply image processing in order to prepare an image for data extraction. Bailey [12] outlines many techniques needed to implement image processing algorithms onto an FPGA. While this resource does not demonstrate techniques specifically for smart camera design, much of this resource can be directly used for the development of smart cameras. To this end, many basic and advanced techniques are examined for remapping software algorithms into a parallel FPGA environment. Furthermore, through the use of examples, many of the common errors and pitfalls of FPGA development are examined and solutions are demonstrated. Other than architecture development, Bailey documents many of his own approaches for filters and functions such as various local filters, connected component, histogram operations, and more.

On the other hand, Wilson [17] is more designed as a reference guide for those with FPGA experience rather than an instructional guide. While this does contain techniques and examples for implementing image processing on FPGAs, it is very limited and vague. Wilson concentrates more on the theoretical side and resource allocation techniques (such as different memory components) rather than algorithm development. However, the examples for component design, such as serial communication and VGA interface, are very comprehensive. These examples outline the basic theory and then demonstrate functional and optimised VHDL algorithms. In short, this resource is ideal for quick implementation of basic and generic FPGA functions, such as communication modules and external interfaces.

2.3. Development and Testing Scheme

When developing any kind of image processing solution, it is very important to follow a procedural development plan. This is even more important for hardware solutions, as debugging FPGA implementations can be very difficult. Furthermore, the long compilation times of hardware languages means that making even small changes to an algorithm is no longer interactive, hampering the design exploration process. Therefore, the initial algorithm is usually designed in a software image processing suite, such as Matlab. This allows for quick prototyping and testing of algorithms with visual and numeric results. Since the image processing algorithm development process is largely heuristic, it involves a lot of trial and error. Once the algorithm has been developed to a practical stage the resulting work can be optimized into a hardware language. This allows the algorithm to be fully tested by multiple scenarios, and provides a standard to test against after it is remapped to hardware.

In the most basic form, a development plan will include gathering test images, developing the algorithm design, and implementing the solution. Figure 1 expands on this principle to illustrate the process to fully develop a smart camera on an FPGA.

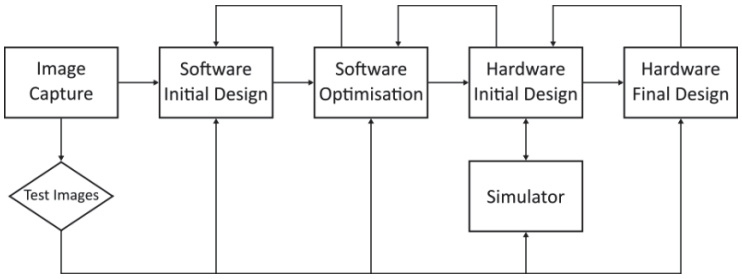


Figure 1 - Development and testing scheme for an FPGA-based smart camera

The first step in the design of any image processing algorithm is to capture a set of test images from the camera. A basic camera skeleton algorithm will need to be developed on the FPGA to allow for images to be taken from the camera and transferred to a computer. Figure 2 shows a basic image capture design that can be implemented on an FPGA development board. This will be the basis for the skeleton camera design which will be used to develop the following case studies.

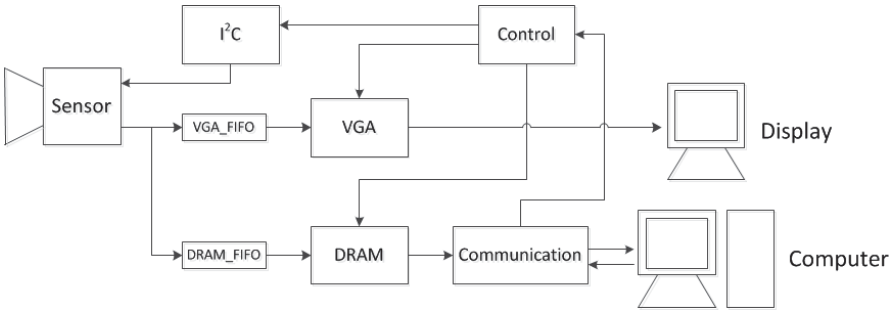


Figure 2 - Block diagram for a basic image capture architecture on an FPGA

During initialisation the *control* unit initialises the camera settings (resolution, pixel skipping, gain, etc.) via the *I²C* controller. Under normal operation the captured pixels are loaded into

the *VGA_FIFO* only. The *VGA_FIFO* acts as a buffer to transfer the image data from the camera clock domain into the display clock domain. The *communication* module receives the commands from a host computer and controls sending the test images back to the computer. Depending on the communication options available on the development board, as well as communication complexity, bandwidth, and speed, will determine which protocol is used. For simplicity, an RS-232 protocol is chosen for this example. Basic serial commands include camera adjustment via I^2C, and commands to initiate a test image capture. When an image capture command is received the *control* unit waits until the start of the next frame, to ensure a full frame is captured. At the start of the next frame the pixel data is loaded into the *DRAM* memory unit via the *DRAM_FIFO*. Memory is needed to temporarily store the image, to avoid data loss, as the *communication* module will likely transfer the image much slower than it is captured. Once the image has been saved it is transferred to the *communication* module to be sent to the computer.

The next stage in the development is to start designing the algorithm in a software environment. The desired characteristics for a good image processing development environment is for it to be interactive and to allow for fast prototyping of algorithms. Matlab is one such package. This allows various design decisions to be effectively explored to get the desired results from processing the image. Figure 1 incorporates two software design stages, the initial design, and the optimisation.

The aim of the initial design is to develop an algorithm and make sure it works as intended, as quickly as possible. At this stage, it may be easier and quicker to use built-in image processing functions to get the basic functionality of the algorithm. An example of this could be implementing standardised Bayer interpolation, YUV transforms, or filters, etc., which are built into Matlab. This saves development time and allows these filters to be tested without needing to program them from first principles.

Once the initial design is complete, the algorithm needs to be optimised to more closely follow the hardware functionality. Matlab generally uses double precision floating point in its calculations, whereas most FPGA designs are implemented using fixed point or integer arithmetic. All equations and built-in functions will need to be replaced with limited precision functions that match those which will operate on the FPGA. At this point, reduced precision can be explored to optimise word lengths. It is much faster to explore such options in software than directly in hardware, because software compilation and simulation times are significantly shorter in the software environment. Planning the hardware implementation is made easier as the functionality of each algorithm block can be examined and optimised. Consequently, any potential implementation problems or limitations can be identified before trying to develop a hardware solution. The output also provides a standard to test the hardware implementation against, as the results from the hardware application should be identical to the software implementation when testing the same image.

Once the algorithm is designed in software it can be adapted for a hardware implementation. Parallel algorithm design is made more complex because there are multiple processes running simultaneously. Not only does the logic and functionality need to be correct but all the

processes must also be synchronised. Hardware adds a temporal aspect that must be explicitly handled, unlike serial programming which handles this implicitly. This can make debugging more difficult with multiple segments of the algorithm running in parallel. In software, each step executes one instruction. This makes error identification relatively easy as only one thing changes each step. In hardware, multiple parallel processes generate numerous changes with each clock cycle. This can make isolating an error very difficult.

The first step to debug a hardware algorithm is to create a test bench within the simulator to emulate the functionality of the smart camera. It must emulate streaming an image from a sensor (reading from a file), and propagate the pixels through the image processing algorithms, finally collecting the results within another file. Using this test bench, it is possible to test each algorithm segment individually or the complete algorithm using the same test images used to develop the software implementation. To speed up the simulation, smaller sub-images designed to test a specific scenario can be used if the simulator has been designed appropriately. The simulation is implemented on a computer so it actually operates serially, however it will simulate parallel operations when intended. This will accurately simulate the operation of the algorithm, though it will take significantly longer to complete the operation. Another advantage of the simulation is that it allows a clock cycle by clock cycle overview of each operation, such as when registers are written to, or what values are present in a FIFO etc. This can be very helpful to identify timing or synchronisation errors. The main purpose however, is to test the image processing algorithm and compare the results to the software algorithm. The results of the simulation should identically match the software results. Any differences present at this time could mean an error in the hardware implementation. Subsequent testing and debugging is required to correct the error.

A disadvantage of the simulator test bench is that it must be exclusively designed in order to emulate the functionality of the smart camera. This can introduce unintended errors or limitations that may not be applicable in the actual hardware implementation. Special care should be taken when creating the simulator to ensure that it operates as intended, and any faults that occur are not being caused by the simulator itself.

Finally, once the algorithm is working in the simulator, it is time to test the algorithm on the hardware itself. This compares test images processed on the FPGA with identical images processed by software. This can help identify limitations in the algorithm by comparing results calculated using different approaches. There are two ways to test the algorithm on the FPGA. The first is to upload a previously captured test image onto the FPGA, process the image, and then download the results back to the computer for comparison with the software results. The second method captures a new image, stores it in memory, and then processes it, avoiding the need to upload a test image. Then both the test image and results are downloaded to the computer, the test image is processed using the software implementation and the results compared. The advantage of the first method is that the camera is not actually required for testing, once the test images have been collected. Hardware emulates the camera by providing the pixel stream from memory. This can be helpful when the smart camera will be operating in areas where it may not be possible to have a computer for a long period of time, such as

outdoors. However, the advantage of the second method is it can help find limitations in the image processing algorithms that were not considered by the initial test images.

2.4. Hardware Design Principles

There are many things to consider when developing the hardware algorithms for an FPGA-based smart camera. Developing and testing an algorithm in software doesn't ensure it can be optimized for hardware implementation. This section outlines some of the basic, but important, design considerations and techniques that should be used when developing any FPGA project.

2.4.1. Programming Languages

Put simply, an FPGA is a collection of programmable logic gates that can be wired together to produce a digital logic circuit. A hardware descriptive language (HDL) is used to represent the design of the circuits within the FPGA. While there are many types of HDL, during compilation they are all translated into a circuit based on available and needed FPGA resources and timing analysis. Both VHDL [18] and Verilog [19] are the industry standard for HDLs and therefore are the most commonly used. These languages are quite low-level languages, not to dissimilar to assembly code. The main advantage of these languages is their ability to make very streamlined and efficient algorithms, allowing them to make changes at the resource allocation level.

Low-level languages such as these can make algorithm development quite complicated and tedious. To implement algorithms efficiently sometimes requires dozens of lines of code to describe even simple circuits. For this reason, it can be difficult to program complex image processing algorithms with VHDL and Verilog. In an attempt to reduce the complexity of programming FPGAs other higher-level languages were developed, such as Handel-C. These languages allow for a more abstract approach to programming and rely on the compiler to manage a lot of the low-level details rather than the programmer. This makes these languages very powerful for larger, more complex, algorithms; however, the cost is that the design may be less resource efficient.

Similarly, it is possible to use serial languages such as C/C++ for both the algorithm design in software and hardware implementation. This is generally implemented using a program language interface to provide a bridge that allows the high-level commands to execute with low-level resources. "Although there have been many attempts to automatically translate a C program to a parallel architecture, they are invariably inefficient, or efficient only to a small subset of C programs." [16]. This can also lead to long compiler times, which slows down development.

Therefore, when choosing an HDL, there is generally a trade-off between resource efficiency, and programming complexity. The low-level languages, such as VHDL and Verilog, have better resource control. Conversely the high-level languages allow for relatively simple development for complex algorithms.

2.4.2. Modularity

Organisation, readability, and reusability are all very important aspects of FPGA algorithm development. Implementing a modular design methodology can not only improve the readability and functionality of the algorithm but also promotes reusability of the modules themselves.

Clearly defined boundaries are needed in order to simplify the development of any FPGA design. It is easiest to consider a module as a black box around the required algorithm, with specified input and output parameters. This, in essence, isolates the algorithm and allows the module to operate independently from other processes. In general, a module can be specified as a high, medium, or low-level module, depending on its size and function. A low-level module would control a single process (such as FIFO, multiplexors, etc.). Medium-level modules would control a small procedure, usually made up of low-level processes, such as memory access and I^2C communication. High-level modules are used to define large sections of code (whole filters, communication, camera control, etc.) and usually comprise of several medium and low-level modules.

Low-level and medium-level modules are important for controlling the basic operations and features of the FPGA. Designing a lot of the functions needed to operate the FPGA is an important first step of any FPGA development. These basic functions are usually accumulated into a single library for each FPGA or development board. The library can be thought of as a collection of drivers that are needed in order to utilise certain features provided by the FPGA development board. While this concept is important, it is well documented [14, 15]. The remainder of this chapter will concentrate on describing the importance of high-level modules.

Splitting the smart camera algorithm into several high-level modules has a few important advantages. First it outlines a clear path for data flow by connecting modules together with data lines. While it is important to clearly establish this design from the start, utilising modules allows for the processing path to change during development relatively easily, allowing modules to be rearranged, added, edited, or removed. Another significant advantage during the development stage is that modules provide a way of isolating sections of the algorithm. This allows individual filters and functions of the image processing algorithm to be designed and tested separately speeding up the development time.

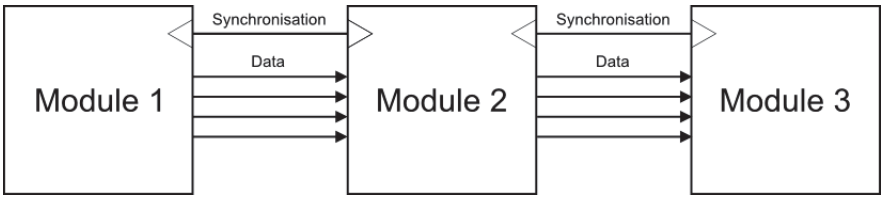


Figure 3 - Block diagram demonstrating module connectivity

Figure 3 shows a basic image processing design using a high-level module approach. In this example, each black box can contain a separate filter or process, and the processed pixels are passed between each module via data lines. In hardware, all modules run concurrently.

However, since each module is a separate entity, each will generally require a different processing time (latency). Therefore, the use of a data synchronous control signal is needed in order to keep the algorithm, as a whole, synchronised. While it is possible to apply this logic globally, from a separate module, a simpler approach is to manage the synchronisation within each module. This allows the modules to be modified individually without having to change other signals or other modules.

2.4.3. Parameterisation

One disadvantage of a modular algorithm design is that even small changes could require the module to be rewritten to account for the changes. For example, changing a filter to operate with 12 bits per pixel instead of 8 bits per pixel. Such a change would require a tedious and error-prone task of examining the design and making all of the appropriate adjustments. The module would then have to be retested, adding extra development time to not only make the changes but also resolve any errors that occur.

The idea of parameterisation is to make the modules more generic. This can be achieved by adding adjustable parameters that are resolved at compile-time, to affect the hardware rather than the functionality of modules [14]. One of the simplest forms of parameterisation for high-level modules is to have all register widths defined by a single global variable. By adjusting the global value the changes will propagate automatically through the algorithm the next time it is compiled. This allows adjustments to be made quickly without changing the functionality of each block. Development time can be reduced considerably as there is no need to retest the module.

Parameterisation can also benefit medium and low-level modules by allowing more commonly used modules to be reused for different purposes. An example for a medium-level module would be to allow the communication baud rate to be adjustable for RS232, or different width memory blocks for DRAM operation. For low-level modules, this can be as simple as designing a variable width adder or multiplexor. Ideally, this allows a few modules to be reused in different situations rather than developing new modules each time.

There are disadvantages to parameterisation however. Unlike hard-coded variables, with clearly labelled widths, the width of parameterised registers is defined indirectly. This can reduce readability of the algorithm. Another disadvantage is that designing a generalised module can be more complex than a single purpose module. Even so, the advantages for reusability, and reduced re-development time for changes, should outweigh any difficulties during the modules initial design.

2.4.4. Stream Processing

With image processing, there are a few different ways to process a video stream. In software, the most common approach is to first capture the image into memory and then begin processing. In software, processing is sequential, and the use of memory in this way decouples the processing of each operation. Furthermore, memory can be used to hold intermediate results. While this approach can be replicated on hardware, this does not exploit the

concurrency offered by a hardware design, and the processing rate is limited by memory bandwidth.

Another approach in hardware is to utilise stream processing. This applies the image processing algorithms in real-time to the pixel stream from the camera [7]. This reduces the need to store or buffer the image, thereby saving a significant amount of memory (and the time required to write to, and read from, memory). Moreover, this requires the processing to begin from the moment the first pixel is available, unlike other approaches that must wait for a fully buffered frame. This reduces the latency of the image processing stage, as results can be fully extracted mid frame for many algorithms. Furthermore, reducing the overheads of memory operations can enable the operating speed of the camera, and by extension the frame rate, to be increased [12]. This allows greater accuracy in tracking rapidly changing scenes.

However, stream processing does have its disadvantages. Because the image is not stored, strict timing requirements must be established to prevent data loss and keep the algorithm synchronised. The processing speed is usually governed by the camera pixel clock. A throughput of one pixel per clock cycle requires the processing algorithms to be relatively simple. Furthermore, the algorithms are limited to pixel based operations or limited neighbourhood procedures utilising a windowed approach, which will be discussed later. Other algorithms can be used, but are harder to implement with stream processing. Despite the strict timing requirements, and somewhat limited algorithm design, stream processing still provides a robust and low-latency approach that can be implemented within an FPGA.

2.4.5. Pipelining

Designing hardware algorithms can be difficult, as both the logic and timing need to be explicitly handled. This is not the case with software, as only the logic is necessary and the timing is often irrelevant. Consequently, the timing requirements can often affect the design of hardware algorithms. This is especially true with the strict timing requirements of stream processing.

Pipelining is a useful technique that allows complex algorithms to maintain a throughput of one pixel every clock cycle. This is achieved by breaking the algorithm down into a series of stages, with each stage taking one clock cycle, similar to the principle of an assembly line. In this way pixels are input every clock cycle, propagate through the pipeline, and exit processed every clock cycle without loss of data, as demonstrated in Figure 4.

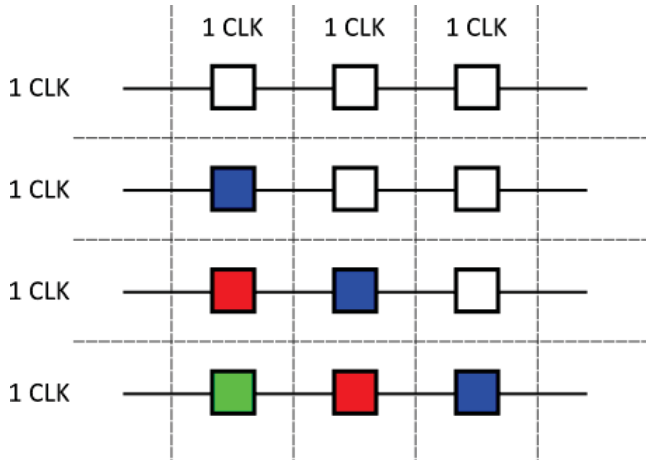


Figure 4 - Basic example of a pipeline algorithm processing a concurrent pixel stream over multiple clock cycles

Pipelining allows algorithms to operate at high clock frequencies by increasing throughput in most situations [14]. This allows multi-stage algorithms to be implemented without interrupting data flow or introducing memory components. The trade-off is that pipelining adds slightly more latency. Another disadvantage is that pipelining can increase the complexity of the algorithm, especially where there are multiple parallel data paths through a module. Such cases require internal synchronisation within the module.

2.5. Module Design

Following the principles of modularity, pipelining, and stream processing it is possible to outline some basic module structures. Module designs can be categorised into two types, synchronous and asynchronous, depending on how information is processed and transmitted. Synchronous modules follow the strict timing requirements of stream processing and output processed pixel information every clock cycle. This is used to implement filters or pre-processing stages in order to prepare the image for feature extraction. While the term asynchronous describes a process that is not bound by a fixed clock cycle, in this context it refers to a module that only operates when valid data is available rather than every clock cycle. In this way, the module acts asynchronously by only activating when a valid signal is received, while still being synchronised by the same clock as all other modules. These modules are mostly used during the feature extraction stage. While these aren't the only ways to design modules for stream processing, these are the most common ways utilised in the case studies in this thesis.

2.5.1. Synchronous Design

To meet the strict timing requirements of stream processing, one pixel must be processed every clock cycle without interrupting the data rate. Successive modules must synchronise with each other. This is achieved with the use of synchronisation signals to identify valid pixels, horizontal blanking (end of line), and vertical blanking (end of frame).

Pipeline Design

A pipeline design spreads the process computations over many clock cycles without losing data or bandwidth. A simple synchronous module, as shown in Figure 5, applies a sequence of operations to the incoming pixel stream. Stream processing requires the latency of synchronous modules to be fixed, enabling the output synchronisation signal to simply be delayed by the number of stages in the pipeline.

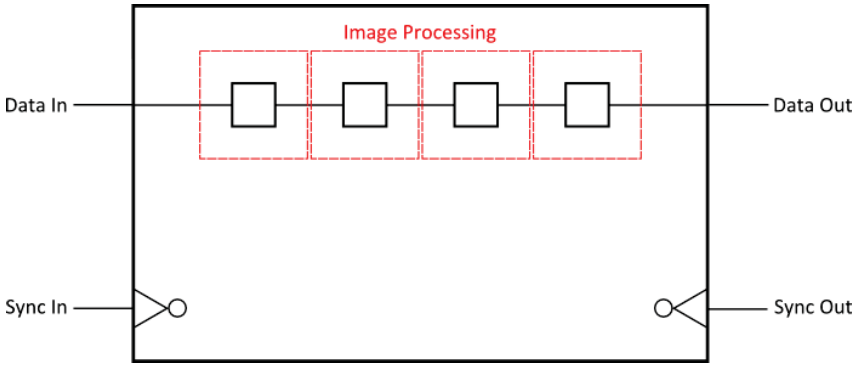


Figure 5 - Black box diagram of a synchronous pipeline design with a 4-stage pipeline

Window Design

However, some filters make use of neighbourhood or block processing, which require a small region of pixels to be available. A sliding window is commonly used to isolate the small region of pixels required to generate each output pixel, as demonstrated in Figure 6. With stream processing the window moves through the image in a raster scan.

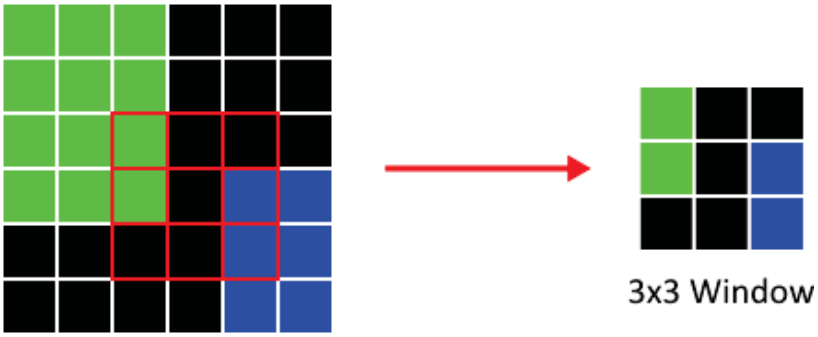


Figure 6 - Demonstration of a 3x3 window within a buffered image

This is straightforward in a frame buffered approach as the entire image is already available (subject to memory bandwidth limitations). However, when utilising a pixel stream, only one pixel is input each clock cycle. Rather than scan the window over the pixels, the window is stationary and the image is streamed through the window. The stream window has two components: a memory component to cache the data required from previous rows, and the window array which provides the pixel values within the window for the filter operation.

The window array is made up of an array of shift registers. Each clock cycle the pixel data shifts along, effectively scanning the window through the image for each output pixel. The memory

storage can be implemented using RAM blocks on the FPGA, set up as row buffers. These are set up similarly to a FIFO, with a delay of one image row. This allows for reusable and fast access storage for a few rows of the image at a time, which utilises only a small amount of the FPGAs memory resources. The diagram for the configuration of the window is illustrated in Figure 7.

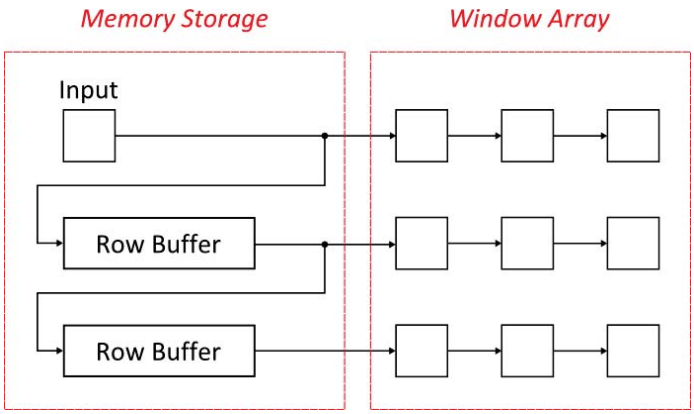


Figure 7 - Windowed design creating a 3x3 window

From this concept, it is possible to create custom sized and shaped windows. Then the image processing algorithm can read directly from the corresponding window registers, in parallel, to the window streaming operations. Figure 8 shows the diagram for the entire high-level module with a windowing component.

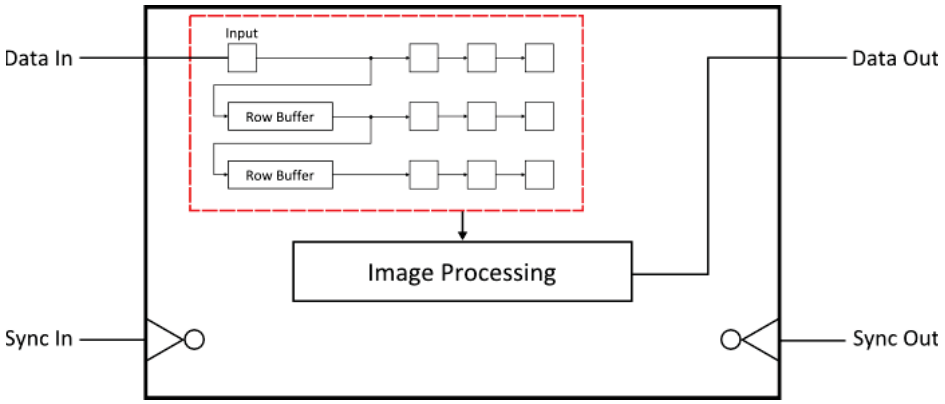


Figure 8 - Black box diagram of a synchronous window design with a 3x3 window

2.5.2. Asynchronous Design

The difficulty with implementing an asynchronous module is handling the irregular input data rate. The strict timing requirements can often be relaxed, allowing multiple clock cycles to complete processing. However, this can introduce data loss if data arrives too frequently. Depending on the complexity and expected latency of the image processing there are generally two design approaches to manage this with asynchronous modules: FIFO buffering, and pipelining.

A FIFO buffer is a robust way of temporarily storing incoming data if the module is currently processing. This is useful if the algorithm must be run separately on individual features, and when the processing takes multiple clock cycles where the chance of a data clash is relatively high. This scenario is most useful when the expected amount of data throughput for the module is relatively low, or the number of clashes is relatively small. On the other hand, pipelining removes the need for extra memory as it operates continuously. While this method is well suited for high data throughput, it can significantly increase the complexity, especially when data dependant processing is required.

FIFO Buffer

Depending on the algorithm it may be inevitable that it will require multiple clock cycles to complete. This is feasible if the average time difference between receiving new features is longer than the processing time. To avoid loss of data, a memory component will be needed to store or buffer the features while processing is completed. Generally, an on-chip FIFO buffer is most appropriate for this. However, if the data volume is significant, then it may be more appropriate to use external memory.

Figure 9 shows an illustration of the module design. The FIFO buffer passes data to the image processing section as it is needed. Input synchronisation controls writing incoming data to the FIFO buffer. The image processing provides the synchronisation signal to match when data is transmitted to the next module.

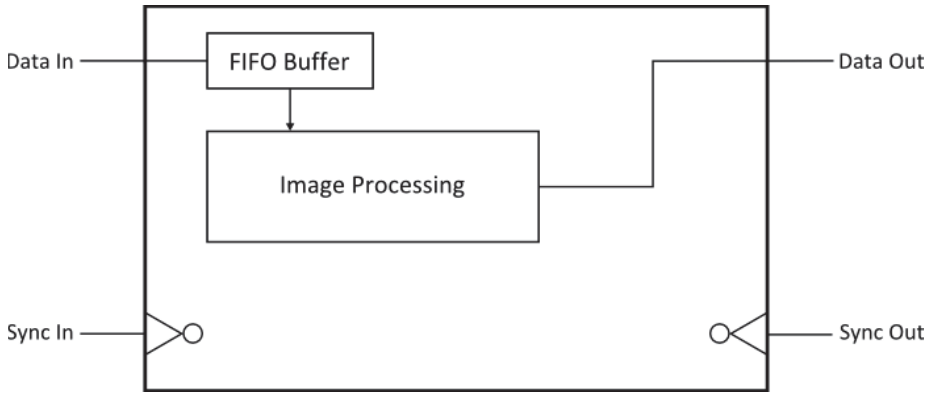


Figure 9 - Black box diagram of an asynchronous FIFO Buffer design

Pipeline Design

The asynchronous pipeline approach is similar to the synchronous pipeline design. The module operates every clock cycle using a predefined pipeline, using input synchronisation to indicate when input data is valid. It outputs data asynchronously after valid data has propagated through the pipeline. This design, Figure 10, operates with a relatively low-latency, and has the ability to process a high data rate without the need for extra memory.

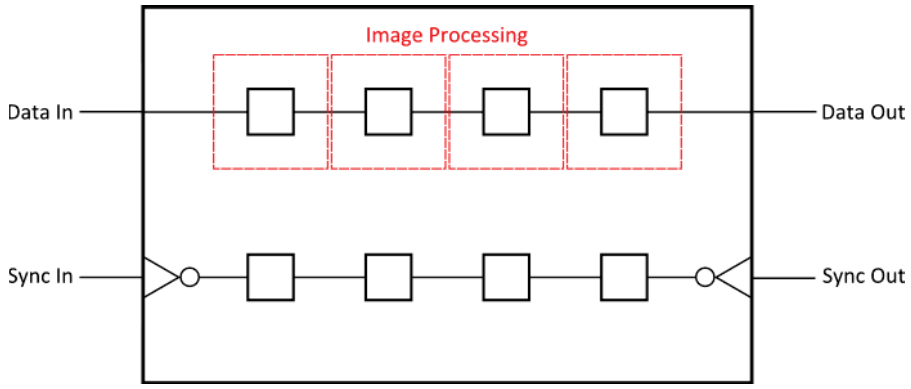


Figure 10 - Black box diagram of an asynchronous pipeline design with a 4-stage pipeline

Like the synchronous pipeline design, the pipeline is made up of separate stages that are carried out in parallel. Similarly, the processing rate is also fixed, meaning it takes the same amount of time to process all valid data, irrespective of when it is entered. Where this approach differs is how the module handles its synchronisation signal. In this approach the signal is propagated in parallel with the processing pipeline. This provides a transparent way of handling the synchronisation signal without introducing any complicated timing or counting system. Pipelining is only feasible with algorithms that can be broken down into a fixed sequence of steps that always take the same fixed processing time.

2.6. Summary

With these techniques, it is possible to solve or simplify many of the issues with FPGA-based smart camera design. A solid development and testing scheme can reduce the amount of time spent recompiling the algorithm and can make up for the lack of debugging resources at run-time. Modularity separates the algorithm into smaller pieces allowing easier testing and synchronisation of the algorithm, while also improving readability. Moreover, having predefined module designs allows for faster development of hardware algorithms. Parameterisation allows the algorithm to automatically adapt to changes made to global thresholds, reducing the time spent modifying each module individually and retesting. Furthermore, parameterisation can be used to alter the properties of the smart camera at run-time, reducing the need to recompile the algorithm to account for changes in environment (such as light or colour). Although it adds strict timing requirements, stream processing allows for a low-latency approach to image processing. In order to overcome the strict timing requirements of stream processing, pipelining can be used to process more complex algorithms over a longer time.

These techniques can be used to form the foundation for any FPGA-based smart camera. The following case studies will demonstrate the use of these techniques, and show how they can simplify the development process and reduce the development time needed to implement smart cameras.

Chapter 3 Case Study 1: Robot Soccer

3.1. Introduction

The first case study that will be investigated is the design and implementation of a smart camera for robot soccer. The aim of this case study is to demonstrate the development techniques needed to create a smart camera, while actually creating an efficient and robust smart camera for robot soccer.

Robot soccer has been used for years as a test bench for new research in the fields of image processing, automation and mechatronics [20, 21]. It allows for the combination of real-world interaction and control from machine based vision. Because there are many documented robot soccer algorithms [21-23], a relatively small amount of research is needed in order to develop a concept algorithm. However, the developed algorithm may seem simplistic or less reliable than other more complex methods researched in recent years. Though, the real advantage of this case study comes from overcoming common problems with algorithm designs and resource management, rather than developing a new robot soccer algorithm.

This case study will demonstrate the design of the smart camera architecture and individual image processing modules. The architecture will consist of the capture of pixel information from the camera, as well as communication and synchronisation needed to develop and test the camera. This will provide an environment in which various image processing modules can be developed to perform the specified task. The case study will also outline different modules needed in order to locate and identify the soccer robots and the ball. While many of the methods and techniques have been well documented by others, some algorithms had to be specifically redesigned and developed to operate within an FPGA environment. These new algorithms will be demonstrated and tested in conjunction with the overall smart camera development. This will verify that the techniques outlined not only work for converting known algorithms, but also help develop new ones as well.

First, this case study will investigate previous FPGA-based smart camera attempts. The basics of what is needed to develop a smart camera as well as hardware choices will be outlined. The basic algorithm overview will then summarise the different modules needed, and how they will interact with each other, while also explaining the testing procedures that will be used to develop the algorithm modules. Lastly each module will be individually and collectively defined in full, demonstrating design techniques and results, justifying all decisions made during the development process.

3.2. Context

This case study aims to design a smart camera that improves upon the image processing method used by Massey University's robot soccer system, as well as creating a platform for future study and development for smart camera research. The proposed smart camera must be robust against changes in environment (light levels) and have a significantly smaller latency than the previous system. Another goal would be to limit the hardware used to low cost off-

the-shelf components. This will reduce the development time by removing the need to first design and develop the hardware itself. Allowing the case study to concentrate on exploring the techniques needed to develop the algorithm.

The image processing latency of the robot soccer system can be described as the time measured between the capture of the first pixel to the sending of the last piece of positional data for strategy processing. In Massey University's existing application this time is measured to be approximately 15ms. While this may be indistinguishable from a human perspective, this can be problematic from a machine vision perspective. The robots used have a maximum speed of 1.5m/s, therefore by the time the robot's positions have been calculated, their physical location could have moved by as much as 22mm. Having high latencies reduces the effectiveness of strategies and impedes proper robot control. It is possible to compensate for higher latencies within the strategy algorithm. However, doing so increases the algorithms complexity and processing time, and may even prevent some strategies from being used. A more effective solution would be to reduce the latency as much as possible. This would allow for faster more accurate reactions and strategies to be employed based on real-world data, rather than predictions.

Leeser et al. [13] state that "By moving the most computationally complex part of the … algorithm into hardware and closer to the camera the amount of software computation is greatly reduced. The overall processing time is reduced to create a high-speed run-time solution". This reinforces the idea that by implementing much of the image processing in hardware, the overall processing time (and latency) will be reduced. The hardware algorithms used in [13] showed speed increases of 20 to 50 times that of their equivalent software algorithms, which allowed the two applications to run in real-time. Leeser et al. [13] also declare that by feeding the images directly to the FPGA it can begin processing the image immediately, removing the need to buffer or store the image before processing. It is shown that most of the time spent on processing a single frame is dedicated to buffering and storing the image in memory [13]. So, if the pixels are able to be processed without the need to save the data to memory, then the latency can be reduced considerably.

The parallelism and pipelining capability of a hardware solution can reduce the processing time of an algorithm compared to an equivalent software solution. Where a software solution can only perform one operation at a time, FPGAs can calculate multiple operations at once because it operates in parallel. Leeser et al. [13] outline the importance of a highly streamed algorithm that fully utilises the pipelining capability of a hardware implementation, thereby reducing processing time.

This case study will use Handel-C as the hardware descriptive language for programming the FPGA. Handel-C is based on standard C with added support for parallel processing. The learning curve for Handel-C is relatively smaller compared to VHDL or Verilog, allowing for easier development of larger and more complex algorithms. Massey University also has an extensive library of Handel-C drivers and modules for many Altera FPGA development boards, which should shorten initial development time. While low-level languages like VHDL and Verilog allow for finer manipulation and preservation of resources, the need to explicitly

specify algorithm control tends to make the design more complex and harder to read. Handel-C is a good choice for the HDL as it would allow a greater emphasis to be placed on the development techniques without adding extra development time and complexity.

3.3. Robot Soccer

In 2004 Massey University designed and implemented a robot soccer system. This system utilised a single CCD camera, in a global vision configuration, connected to two separate computers. The computers are used to process the images, using a software algorithm, and determine the strategies for each team with each computer focusing on a single team. The need for a reliable and accurate vision system is crucial, as the information collected will greatly influence the strategy and robot control stages.

In a global vision system, the camera is placed above the field looking down, giving the camera a global view of the field. Figure 11 shows the view of the field from the camera.

Figure 11 - View from the camera in a global vision configuration

In the FIRA small league [24], the playing field is 1550mm long by 1350mm wide. With six robots, three for each team, and a ball. Each robot has a unique coloured jacket on top to allow for individual identification, and to determine the robot's orientation. A common colour patch design is the quadrilateral design shown by Figure 12 [23, 25]. Whereby, four squares are used with various colour combinations in order to identify the team, robot, and orientation.

Figure 12 - Quadrilateral robot patch design

However, it was document by Sen Gupta et al. [26] that by using an oblique patch design, shown in Figure 13, the orientation can be calculated more accurately because the baseline between the orientation patches was longer. The patches shown in Figure 13 also allow for a variable team identification patch which would allow for more flexible processing for robot identification. Therefore, this case study will utilise these patches over the more common ones (although in principle the process is similar).

Figure 13 - Robots with an oblique robot patch design, and the ball

Each robot is 75mm x 75mm x 75mm and belongs to one of two teams: designated yellow and blue. The team can be identified by the central colour on the robot colour jacket, known as the team patch. The shape of the team patch is used to identify the specific robot. The three shapes, circle, square and rectangle are also different sizes, making the robots easy to identify by the shape and/or size of the team patch. The pink and green triangles are used to determine the robot's orientation, with the pink triangle corresponding to the left front edge of the robot. The orange ball pictured in Figure 13 has a diameter of 45mm.

3.3.1. Review of Algorithms

The main goal of the robot soccer vision system is to detect and isolate the colour patches on the robot, and use them to calculate the real-world position and orientation of the robots and the ball. This is usually achieved using a single global vision camera, whose field of view covers the entire field. Many approaches have been documented since the foundation of robot soccer in the mid-nineties. Most of them consist of employing software approaches to perform the image processing [26-29], however in recent years some have experimented with hardware approaches and smart cameras [30, 31].

A basic robot soccer vision system can be divided into three main sections, as shown in Figure 14: pre-processing, feature extraction, and object processing [21, 23]. Pre-processing takes the raw pixels and converts them to an appropriate colour space for thresholding, and applies filters such as edge enhancement to clean the image up. Many use the YUV colour space [26, 32, 33], some chose the HSV/HSI colour space [27, 28], and a few work directly in the RGB colour space [34]. Although, using the RGB colour space is rare as it is well documented that it is difficult to threshold, due to its susceptibility to lighting changes [23, 26]. The feature extraction stage thresholds the image, and begins discriminating the colour patches in order to extract measurements from their features. The most common method for feature extraction is connected components analysis. Lastly, the object processing phase calculates the higher-level features, such as position and orientation, for each robot.

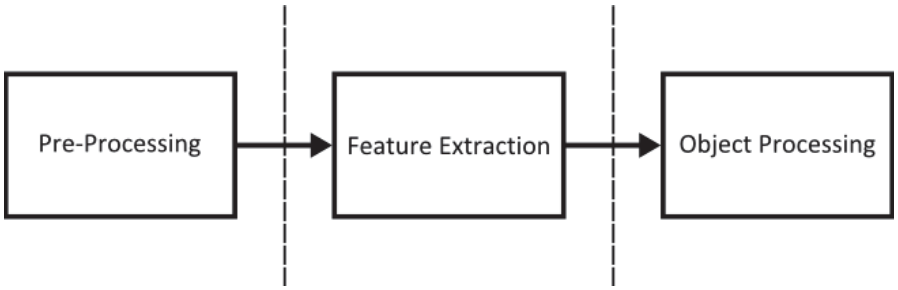


Figure 14 - Basic robot soccer image processing algorithm design

The most common difference between image processing approaches is how the colour transformation and the thresholding is implemented. With older approaches, the most common colour space was YUV. This was because the RGB to YUV transform requires only a relatively simple transformation matrix. This is much simpler than more complex colour spaces. However, it was documented that performing the YUV calculations was a bottleneck on the system and limited the overall frame rate [35]. To overcome this, many software approaches utilise a large YUV lookup table [35, 36]. This pre-calculates the YUV results for every possible RGB combination and gives the transformed values via a single lookup.

In recent years however, with the increase of computing power, many have attempted using other colour spaces such as HSV. HSV describes a colour in terms of hue, saturation and value, where hue is represented as an angle on a colour wheel, and saturation represents the boldness or strength of the colour. The transform from RGB to HSV is more complicated than to YUV, however HSV is a more robust colour space that is resistant to changes in light [28].

Colour segmentation is another area that has seen a wide range of research. The most common methods involve using discrete thresholds that are either manually or automatically tuned. While manual calibration is difficult to configure and quite susceptible to changing lights, it exploits the least amount of resources and requires a very low-latency. The main disadvantage of this approach is that each threshold will need to be set and fine-tuned manually, and recalibrated in the event that the environment changes. In many software cases, a graphical user interface is used to allow fast calibration of the colours. Sen Gupta [26] suggests selecting a coloured pixel within a captured image, and initialising the YUV thresholds at a fixed offset from the measured value. These thresholds can then be fine-tuned in order to achieve better results.

Brusey and Padgham [27] describe two different methods for colour segmentation. The first works by selecting pixel samples and deriving thresholds based on the mean and standard deviation. Afterwards a bounding box is formed around the samples and all values within a specified number of standard deviations are linked with that colour. These colour values then form lookup tables for faster processing. However, Brusey and Padgham found this method to be flawed based on the initial assumptions. First, it assumed all distributions would be even and neat. This was not the case, as glare and uneven light conditions caused the standard deviations to spread. Secondly, the calculated thresholds tended to overlap due to the difference in light levels. Therefore, this method is largely susceptible to uneven and changing light levels.

The second method described by Brusey and Padgham [27] made an observation that all colours had a bright area and a dark area. By selecting a sample in both light and dark areas the range of the colour values could be found. This allowed for a more accurate standard deviation by centring on an average value that was statistically between light and dark. It was reported that using this method alone would only allow half of the robots to be reliably found over a period of time. The other robots couldn't be properly identified due to overlapping thresholds. However, utilising this method in conjunction with manual adjustments to the thresholds, to avoid overlaps, achieved a much higher success rate for classifying robots.

Other than manually setting the thresholds, automatic colour segmentation methods have also been researched. An automatic approach gathers colour thresholds directly from the image itself without human input. It is believed that these are more resilient to changes in light because they automatically change with the light conditions [37]. While these methods often reduce the amount of time needed to calibrate the thresholds, they tend to increase the complexity of the algorithm. This therefore poses a challenge to FPGA implementations.

Amaroso et al. [28] outlines an automatic approach for colour segmentation, based on a neural network and histogram approach. This paper makes use of a standard HSI colour space, in particular concentrating on the hue angle. The hue is used to compile a global histogram. The histogram is then smoothed to make the valleys more distinctive by giving them a specific circular property using a valley detection function. These values are then entered into a feed-forward neural network to define the ideal thresholds within those valleys. Based on the results, a worst-case scenario of 10% misclassification was found when testing against 60 patterns, across various lighting conditions. The test images not only contain varying light levels within the lab environment, but also different lighting conditions taken from other locations. This allowed the testing groups to be comprised of not only uneven light patterns, but also differing tones, hues, and lighting shadows caused by the use of different lights and light sources. This demonstrates the method is very robust for learning and compensating for a wide variety of different conditions and environments. While this approach returns impressive results, it would be difficult to implement on an FPGA due to the amount of resources needed to implement the neural network.

During the feature extraction stage, most researchers employ a connected component labelling and analysis algorithm [23, 26, 29]. This groups neighbouring pixels with similar labels into a large group, or blob. While there are many versions of connected components labelling, most are based on a standard two-pass approach [26]. As well as grouping the pixels, the connected components analysis allows specific features to be calculated from each blob that can be used during object processing.

Object processing is largely dependent on the type of colour patches that are used. Nevertheless, a common method, outlined in [25], is to first calculate the centre of gravity of two coloured patches opposite from each other. The vector between the centre of gravities is then used to define the angle of the robot. The robot's identity can be determined based on colour combinations, as is the case for quadrilateral style patches [25], or via some form of shape or feature recognition when using oblique patterns [25].

To summarise, an FPGA algorithm would need to be implemented on a pixel stream without buffering the image. This would allow for the raw Bayer patterned pixels to be transformed into RGB and then into a more robust colour space. While YUV and HSV colour spaces are both common, the HSV conversion is more complex. In order to keep the resources to a minimum a YUV colour space will be used. Due to the FPGAs capability to process high volumes of low-level processing, a YUV lookup table will not be required like previous software approaches [35]. Instead it is more efficient to perform the transform on each individual pixel using a dedicated pipeline. Similarly, in order to reduce complexity and resources, a manual box method threshold technique [26] will be used. While most feature extraction stages utilise a multi-pass connected components labelling algorithm, this is less practical within an FPGA because of the need to buffer the intermediate image. Therefore a single-pass algorithm will need to be developed such as in [38], or other similar approach investigated.

3.3.2. Review of Smart Camera Architecture

In a smart camera, the image processing isn't the only important research topic. The architecture and method of controlling the images and extracted results can be just as important and varied. For use with robot soccer it is important to keep the latency as low as possible. The latency is the time elapsed from the beginning of the first pixel to the last processed object location.

While not FPGA-based, Bender and Beekman [30] document a method where a cluster of smart cameras is used to locate objects within a field. This method uses four consumer-grade digital cameras to extract information and output data for further analysis. Each camera is placed in a corner of the field and calculates the offset angle of the objects in relation to the camera's centre of vision. These angles are then relayed back to a centralised server for triangulation. Knowing that the field is an NxM grid, the server uses the four angles to calculate the X and Y coordinates of each object. The advantage of this approach is its use of off-the-shelf consumer grade cameras, because the cameras already contain WiFi communication and basic RGB demosaicing. This would greatly decrease the development time and allows faster communication bandwidth, with relatively accurate RGB conversions. However, since each camera is only aware of its localised information, this design is reliant on a master device in order to derive the physical location. Furthermore, even though the cameras can operate at fast clock cycles to calculate the angle offsets, the server still acts as a bottleneck because it needs to first accumulate the values from all four sources for maximum accuracy. The communication needed to synchronise the cameras and transfer the data adds extra latency and complexity to this method. This is not the case with a single global vision camera, as it is not reliant on any separate entity. Therefore, in order to keep the design as low-latency as possible a single camera approach should be used.

Broers et al. [31] outlines a method of using a single smart camera as the global vision system for a robot soccer team. Designing application specific smart cameras can be time-consuming and reliant on hardware [31]. Therefore, this approach created a more generic and versatile design that uses an optimised software algorithm for faster development. The design allows more conventional software image processing algorithms to make use of parallelism in order

to decrease processing latency. The parallelism is provided by multiple linear processor arrays (LPAs) interconnected via an FPGA. The LPAs perform all of the image processing operations on pixels from the image sensor in real-time. The FPGA is used as a parallel transfer pipeline to pass data to and from the LPAs and stores the final results onto a memory card. It was revealed that utilising a software algorithm decreased the processing latency by 42% using parallelism, compared to a single processor approach. However, software algorithms can be inefficient when implementing parallelism compared to hardware specific approaches [16]. Furthermore, each processing element within the LPA has the potential to process at different rates, depending on the filter or operation being performed. This can make synchronisation complex and leads to further inefficiencies within the algorithm, adding latency. As a result, this approach is relatively fast to develop at the expense of longer processing latencies. For the robot soccer case study, the smart camera is being designed for a specific purpose, therefore a generic design is not required. Additionally, in order to reduce the latency and complexity as much as possible it would be more practical to implement the smart camera using a single FPGA, and an optimised hardware specific algorithm.

3.4. Design Specifications

The primary goal for this case study is to develop a robust smart camera that operates with a single FPGA and camera. The purpose of this smart camera is to replace and improve upon the existing robot soccer vision system implemented by Massey University. In order to be beneficial for robot soccer this implementation will need to be as low-latency as possible. Therefore, to reduce the latency as much as possible a few design specifications can be set:

- Only transmit finished robot data
- 640x480 frame resolution
- Use stream processing
- Highest possible frame rate

All image processing will be completed on the smart camera, and only robot data will be transferred to a host computer for strategy processing. This will reduce the communication bandwidth considerably as the robot's identity, position, and orientation has far fewer bits than a full image.

During the initial development of the smart camera, a VGA display will be necessary in order to help debug the algorithms. In order to allow this, a standardised frame size is preferable in order to keep the frame synchronised and correctly proportioned on the display. 640x480 is known as 'standard definition' and is commonly used with VGA connections. This resolution would allow for faster processing compared to larger resolutions, as it contains fewer pixels. It is also double the resolution used by Massey University's existing camera (320x240) allowing for greater image detail.

Many FPGAs do not have enough resources to store large resolution images without the use of external memory. Therefore, stream processing will be used in order to process the pixels directly without the need for memory, if possible. This should also allow for faster processing

and reduced latency as the image can be processed immediately from the first pixel, instead of having to wait for a full frame to be buffered first.

Limiting the resolution to 640x480 allows the frame rate to be increased. Most VGA screens allow for a maximum of 75 to 80 frames per second. Depending on the FPGA, camera, and algorithms it should be possible to reach this frame rate, if not faster. This is nearly 3 times faster than the 30 frames per second possible by the existing CCD camera.

3.5. Hardware

3.5.1. FPGA

To shorten development time, only off-the-shelf FPGA development boards were considered. There are two main FPGA manufacturers, Xilinx and Altera, both produce products with similar capabilities. Massey University uses Altera FPGA-based development boards made by Terasic in their curriculum: the DE0, the DE0-Nano, and the DE2-115, so these will be considered for this case study.

The DE2-115, while the most powerful, is too large to easily mount above the playing area. The DE0-Nano is the smallest of the three options and would be best for integrating within a compact smart camera. However, it lacks many of the features that would aid in initially developing the algorithm. These peripherals, such as VGA and serial communication, can be added via a daughterboard, but this would take time to develop and test the components. Since the DE0 already has the on-board peripherals, this makes the DE0 preferable for the initial design of the algorithm. Figure 15 shows an image of the DE0 FPGA development board.

Figure 15 - Terasic DE0 Development Board

3.5.2. Camera

At first glance, a USB based webcam would seem like a good option as an image sensor. There is a wide variety of webcams that can be bought for under $100 NZD with resolutions of 1280x720 (720p) or better. Webcams provide the hardware for creating and streaming high quality colour images. However, within the FPGA environment, the complexity of the USB protocol makes it difficult to interface with hardware. Additionally, many manufacturers use propriety compression and encoding schemes over USB, making it difficult to access the pixel

data. Furthermore, these cameras often employ fixed Bayer demosaicing algorithms to create colour images. Without documentation or understanding for how these algorithms operate it is difficult to customise or alter the output to better suit the project requirements.

Another option is to utilise a camera development kit. These usually consist of an image sensor soldered to a circuit board with the necessary components to control the camera and read pixel data. Camera development kits usually allow direct control of the camera characteristics, such as the operating clock speed, gain and exposure control, readout mode, etc. Furthermore, these cameras generally do not come with built-in Bayer demosaicing algorithms, allowing a greater amount of customisability for colour conversion and pixel readout. Terasic also make a 5 megapixel camera development board, the TRDB-D5M, shown in Figure 16, designed to interface with many of the development boards produced by Terasic, including the DE0. This image sensor can capture video at a resolution of up to 2592x1944 with 12 bits per pixel at 15 frames per second. However, windowing can be used to read out a smaller region, allowing the frame rate to be increased. The TRDB-D5M also enables direct access to the sensor features such as shutter control, gain and exposure timing, window control, pixel binning and skipping, etc. With window control, it is possible to reduce the resolution to 640x 480. At this resolution, reading out pixels at the cameras maximum clock speed (96 MHz) it is possible to achieve a frame rate of 120 frames per second, well above the design specifications. This makes the camera very customisable, and a good choice for the case study.

Figure 16 - TRDB-D5M Camera

The TRDB-D5M uses a colour filter array to capture colour images. While there are different colour array filter patterns [39], the most commonly used pattern is the Bayer pattern [40], which is used by the TRDB-D5M. Each pixel is only able to measure the luminosity of one of the red, green, or blue channels. The Bayer pattern alternates the red, green, and blue pixels in a checkered pattern, shown in Figure 17.

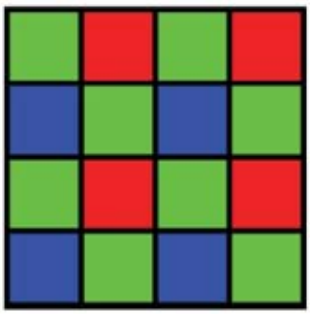


Figure 17 - Bayer Pattern

Like many camera development kits, the TRDB-D5M does not have a built-in Bayer demosaicing filter. Therefore, one will need to be developed within the robot soccer algorithm to convert the raw pixel information from the sensor into colour images.

Other than windowing, two other common readout modes are pixel binning and skipping. Skip mode [41] is where 2x2 clusters of pixels (2 green, 1 red, 1 blue) are read out in groups skipping every second (third, or fourth, etc.) group. Sub sampling in this way allows the resolution to be reduced while maintaining the same angle of view. However, because of the Bayer pattern, the pixels read out are not uniformly spaced, as shown in Figure 18. This can lead to artefacts in the resulting colour image and possibly reduce the image quality.

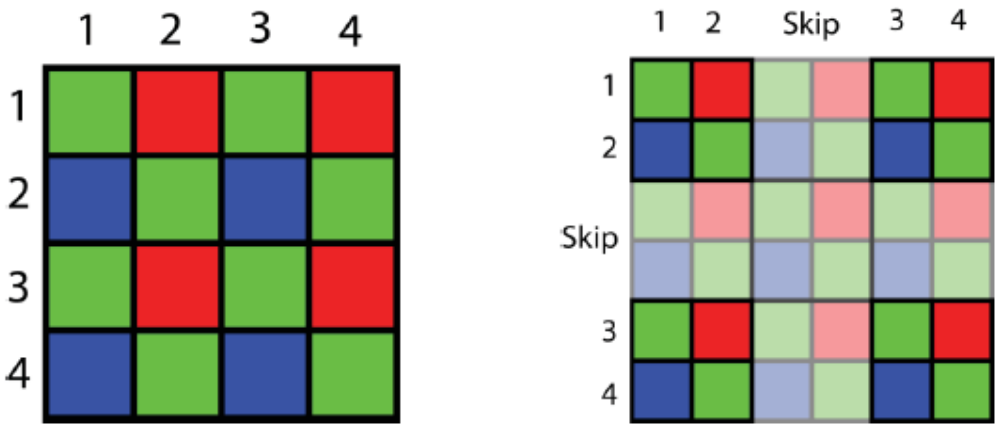


Figure 18 - Actual physical location of captured pixels

Binning combines together the corresponding pixels from neighbouring pixel clusters. In most cases, binning can be used to reduce the colour noise within an image, at the cost of resolution.

The camera also has blanking intervals at the end of each row, and frame, as pictured in Figure 19.

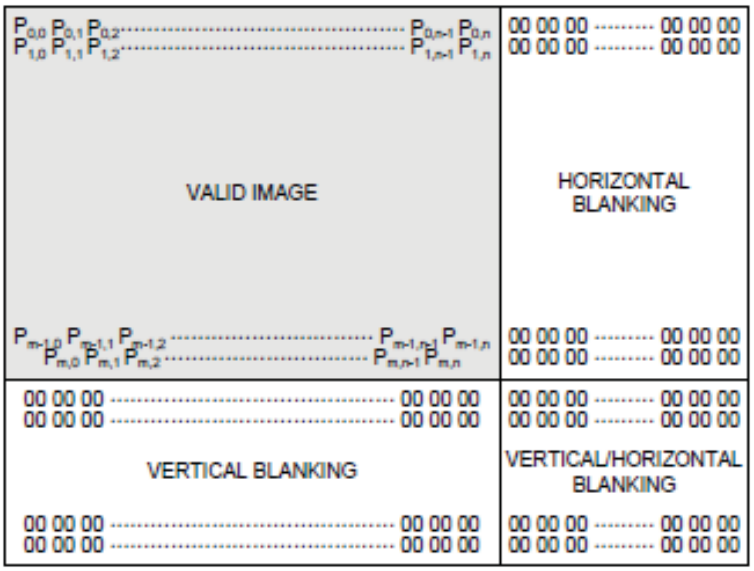


Figure 19 - Camera blanking areas (image courtesy of Terasic hardware specifications [42])

During the blanking period, no pixel data is sent. Therefore, synchronisation signals are used to indicate active pixels on a row, and active rows within a frame. Blanking areas can provide a small amount of time at the end of each row and frame for image processing algorithms to reset and prepare for the next row or frame. The signal behaviour for the camera synchronisation is demonstrated in Figure 20.

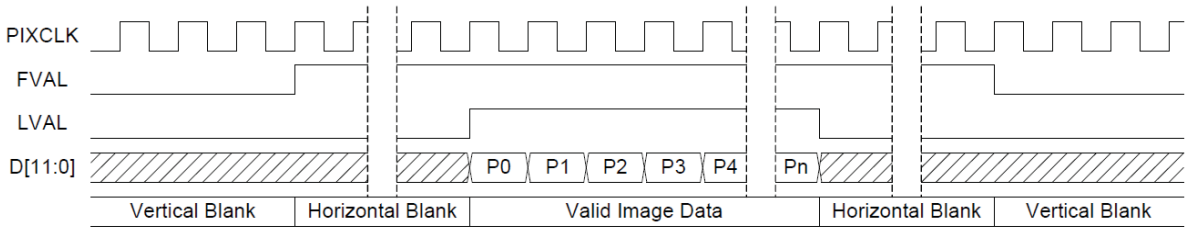


Figure 20 - Synchronisation signal behaviour (image courtesy of Terasic hardware specifications [42])

Lens Selection

Figure 21 shows an image captured using the camera and the standard lens with 4x skipping, and shows that the entire board cannot be fit within the image. This could be fixed by raising the height of the camera, or by using a wide-angle lens. To display the entire field, the camera would have to be at a height of 2.3 metres, which does not conform to the FIRA small league regulations [24].

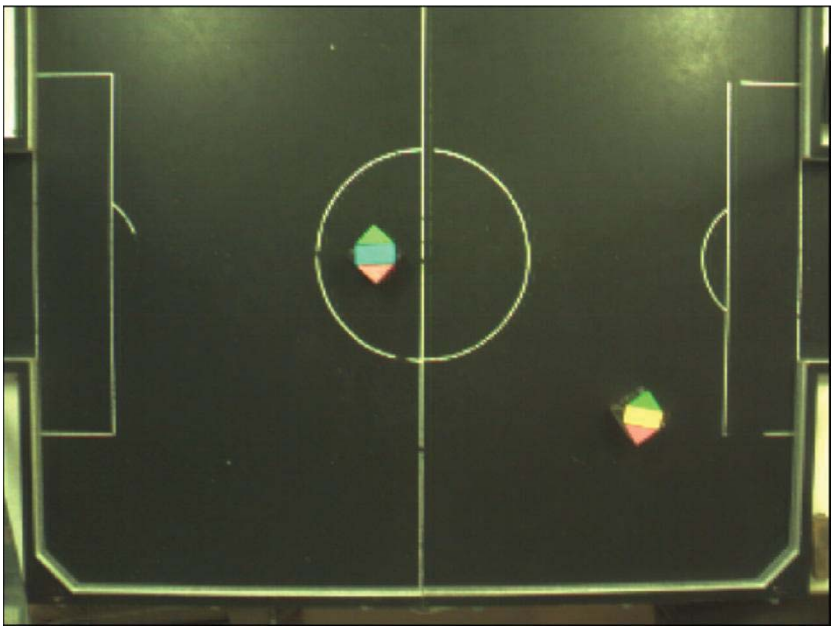


Figure 21 - Image captured using the TRDB-D5M using the standard lens

The 7.12mm lens that comes with the camera has a 55° field of view. Using a height of 2 metres and taking 2x skipping into account the replacement lens will require a focal length of 3.16mm or smaller. The DSL213A-670-F2.0 has a focal length of 3mm, a fixed aperture of 2 with a 170° field of vision, and is the closest size available from Sunex. However, this lens is designed to operate with a 1/3" sized image sensor, which is smaller than the D5M's image sensor. This means that the focused image will not fit the entire sensor and the edges of the sensor will not be illuminated. Since only the centre of the image is used, due to windowing, this should not affect the resulting image. Such short focal length lenses are prone to introducing barrel distortion. However, this should be somewhat minimised as only a small central part of the image sensor is being sampled, where the distortions are the smallest. The lower aperture of the new lens will allow more light through, resulting in shorter exposure times. This allows faster physical motions to be captured more clearly.

Utilising the new lens and 2x skipping the camera is capable of capturing the entire field. While the image does contain a small amount of barrel distortion, it is only minor and shouldn't affect the initial design of the robot soccer algorithm. However, in order to improve accuracy of the coordinate system, the barrel distortion will need to be corrected. An example image can be seen in Figure 22.

Figure 22 - Raw image captured from camera using DSL213A-670-F2.0 lens and 2x skipping

From scaling, each pixel represents roughly 3mm on the playing field.

3.6. Smart Camera Architecture

Before the image processing algorithm can be designed the smart camera architecture needs to be developed. The architecture controls the high-level functionality for how the camera operates. Before the image processing begins the camera must first read out the pixels and deliver them for processing. This requires the camera readout to be defined as well as a low-latency pipeline to the image processing modules.

Figure 23 shows the basic design for the proposed smart camera architecture

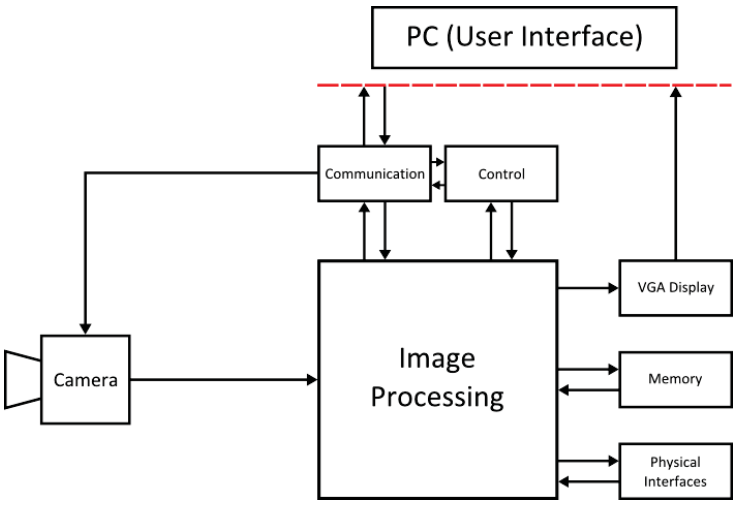


Figure 23 - Basic smart camera architecture

The camera, communication, and control modules make up the backbone of the smart camera architecture. The control module provides logic and instructions for the operation of the smart camera as a whole. The communication allows for instructions, information and results to be transferred to and from the FPGA and the user. These modules are unlikely to change significantly across different smart camera projects, as they provide the foundation for the smart camera. The image processing provides the real change in functionality. Depending on the image processing developed, the smart camera can change from an object tracking camera, to quality assurance checking, to security functions. This section will concentrate more on the design of the smart camera architecture; the image processing algorithm will be discussed in section 3.8.

Because the camera has the ability to provide 12 bits per pixel, it would be advantageous, at least during the initial design, to use all 12 bits. This allows the widest range of colour separation to be observed, and can help improve colour segmentation and noise removal, without affecting latency or frame rate. However, using only 8 bits per pixel enables frames to be loaded from, and saved to, a computer faster which decreases testing time. Even with reduced resolution, 8-bits is sufficient for verifying whether the algorithm is functioning correctly. In the final system implementation, 12-bits can be used to provide greater colour resolution which can improve colour classification performance.

A VGA display module will be needed to display frames on a display. Even though a smart camera can be operated without a display, it is usually best for early development to use a screen as this can shorten development time. A display allows a visual representation of the algorithm operation; this provides a debugging resource and in many cases this can help identify errors in the algorithm without the need to use test images.

In order to keep the camera output in sync with the image processing algorithms the FPGA should ideally operate at the same clock rate as the camera. However, the problem comes when trying to synchronise the FPGA to a VGA display. Low cost monitors can operate up to 75 to 80 frames per second, but this is significantly slower than the 120 frames per second that the camera is capable of at 640x480 resolution. When using a display, it may be necessary to lower the camera speed to match. However, this will not provide an accurate environment for capturing high speed images and testing low-latency algorithms. When developing the image processing algorithms, it is better to test in an environment that is as close to the final conditions as possible.

This creates a challenge, where the camera needs to operate as fast as possible, while still being able to synchronise with a VGA display. One approach is to split the architecture of the smart camera into two separate clock domains operating at different speeds. This would allow the camera to operate at a faster clock speed, while the VGA display can synchronise with the processed pixels.

However, with this design it is not intuitive which clock domain to implement the image processing. It could be implemented in the camera clock domain and be synchronised with the camera feed. However, this will require more resources to transfer the processed pixel to the display domain as each pixel is represented by three 12-bit channels. On the other hand, by

performing the image processing in the display domain only one 12-bit channel is needed to transfer the RAW pixels from the camera. Furthermore, the slower clock domain allows slightly more relaxed processing times. Therefore, during the algorithm development stage, it would be best to implement the image processing in the display domain.

Although, this poses a number of difficulties with synchronising the camera with the image processing.

- How to transfer pixels between clock domains
- How to stop pixels from being overwritten or lost
- How to keep both clock domains accurately synchronised

It can be difficult to implement stream processing when the pixels are read out from the camera at a faster rate than they are processed. In this situation, memory such as DRAM or FIFO is needed to buffer the pixel stream in order to avoid data loss.

An advantage to using a DRAM is the DE0 provides enough space to store a complete frame of 640x480 pixels with 12-bits per pixel. As well as buffering, this could also provide a method for storing images in order to transfer them to a computer for testing. In order to store and access DRAM, FIFOs are needed on the input and output to transfer a full row at once. This reduces the latency for accessing DRAM as much as possible. Figure 24 shows the DRAM buffer from the camera to the processing.

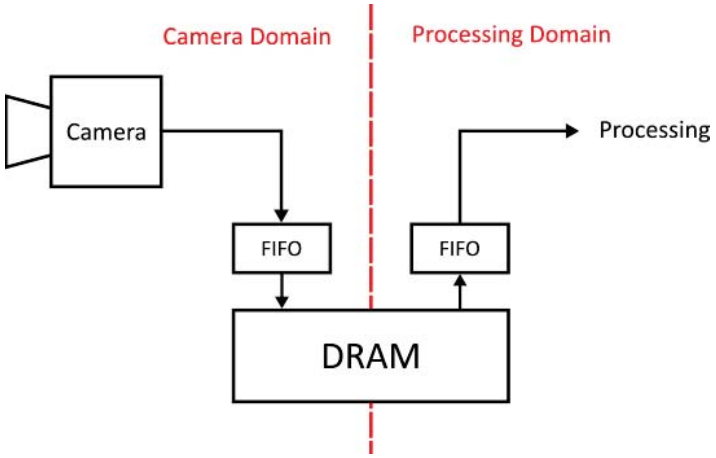


Figure 24 - Cross domain architecture using DRAM buffer

However, due to the complexity and variable latency introduced by utilising DRAM, this method is not ideal.

Another similar approach is outlined by Bailey [43] and Stevanovic et al. [44]. These papers propose high speed FPGA-based cameras that use FIFOs to transfer raw pixel data to the image processing algorithms. Following the same principle, the DRAM can be replaced with a single FIFO, shown in Figure 25. The FIFO allows for simultaneous read/write access, allowing for the FIFO to be written to and read from at different clock speeds. In order to reduce resources the FIFO can be used as a row buffer to store a single row at a time, instead of the whole image.

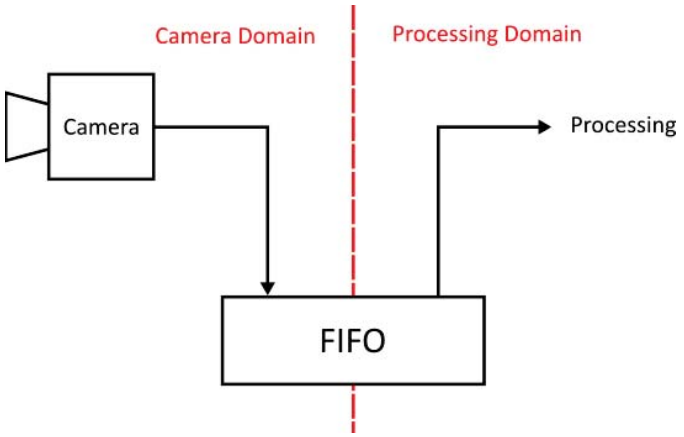


Figure 25 - Cross domain architecture using FIFO buffer

It should be noted that with this approach the information can only be read once from the FIFO before it is lost. While this is not a problem for stream processing, it can create some complications for transferring a complete image to the host computer for testing. Though, this can be solved by sending the pixels to DRAM, instead of the processing stream, when test images are required.

The FIFO approach uses fewer resources to transfer pixels from the camera domain to the display domain. Furthermore, it is simpler to implement and has a fixed low-latency transfer time compared to the DRAM method. This makes it the preferable method for passing pixels between clock domains.

Using the camera blanking periods at the end of each row, it is possible to synchronise the image processing with the camera readout so that no pixels are lost. In this way, the blanking periods act as a time buffer zone, allowing the image processing time to 'catch up' before the next row. However, this will require both clock domains to remain seamlessly synchronised with each other.

To keep both clock domains synchronised, it is best to derive them using a single phase-locked loop (PLL). The VGA screen that will be used has a maximum refresh rate (max frame rate) of 86 Hz. To be line synchronous, the total line time (including blanking) of the display (800) must match that of the camera (1550). This ratio 800:1550 = 16:31 requires running the camera at 68.78MHz. This corresponds to a pixel rate of 35.5MHz from the processing domain.

Using the PLL is a form of parameterisation, as it allows a simple means of adjusting the clock speeds in a centralised location without changing the functionality of the algorithm. During the final deployment of the smart camera the screen will be removed. The PLL can then be removed or adjusted in order to balance the two domains and increase the processing speed to the camera's maximum of 96 MHz, producing roughly 120 frames per second.

Figure 26 outlines the smart camera architecture that will be used to implement the smart camera.

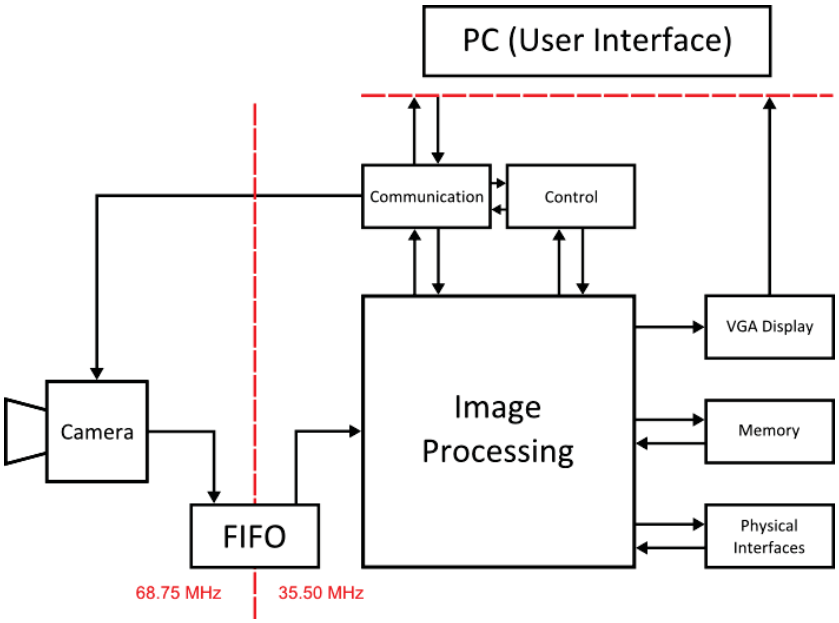


Figure 26 - The architecture for the robot soccer smart camera

3.7. Communication

While there are many possible options for communication between the host computer and smart camera, the RS-232 protocol is a fairly simple communication standard that is easy to implement in hardware. On the host computer, USB-to-Serial cables are common, and there are many computer programs available to handle the transmission and capture of serial information. However, the RS-232 standard has a comparatively slow transfer rate compared to other options, with a maximum useable baud rate of approximately 0.9Mbit/s. Nevertheless, while speed is important for transmitting images and data quickly for testing, in actual operation, it is only the robot and ball data that needs to be transmitted, which has significantly lower data volume. Therefore, the RS-232 standard will be used for the communication method, as it provides the lowest development time and a robust, well documented communication routine.

3.7.1. RS-232

Algorithm Development

To enable the designed RS-232 module to be used in other projects, and to be easily and quickly altered, the clock rate, baud rate, parity, end bits, etc. should be parameterised. Figure 27, shows the black box diagram for the serial driver.

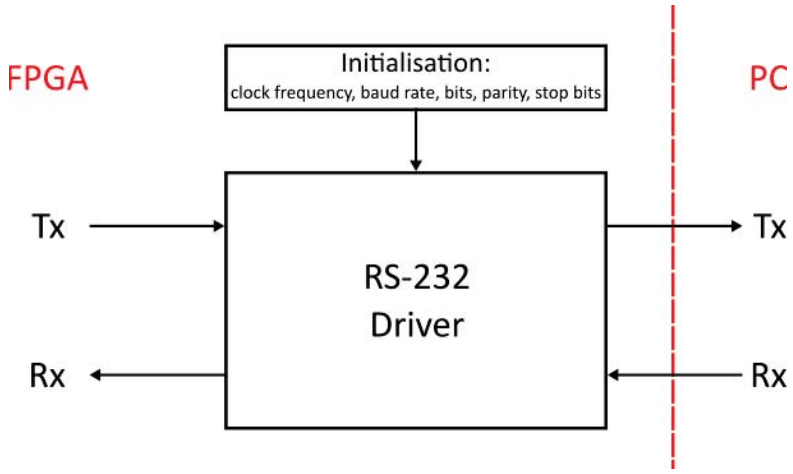


Figure 27 - RS-232 driver black box diagram

Since the clock frequency is significantly higher than the baud rate, dividing the clock down using a counter can give sufficient accuracy.

$$divider = clock\ frequency / baud\ rate \quad (1)$$

The baud rate can even be adjusted during run-time by changing the divider.

Testing

To test that the driver is working, a slow baud rate such as 9600 is used. Since the transmit and receive channels operate independently, they should be tested separately. Three specific test cases examine the performance and functionality:

- Transmission of a single data word, to test the basic operation of the driver, this will discover any fundamental problems with the timing or algorithm design.
- Transmission of a small string of data words, to test the capability of the driver to receive and transmit multiple words in quick succession.
- Transmission of a full 640x480 image, to test the transfer of bulk data, and ensure the process can stream a complete image without fault.

ASCII characters were chosen to represent the data bits for the first two tests because many computer-based serial communication programs can easily recognise and send ASCII values. These are also used for sending commands from the host to the FPGA.

The first test should be the FPGA transmit function to ensure the FPGA is able to correctly send data to the computer. Once FPGA transmission has been verified successfully, it can be used to debug the FPGA receive function. The FPGA is set to echo back all information that is received, and since transmission is known to be correct, any errors must be caused by the FPGA receive function.

Finally, when the RS-232 driver is completely functional the baud rate can be increased. Through experimentation it was found that 460800 is the maximum baud rate for reliable streaming. With this baud rate, it is possible to transfer the test image in approximately 15

seconds. Any speed above this baud rate causes random losses of data which corrupts the whole image.

Commands

The FPGA is programmed to accept multiple ASCII character commands. This allows for the FPGA to be operated through the serial port without the need to physically push buttons or switches. The commands that need to be implemented include: adjusting camera properties, initiating a captured image stream, or adjusting colour thresholds.

3.8. Algorithm Overview

To lower the design complexity, each image processing operation in the algorithm will be developed as a separate module. This allows smaller sections of the algorithm to be developed individually, and can make development easier by isolating errors for debugging. The modules can then be treated as black boxes, Figure 28, that will receive data from the previous module, process it, and pass it on. Each module will inevitably have a different latency, based on the complexity and characteristics of the process. Therefore, each data signal will have an associated synchronisation signal.

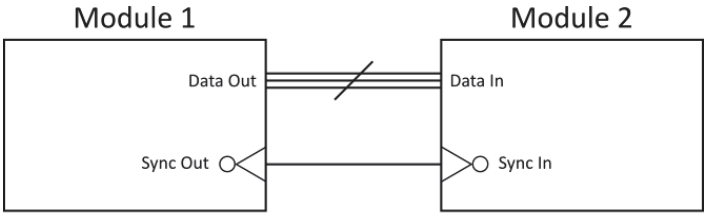


Figure 28 - Basic module connectivity

The main objective of the robot soccer algorithm is to calculate the position and orientation of each robot and the ball. To enable this, each robot has colour patches used to uniquely identify each robot, and to recognise the front. The ball is also a unique colour from the robot patches. Figure 29 shows an overview of the entire algorithm for achieving this goal. As shown in Figure 29, the algorithm can be split up into 3 stages: the pre-processing, feature extraction, and object processing stages. Pre-processing is designed to convert the raw pixel stream into a colour image that can be reliably classified. The feature extraction stage is then used to identify the correct colours and extract features from the components. These features are then used by the object processing stage to calculate the desired information about each robot, such as location, identity, and orientation.

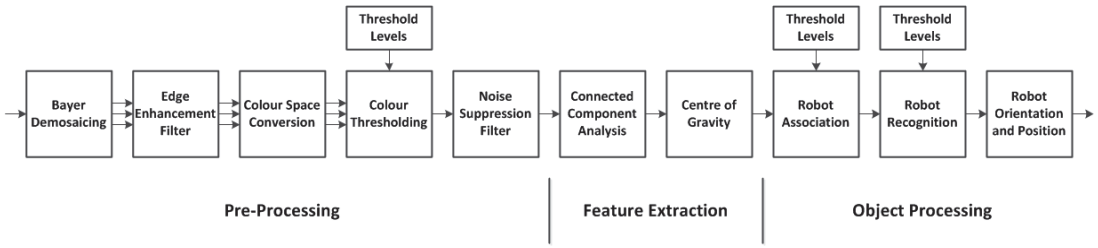


Figure 29 - Overview of the image processing algorithm

The first step is to convert the raw pixel data into a full colour image. A Bayer demosaicing algorithm is used to interpolate the raw pixels into RGB colour. However, demosaicing can introduce artefacts into the image such as blurriness, colour bleeding, and zipper effects. These artefacts affect the edges and corners of shapes the most, so an edge enhancement filter can be applied to sharpen the colour boundaries. The RGB colour space is not ideal for colour segmentation in situations with uneven or changing lighting conditions. The artificial lights on the robot soccer table create an uneven lighting pattern and the fluorescent lights flicker creating a changing lighting condition. Therefore, the RGB colour space will be transformed into a YUV-like colour space.

The feature extraction stage first thresholds the image into minimum and maximum thresholds for each colour channel. A morphological filter is used to remove further colour artefact noise in the form of single or small clusters of isolated thresholded pixels. This reduces the number of blobs that the connected component analysis algorithm will have to store and process, consequently reducing resources. Finally, the connected component analysis algorithm groups adjacent pixels of the same colour into component entities and calculates basic statistics for each of them. These statistics, such as the number of pixels present, sum of x and y, and sum of x^2 and y^2, are used to analyse the characteristics of the robots.

The final stage analyses the information from each individual component and groups closely spaced shapes into robots within the robot association module. Shape recognition is then applied to the team patch in order to recognise each robot's unique identity. Lastly, the angle between the centre of gravity of the triangles is used to calculate the robot's orientation. The coordinates of the ball and team patches centre of gravity is used to define the ball and robot's location.

Hardware based image processing systems are most efficient when processing high volumes of low-level repetitive routines. This makes them particularly good at applying filters and pixel-based transformations, as they can be applied in real-time directly on the pixels as they are streamed from the camera. However, some image processing operations are better applied in software. These are usually high-level, low volume routines, such as camera calibration or strategy calculation. In hardware, these would require considerable resources that would be underutilised. The object processing stage, shown in Figure 29, is on the boundary between these. The argument can be made that the sequential nature of the robot association module implies that it would be better implemented in software. However, these particular algorithms are not so resource intensive that warrants them to be implemented on the computer over the FPGA. Moreover, doing so would introduce extra latency and require large communication bandwidth in order to transfer all of the components and the extracted features for further processing. Therefore, while the argument can be made either way, it was decided to keep the processing on the FPGA. To minimise the latency and reduce the communication bandwidth.

3.9. Bayer Demosaicing Filter

A Bayer filter is required to convert the raw Bayer patterned pixels into an RGB colour image. Since each pixel only captures a single colour channel, a filter is necessary to interpolate the missing information; this process is called demosaicing. There are many different kinds of Bayer demosaicing filters, all with differing levels of 'quality' and complexity [45]. All demosaicing interpolations are only approximations of the original full colour scene. The most commonly introduced artefacts are blurring, colour bleeding, and zipper effect.

- Blurring – The reduced sampling of channels results in a loss of fine details. Recovering the missing details by interpolating across edges will blur those edges.
- Colour bleeding – The samples for the different channels are in different locations. Interpolating them independently can result in the different colour channels being interpolated differently, resulting in unnatural colours, particularly visible around edges.
- Zipper effect – The alternating rows and columns within the colour filter array pattern can result in an alternating colour pattern most visible around edges and lines. This alternating pattern looks a little like a zipper.

These artefacts are unavoidable, but can be reduced by implementing more complex demosaicing algorithms, with a trade-off between development time, resources used, and image quality. Also, the uneven spacing of skip mode introduces additional artefacts.

Algorithm Development

Each colour channel requires different calculations for each local context, determined from the phase of the Bayer pattern within the window. The notation is demonstrated in Figure 30.

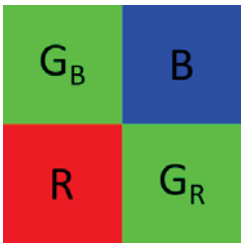


Figure 30 - Notation for pixel channels

The simplest Bayer interpolation simply copies the missing colours from the closest available location, known as the 'nearest neighbour' interpolation. This requires a 2x2 window. Although the resource count is small, it has significant artefacts, resulting in a low-quality output image.

Bailey et al. [46] analyse the quality of various Bayer demosaicing filters combined with skip mode. It was determined that a modified bilinear interpolation provided the best results out of the methods tested. The bilinear interpolation has low resource requirements and can be implemented using a synchronous window module. This makes it well suited for this case study.

A bilinear interpolation is another common method used for Bayer pattern demosaicing. While it is more complex than the nearest neighbour, it generally gives better results [45, 47]. The standard bilinear interpolation [45] (shown in Figure 31) utilises a 3x3 window to interpolate between two or more pixels around the output pixel '*P*'.

P_{-x-y}	P_{-y}	P_{+x-y}
P_{-x}	P	P_{+x}
P_{-x+y}	P_{+y}	P_{+x+y}

$$
\begin{aligned}
G_B: \quad & R = \left(P_{+y} + P_{-y}\right)/2 \\
& G = \left(P\right) \\
& B = \left(P_{+x} + P_{-x}\right)/2 \\
B: \quad & R = \left(P_{+x+y} + P_{+x-y} + P_{-x+y} + P_{-x-y}\right)/4 \\
& G = \left(P_{+x} + P_{-x} + P_{+y} + P_{-y}\right)/4 \\
& B = \left(P\right) \\
R: \quad & R = \left(P\right) \\
& G = \left(P_{+x} + P_{-x} + P_{+y} + P_{-y}\right)/4 \\
& B = \left(P_{+x+y} + P_{+x-y} + P_{-x+y} + P_{-x-y}\right)/4 \\
G_R: \quad & R = \left(P_{+x} + P_{-x}\right)/2 \\
& G = \left(P\right) \\
& B = \left(P_{+y} + P_{-y}\right)/2
\end{aligned}
\tag{2}
$$

Figure 31 - Standard 3x3 bilinear interpolation

With skip mode, the pixels are no longer equally spaced, so different weights must be used. Using Figure 32 to demonstrate the interpolation pattern, the skip weighting can be derived and demonstrated in Figure 33.

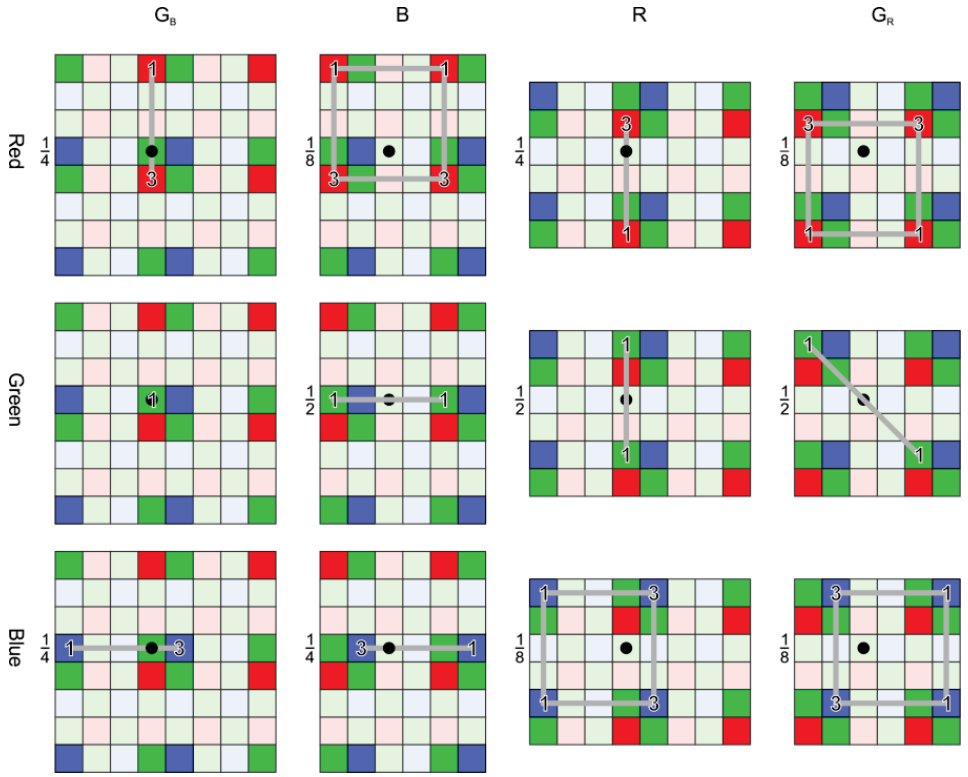


Figure 32 - Closest physical pixels for processing bilinear interpolation, with appropriate weightings by distance

While some distances can be estimated diagonally, the decision was made to use vertical and horizontal lines only in order to keep the weighting as powers of 2. This would make the equations more efficient to run on hardware implementations.

P_{-x-y}	P_{-y}	P_{+x-y}	P_{+2x-y}
P_{-x}	P	P_{+x}	P_{+2x}
P_{-x+y}	P_{+y}	P_{+x+y}	P_{+2x+y}
P_{-x+2y}	P_{+2y}	P_{+x+2y}	

$$
\begin{aligned}
G_B:\quad & R=\left(3P_{+y}+P_{-y}\right)/4 \\
& G=\left(P\right) \\
& B=\left(3P_{+x}+P_{-x}\right)/4 \\
B:\quad & R=\left(3\left(P_{+x+y}+P_{-x+y}\right)+P_{+x-y}++P_{-x-y}\right)/8 \\
& G=\left(P_{+x}+P_{-x}\right)/2 \\
& B=\left(3P+P_{+2x}\right)/4 \\
R:\quad & R=\left(3P+P_{+2y}\right)/4 \\
& G=\left(P_{+y}+P_{-y}\right)/2 \\
& B=\left(3\left(P_{+x+y}+P_{+x-y}\right)+P_{-x+y}+P_{-x-y}\right)/8 \\
G_R:\quad & R=\left(3\left(P_{+x}+P_{-x}\right)+P_{+x+2y}+P_{-x+2y}\right)/8 \\
& G=\left(P_{+x+y}+P_{-x-y}\right)/2 \\
& B=\left(3\left(P_{+y}+P_{-y}\right)+P_{+2x+y}+P_{+2x-y}\right)/8
\end{aligned}
\tag{3}
$$

Figure 33 - Proposed 4x4 bilinear interpolation using weighted equations to compensate for skipping

The equations described in Figure 33 are based on the output pixel '*P*' being located in the centre of an input pixel, Figure 34 (a). Bailey et al. [46] also considered a few different measurement schemes. Placing the output pixel on the centre of the 2x2 block Figure 34 (b) and the corner of the 2x2 block Figure 34 (c). Each scenario would give different weights, with varying complexity and results.

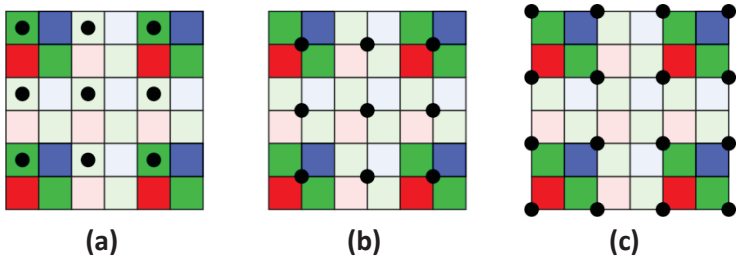


Figure 34 - Pixel weighting based on distance measured (a) pixel centred, (b) block centred, (c) block cornered

Bailey et al. [46] found that the block cornered method generally gave the best results in terms of PSNR. The block cornered method had the least amount of colour bleed and geometric distortion. The pixel centred approach, as described in Figure 33, was only slightly worse in terms of PSNR of the three methods. However, this approach does use the simplest algorithm that can easily be implemented on hardware. The other approaches would require more resources.

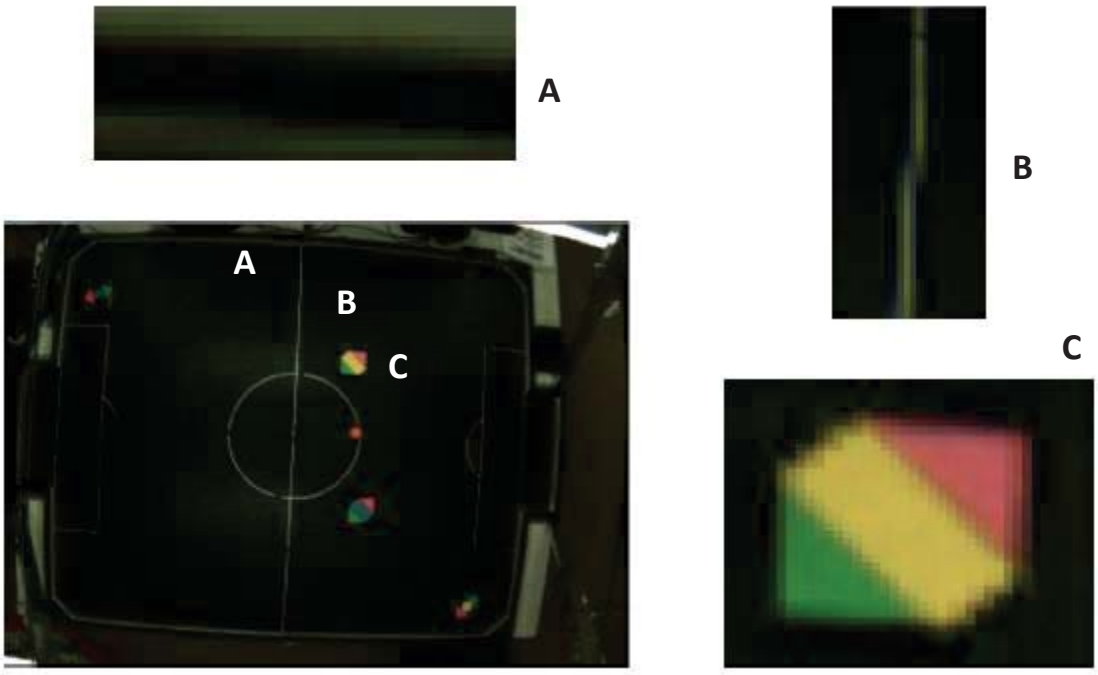


Figure 35 - Proposed bilinear test image with artefact examples

The bilinear interpolation, proposed by Figure 33, introduces a significant amount of blurring and colour bleeding to the image, demonstrated in Figure 35. This can mostly be fixed with an edge enhancement filter.

Of the many Bayer interpolation methods trialled, the bilinear interpolation method, outlined in Figure 33, provided the best result to resources ratio. The equations have been designed to conform to power of 2, making them more efficient to implement on hardware.

Hardware Realisation

The bilinear interpolation shown in Figure 33 uses a 4x4 window. This can be created using 3 row buffers, and an array of 16 registers. The individual calculations for each colour channel are relatively simple to implement within hardware, only requiring a single clock cycle per colour to calculate. There are two approaches for implementing the bilinear interpolation within hardware: the direct implementation, and the optimised design.

The direct implementation uses a separate circuit path for each colour channel and each phase of the Bayer pattern, shown in Figure 36. The multiplexors select the output depending on the phase, which can be determined from the least significant bit of the X and Y position of the window in the image. This method uses a total of 27 adders.

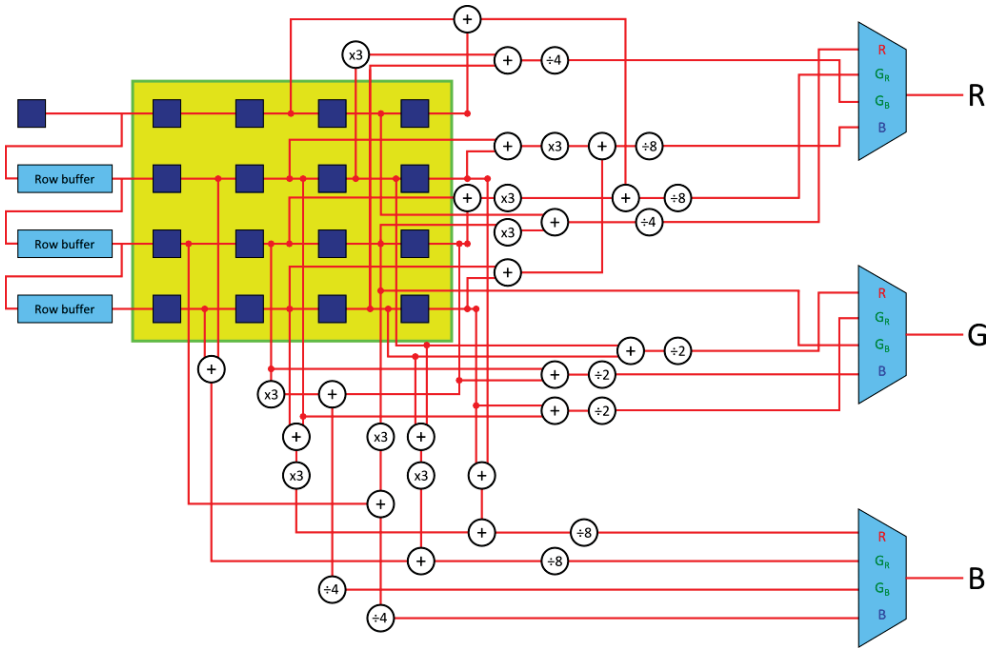


Figure 36 - Direct implementation block diagram for bilinear Bayer interpolation

The optimised design looks at exploiting common terms that can be reused between the parallel circuits, as shown in Figure 37. This method uses a total of 21 adders and an extra 6 registers to store intermediate data.

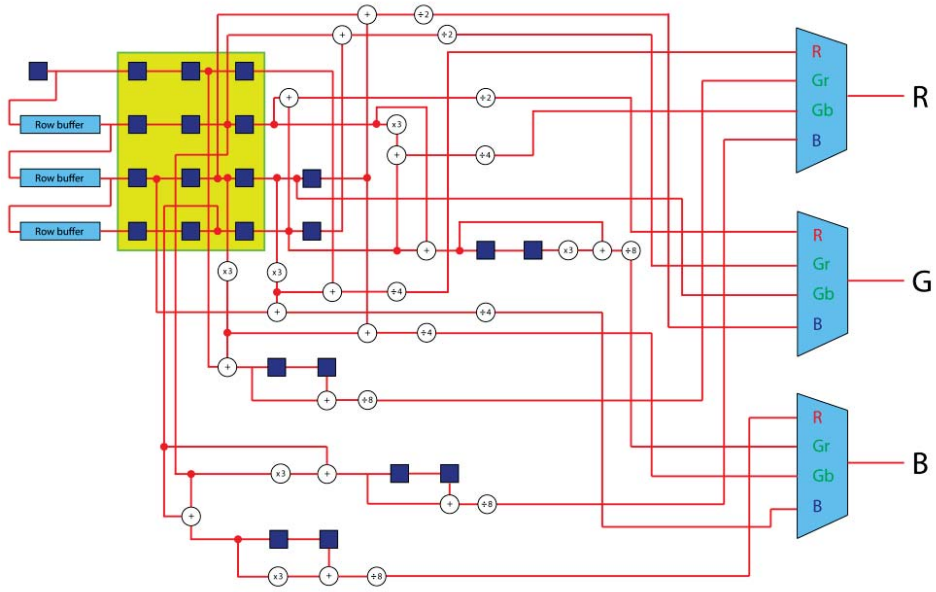


Figure 37 - Optimised design block diagram for bilinear Bayer interpolation

The optimised design reduces the required logic resources by 23% compared to the direct implementation. This makes the optimised design preferable for implementation.

3.10. Edge Enhancement Filter

Blurring around edges from Bayer interpolation can reduce the accuracy of image segmentation. With robot soccer, this loss of edge details reduces the accuracy of the shape recognition process. This blurring can be reduced by using an edge enhancement filter to sharpen the edges.

Algorithm Development

Gribbon et al. [48] describe a low complexity rank-based colour edge enhancement filter designed for an FPGA implementation. The basic concept of the filter, Figure 38, considers the horizontal and vertical differences, *A-B* and *C-D*. The largest difference indicates the orientation of the edge. If the pixels across the edge are sufficiently different, then the centre pixel *'X'* is considered to be on an edge. The centre pixel is then reassigned the value of the closest edge pair.

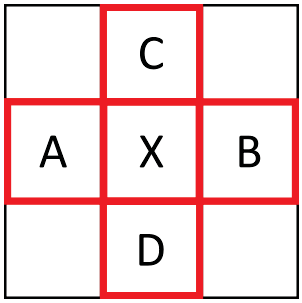


Figure 38 - 3x3 edge enhancement filter window

For a grey scale implementation, this filter would assign the centre pixel to the closest neighbour, based on the smallest absolute difference

$$f_{centre} = \begin{cases} Enhance\ (A,B) & \text{if} \quad \|A-B\| > \|C-D\| \\ Enhance\ (C,D) & \text{if} \quad \|A-B\| \le \|C-D\| \end{cases} \tag{4}$$

$$Enhance\ (P_1,P_2) = \begin{cases} P_1 & \text{if} \quad \left(\|P_2-X\| > \|P_1-X\|\right) \\ P_2 & \text{if} \quad \left(\|P_2-X\| \le \|P_1-X\|\right) \end{cases} \tag{5}$$

However, converting the algorithm to work for colour images across 3 colour channels is not obvious. Comer and Delp [49] propose reducing the components for each individual channel into a single scalar vector, thus creating a vector ranking algorithm. To detect an edge between vectors Gribbon et al. [48] suggests that a "colour median can be defined as the vector that minimises the sum of the distances to all other vectors".

$$x_{med} = \left\{ x_j \left| \min_j \sum_{i=1}^{N} \|x_j - x_i\| \right. \right\} \tag{6}$$

The median provides a measure for determining if the centre pixel is on an edge.

$$\|P_1 - P_2\| = |R_{P_1} - R_{P_2}| + |G_{P_1} - G_{P_2}| + |B_{P_1} - B_{P_2}| \tag{7}$$

$$Enhance\_Colour = \begin{cases} Enhance\ (X, P_1, P_2) & \text{if} \quad (\|P_1 - X\| < \|P_1 - P_2\|) OR (\|P_2 - X\| < \|P_1 - P_2\|) \\ X & \text{if} \quad Otherwise \end{cases} \tag{8}$$

$$Enhance\ (X, P_1, P_2) = \begin{cases} P_1 & \text{if} \quad (\|P_1 - X\| < \|P_2 - X\|)\ AND\ (\|P_1 - X\| < \|P_1 - P_2\|) \\ P_2 & \text{if} \quad (\|P_2 - X\| < \|P_1 - X\|)\ AND\ (\|P_2 - X\| < \|P_1 - P_2\|) \\ X & \text{if} \quad Otherwise \end{cases} \tag{9}$$

Then the L_1 norm is calculated to find the closest vector to the centre pixel for relabelling. This creates a relatively simple edge enhancement filter that can be implemented using only accumulators, subtractors, and low-level binary logic elements.

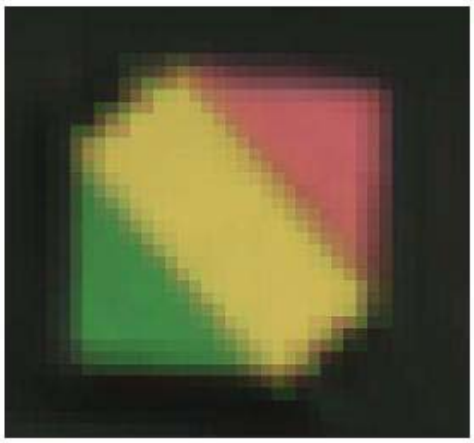


Figure 39 - Blurry image from the Bayer Interpolation module

Figure 39 shows a close up of one robot after the Bayer interpolation, where the edges gradually fade into each other (blurring). Figure 40 shows the result of applying the edge enhancement filter utilising a 3x3 window.

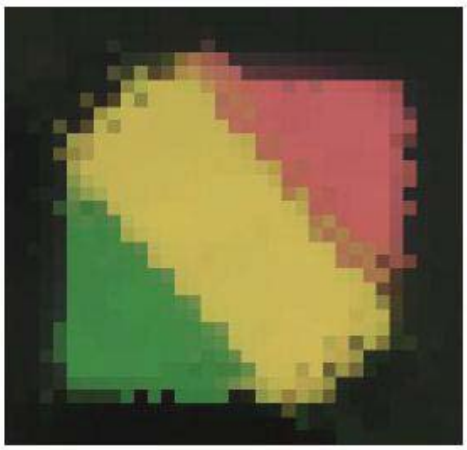


Figure 40 - Edge enhancement with a 3x3 window

Figure 40 shows some improvement with cleaning up the blurring between the colours, although there seems to be noise introduced away from the colour patches in the background. The process also seems to introduce stepping errors; this is most apparent in the space between two colours. While the reduced blurring may improve thresholding, the stepping errors and isolated pixels will detrimentally affect the shape recognition process. From Figure 39 the blur is 2 to 3 pixels wide. By increasing the window size, it should be able to improve the distinguishability of the edges. Figure 41 (a) shows the 5x5 window and Figure 41 (b) shows the results of implementing edge enhancement with this window.

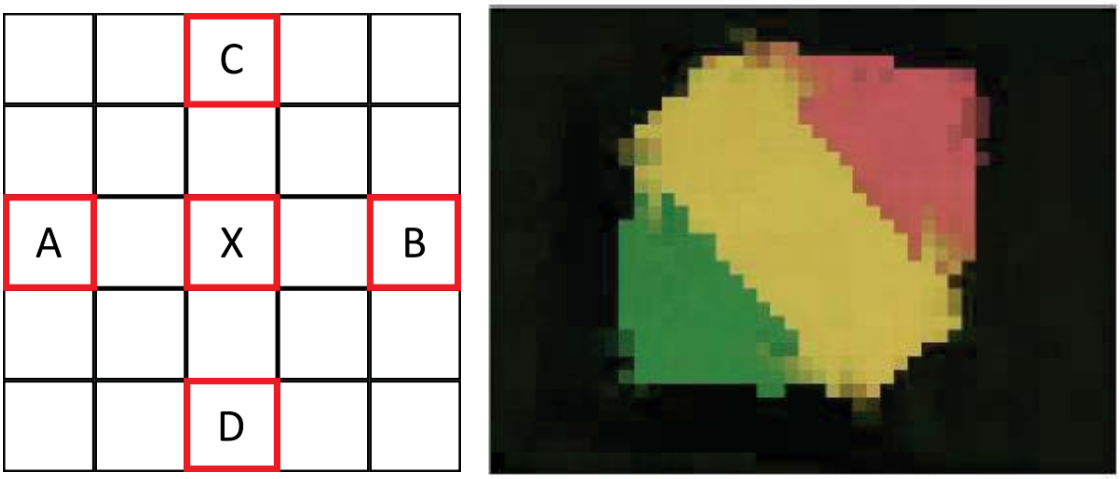


Figure 41 - (a) 5x5 filter window, (b) Edge enhancement with a 5x5 window

As expected increasing the size of the window does reduce the blur, except in sharp corners. While there is still a small amount of stepping present in the image, this is significantly better than with the smaller window.

Hardware Realisation

Implementing a 5x5 windowed filter within an FPGA will add two rows and three columns of latency. To efficiently implement this algorithm in hardware, some of the calculations will need to be reordered. In software, the median is used to first determine if a vertical or horizontal edge exists. Then depending on the answer the relevant vectors are calculated to find the minimum distance. Software optimises processing time by only calculating only what is needed. In hardware, all of the operations are performed in parallel, with the required result selected at the end, shown in Figure 42.

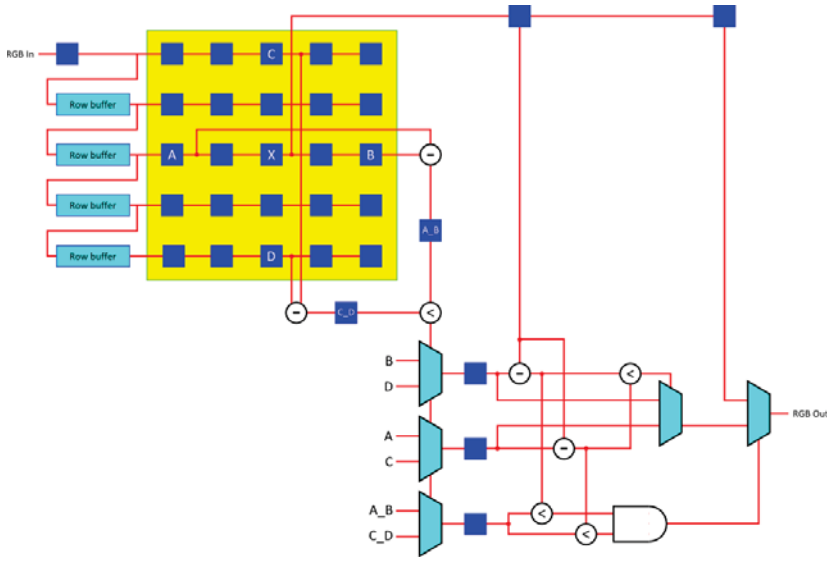


Figure 42 - Block Diagram for Edge Enhancement filter

The processing takes 2 clock cycles to complete, increasing the total latency of the module to two rows and five columns. In the first clock cycle, all of the vector differences are calculated, regardless of direction. Then in the second clock cycle the median and direction is calculated and the correct vectors are selected.

3.11. Colour Space Conversion

The RGB colour space is commonly used for displaying colour images. However, it is not ideally suited for image segmentation as the RGB channels are very susceptible to changes in light intensity. To counteract the limitations of the RGB colour space, it is necessary to first transform the image into a different colour space that separates the luminance and chrominance components [50-52]. Separation of the luminance and chrominance makes the colours easier to threshold in situations with uneven lighting patterns or fluctuating light levels [50]. The most common colour spaces used for image segmentation are either a variation of the YUV colour spaces (YUV, YCbCr, YPbPr) or a variation of HSV (HSI, HLS, HSL, HSB).

Converting RGB to YUV is relatively simple compared to the HSV colour space. Both RGB and YUV colour spaces use Cartesian coordinates, requiring a simple rotation and scaling. HSV on the other hand uses a polar coordinate system. For this reason, the initial tests will be implemented using a YUV-based colour space.

The YUV colour space is a distorted rotation of the RGB colour space, as shown by Figure 43. The scale factors are chosen to ensure that every RGB combination has a legal YUV representation.

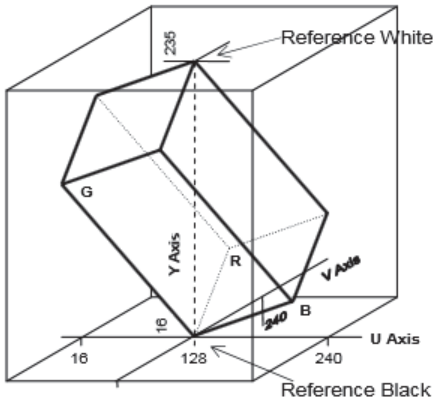


Figure 43 - Illustration of the YUV colour space transform (image courtesy of Sen Gupta et al. [36])

Algorithm Development

The standard conversion transform is shown in equation (10). This defines a true colour conversion where the visual colours between each colour space are equivalent.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{10}$$

The main limitation for this transform is that the factors are awkward to calculate. Sen Gupta et al. [36] propose a different weighting that is more suitable for hardware implementations, shown in equation (11). With machine vision, an evenly weighted colour space is not as important as mapping to a colour space that is easy to threshold. Since this colour space isn't strictly YUV it will be referred to as YC_1C_2, where *Y* still represents the luminance component and C_1 and C_2 represent the chrominance components.

$$\begin{bmatrix} Y \\ C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{11}$$

Equation (11) utilises power of 2 multiplications which are free in hardware, as they can be performed by shifting bits. This allows the algorithm to be performed in a single clock cycle within the FPGA.

Hardware Realisation

The module will need to consist of 3 inputs for the RGB channels and 3 outputs for the converted YC_1C_2 channels, shown in Figure 44. Since the transformation matrix is a point operation, a window design template is not necessary.

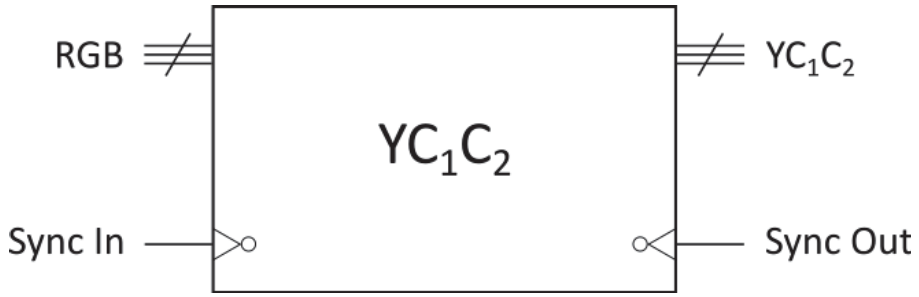


Figure 44 - YC_1C_2 module black box diagram

With factorisation, it is possible to complete the transformation in a single clock cycle using bit shifts and accumulators, shown in Figure 45.

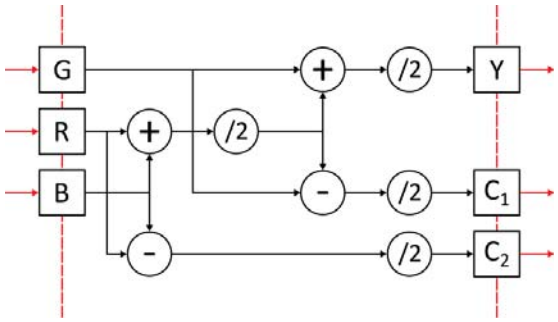


Figure 45 - Block diagram for the YC_1C_2 module

Testing is a simple matter as the same processes and procedures can be used that have been used for the Bayer and edge enhancement modules.

3.12. Colour Thresholding

The first step in detecting the robots is detecting the colours. With multiple different colours, each colour has an associated label, with 0 signifying the background. This case study will work with manually set threshold values. This removes the extra resources needed to operate filters and logic required to set the thresholds automatically. However, finding and selecting the optimum threshold values can be tedious and time-consuming. The simplest method is to define a set of minimum and maximum threshold values for each colour channel, requiring a total of 6 thresholds per colour. This effectively creates a box around each colour.

Every pixel that falls within a threshold box will be labelled as that colour. Special care should be used to not allow the different colour boxes to overlap; this would identify pixels as belonging to two different colours.

The threshold checking logic, shown in Figure 46, is fairly simple and is identical for each colour.

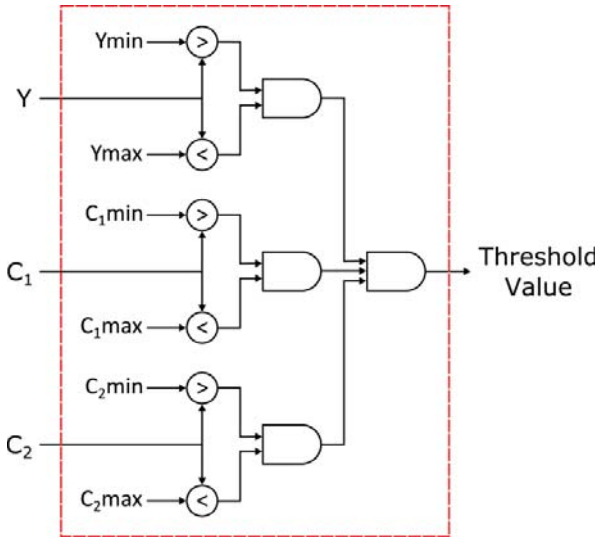


Figure 46 - Individual colour thresholding logic for YC_1C_2 thresholding

Figure 47 illustrates the block diagram for the entire colour threshold module, with each colour represented as a box which contains the logic shown in Figure 46.

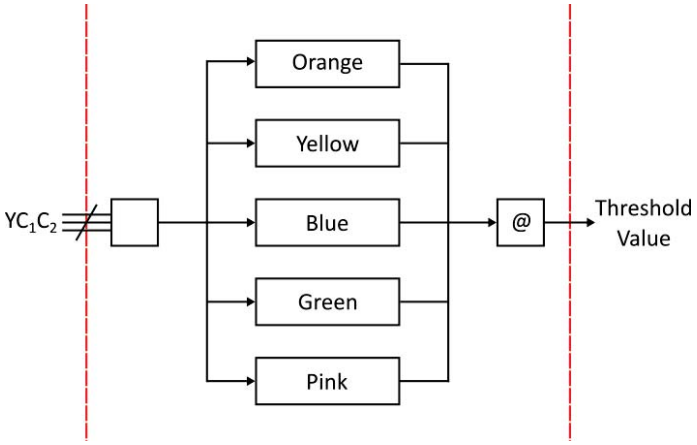


Figure 47 - Block diagram of colour thresholding module

Naturally, the first idea for labelling the separate colours would be using a sequential number order (1, 2, 3, 4, 5). While this would minimise the number of bits per pixel, it is prone to complications caused by overlapping errors. The pixel is checked in parallel by the logic for each colour. If two boxes were to overlap, the output label would be assigned multiple values at once, causing an unpredictable clash. To counteract this, a 5-bit value is used to represent the threshold value, with each bit representing a separate colour. Ideally the output label will have at most a single 1 in it (1, 2, 4, 8, 16). However, if an error does occur then the worst that will happen is that multiple ones will appear in the labelled number. This can be identified and handled further along the processing stream, and the user can be alerted that an error has occurred. This method does not add any significant amount of resources to the system.

During the first stage of FPGA testing (with 8-bit images) manually setting thresholds was very difficult. The YC_1C_2 values were clustered relatively close to each other, and threshold limits often clashed or overlapped, the results of which are shown in Figure 48 (a). Increasing the

resolution (from 8-bit to 12-bit) increased the overall resolution of the colour space. This increased the separation of the labelled pixels, making them easier to threshold. There was visual improvement to component shape with fewer misclassified pixels. Figure 48 (b) shows the comparison.

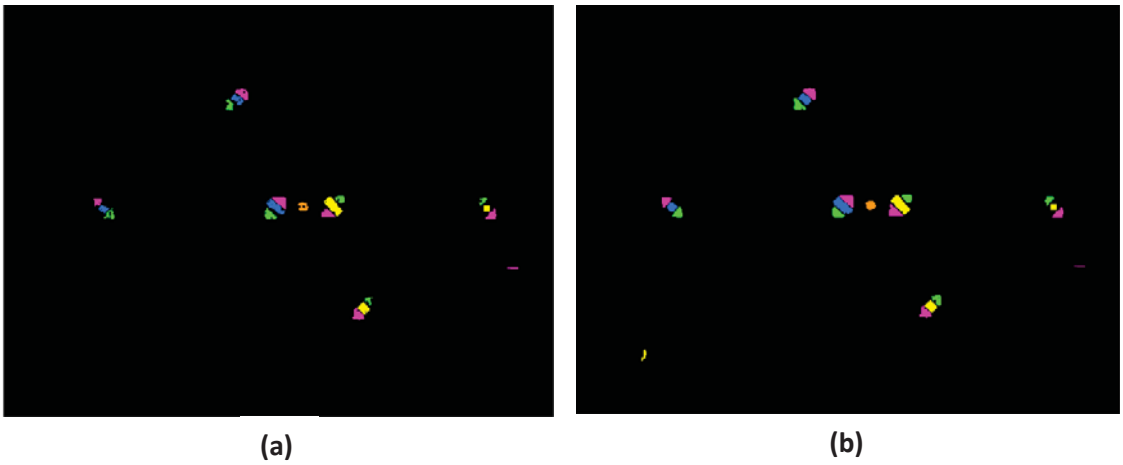


Figure 48 - (a) 8 bits per pixel labelled image, (b) 12 bits per pixel labelled image

This is an excellent example as to why parameterisation is so important. In this situation, if all the modules are properly parameterised then a single change can redefine the complete design. Allowing the changes to be made and tested within a few minutes. Without proper parameterisation, each module would have to be individually adjusted and retested to insure the changes did not affect the operation of each module. Manually changing every module takes much longer, and it is easy to introduce errors.

3.13. Noise Suppression Filter

Inevitably there is a lot of noise introduced from the misclassification of pixels. Figure 49 shows a closer inspection of a small region of the image, to give a better view of the noise pixels.

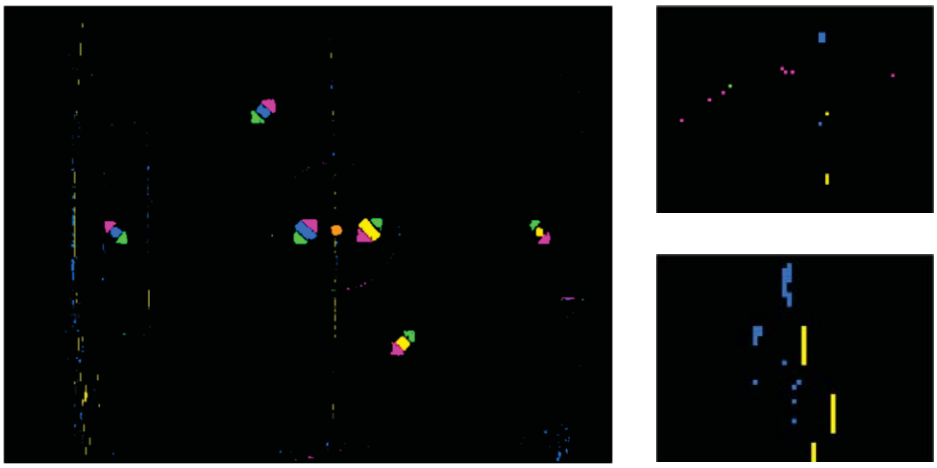
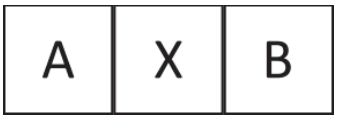


Figure 49 - Close up view of noise introduced during labelling

This amount of noise would not affect the operation of the connected component analysis algorithm. Each noise pixel will be classified as a separate component and processed accordingly. However, in a hardware implementation this would require a lot of memory to store these noise components as they are being processed. In very noisy images this could even fill up the allocated memory leading to overflow and loss of data. Since this noise is redundant information, it would be best to remove as much of it as possible in order to reduce the memory requirements for connected component analysis.

From Figure 49 it is apparent that much of the noise is made up of single pixels or small groups of pixels in a line. While adjusting the threshold values does remove some of the noise, it also significantly affects the shape of each component. Therefore, a morphological filter was developed to reduce the amount of noise in the picture. The basic idea of the filter is to first detect noise pixels, and then re-label them to conform to the area around them. Since much of the noise is only one pixel wide, a basic way of detecting noise would be to check if the centre pixel is surrounded by two of the same pixels of a different value. Since the direction of the noise is not horizontally or vertically biased, and appears seemingly random, this check will need to be performed on both vertical and horizontal pixel pairs. This can be done either using a two-pass unidirectional approach or using a single-pass multidirectional approach.

Multiple pass algorithms are generally not ideal in stream processing algorithms, because the image needs to be stored between each pass. In this case however, both unidirectional filters can be run sequentially as separate modules. Figure 50, shows the two windows needed to perform both the vertical and horizontal filters. Together both filters would introduce 3 rows and 3 columns of latency into the algorithm.



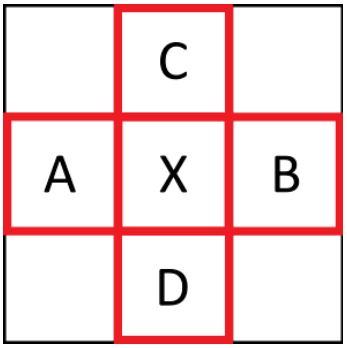
Vertical $$X = \begin{cases} A & \text{if} \quad A = B \\ X & \text{if} \quad Otherwise \end{cases} \tag{12}$$

Horizontal $$X = \begin{cases} C & \text{if} \quad C = D \\ X & \text{if} \quad Otherwise \end{cases} \tag{13}$$

Figure 50 - Horizontal and vertical unidirectional filter design for noise suppression

The multidirectional approach combines both the horizontal and vertical checks in a single clock cycle. As such, the window is shaped like a plus sign, as demonstrated in Figure 51. Compared with the two unidirectional filters, it does not add any extra latency to complete.

However, it does add a negligible amount of logic to handle the case when both horizontal and vertical tests are positive.



$$X = \begin{cases} A & \text{if} \quad A = B = C = D \\ A & \text{if} \quad A = B \\ C & \text{if} \quad C = D \\ X & \text{if} \quad Otherwise \end{cases} \tag{14}$$

Figure 51 - 3x3 multidirectional filter design for noise suppression

After testing each approach in software, both methods performed well and removed most of the noise. The results are shown in Figure 52. However, it was discovered that in some cases the unidirectional filters failed to remove some of the noise that the multidirectional filter could.

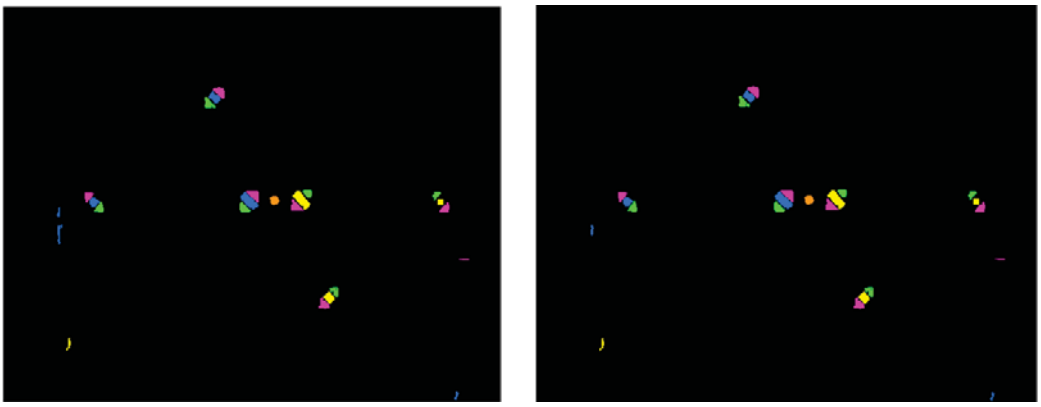
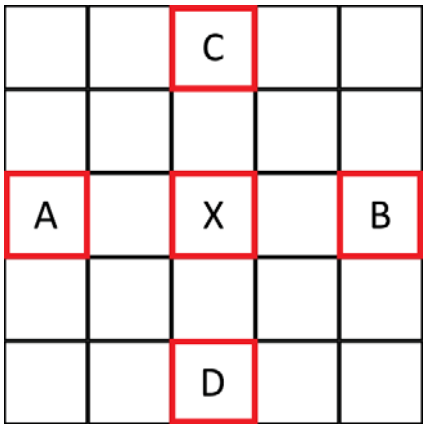


Figure 52 - (left) processed image using unidirectional filters, (right) processed image using multidirectional filter

Because the order of the unidirectional filters is fixed there is the potential for some noise to be missed. This could be fixed by running multiple passes; however, this adds unnecessary latency to the total algorithm. Conversely, the multidirectional algorithm allows for variable orientation checks. This allows the orientation to be changed in real-time, depending on the situation. For this reason, the multidirectional algorithm is the better option, as it returns the best results in the shortest latency.

Further testing of the multidirectional filter revealed that after the single pixel noise is removed a lot of the noise remaining consists of small groups of clustered pixels that are 2 pixels wide. This could be fixed by increasing the window size of the filter, shown in Figure 53.



$$X = \begin{cases} A & \text{if} \quad A = B = C = D \\ A & \text{if} \quad A = B \\ C & \text{if} \quad C = D \\ X & \text{if} \quad Otherwise \end{cases} \tag{15}$$

Figure 53 - 5x5 multidirectional filter design for noise suppression

Initial tests in software, shown in Figure 54, demonstrate that the increased window size removes more noise than the previous 3x3 window, leaving only much larger groups of pixels.

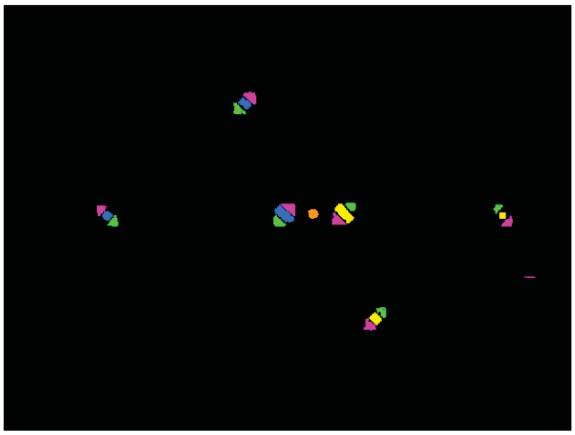


Figure 54 - Processed image using 5x5 multidirectional filter

It is possible to again increase the window size of the noise filter, in an attempt to remove these larger clusters. However, increasing the window also increases the latency of the filter, and the resources needed to implement the window. Increasing from 3x3 to 5x5 adds an extra 2 row buffers and 2 rows of latency. Since the connected components analysis module is able to eliminate small components below a thresholded size, it seems unnecessary to increase the filter to more than a 5x5 window. The function of the noise filter wasn't to eliminate 100% of the noise, but to reduce it as much as possible so that the connected component analysis algorithm does not get overloaded.

This module is a great example as to why the window processing method is so customizable. The size of the window is able to be increased or decreased without needing to alter the filter logic. Changing the window size also does not affect the synchronisation of the internal module timing. However, it does affect the latency of the module as a whole. For example, increasing the window from a 3x3 to a 5x5 will not change the rate at which the filter operates however it does add an extra 2 rows and 2 columns of latency to the overall algorithm.

An unexpected outcome of the noise suppression filter is the ability to clean up shapes. During testing it was discovered that the filter was also filling in shapes that had 'holes' inside of them, as well as further cleaning up of edges. This helps the connected component analysis module as it was not designed to handle concave shapes, or shapes with holes in them. Figure 55 (a) shows a close up of a robot with noise around the components, and Figure 55 (b) shows the same processed image after the noise filter. The edges have been cleared of the mislabelled pixels and the components have been further isolated from each other.

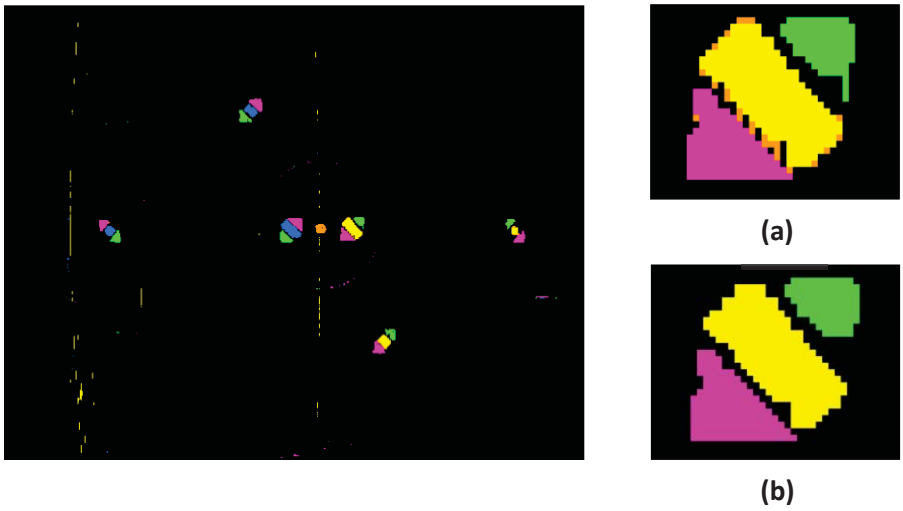


Figure 55 - (left) Test image, (a) close up of robot from test image, (b) close up of robot after noise filter

3.14. Connected Component Analysis

Connected component analysis, sometimes referred to as blob detection, is the process of identifying sets of connected pixels and assigning a unique label to the group. The groups of pixels can then be further processed to extract specific features or statistics. This process is required by the robot soccer algorithm to extract coordinate data about the robots and ball. These components, that represent the detected colour patches, will be used in later modules for shape recognition, robot identification, and determining the position and orientation of the robots and the ball.

There are many different types of connected component analyses ranging in complexity, number of passes, type of connectivity, etc. Commonly, software algorithms adopt a two-pass approach, similar to the method outlined by Rosenfeld and Pfaltz [53], an operational example is shown in Figure 56.

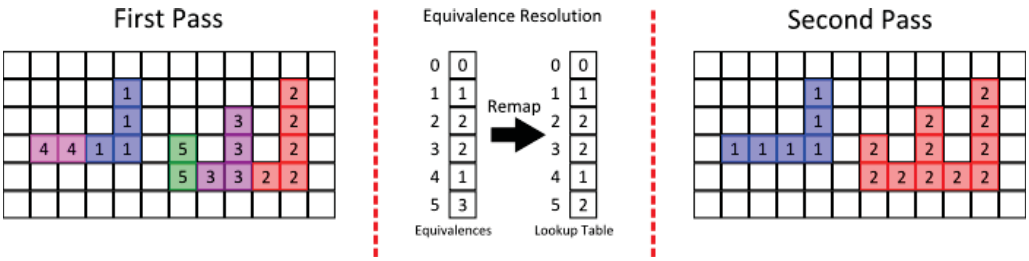


Figure 56 - Example of a common two-pass connected component labelling algorithm

The main difference between the different two-pass implementations is the technique used for resolving the equivalence table. Many techniques have been documented for increasing the efficiency of the equivalence table, from reducing the memory requirements to shortening the processing latency.

Due to the streaming and resource limitations for this case study a two-pass approach is not practical. This is because a full frame of partially labelled components would need to be stored between passes, which would require a lot of memory, and would introduce a full frame of latency.

Based on the design methodology of the smart camera, there are a few fundamental design requirements that this module has to meet; these include:

- Implement a robust and reliable single-pass approach using streamed data
- Calculate the features and statistics of the components in real-time
- Output finished components for further processing as soon as possible
- Utilise the smallest amount of FPGA resources possible
- Have the shortest latency possible

Due to the streaming environment of the camera algorithm, a single-pass algorithm [38] makes the most sense. This would have the lowest latency and would not require large amounts of memory to store frame information. However, the challenge for a single-pass design is how to resolve equivalences from concave or "U" shaped components.

Calculating the component features is another problem. Traditionally software approaches leave feature extraction to a separate region processing stage after the components have been labelled. However, for a stream processing approach it is necessary to calculate the features for each component in parallel with the connection analysis. This requires maintaining a feature data structure which is updated as each pixel is processed.

The last design goal is to output the finished components as soon as possible. The difficult part is detecting when a component is complete. This must be performed on a row-by-row basis to minimise the data storage required, and reduce the latency. Subsequent modules must therefore be able to manage the infrequent rate at which valid information is passed on.

Review of Connected Component Analysis Algorithms

The difficulty with designing a single-pass algorithm is how to handle mergers resulting from concave shapes (like "U") or shapes with 'holes' in them (like "O"). However, there are some methods that could be used or adapted for single-pass and streaming implementations, such as contour tracing [54, 55] and standalone single-pass algorithms [56-58].

Contour tracing is a method used to identify and label the boundaries of components. Contours can then be analysed to extract a range of shape features. The problem is that most contour tracing algorithms require random access to the complete image, making them unsuitable for stream processing.

Mandler and Oberlander [54] and Zingaretti et al. [55] both describe similar methods for creating chain coding. These proposed algorithms use a complete raster scanning approaches to scan the entire image, which could be adapted to operate with streamed processing. As with the contour tracing, this would allow for an accurate representation of the component shape, with the boundary containing all the feature information necessary. While these algorithms can be made to work for this case study, the complexity and storage requirements make them impractical.

Bailey and Johnston [56] developed a single-pass algorithm that has been adopted by other FPGA-based streaming implementations [57, 58]. The algorithm outlined by Bailey and Johnston works very similar to the first pass of the standard two-pass approach (outlined by Rosenfeld and Pfaltz [53]). However, since this is only a single-pass implementation, a different approach is needed to handle mergers and extract feature data. Rather than store an entire image, a row buffer is used to store the previous row. A merger table updates the output of the row buffer, in real-time, so when merges occur the current label is always available for propagation. The key to a single-pass algorithm is to collect component features during the first pass, and merging the feature data when mergers are detected.

Bailey and Johnston[56] outline a successful and practical connected component algorithm that can be implemented on an FPGA. However, because all the feature data was retained until the end of the image, it required memory resources proportional to the area of the image in the worst-case. Ni et al. [57] optimised the algorithm by reducing the total number of labels required, and outputting feature data as objects were completed. Memory requirements are now proportional to the width of the image rather than the area. Walczyk et al. [58] also documents improvements for the basic design set forth by Bailey and Johnston [56]. Walczyk et al. reduce resources by simplifying some of the control and merge logic. Instead of updating the row buffers with merged values in real-time like the original algorithm, merges are stored in a separate table and resolved at the end of the row. While the memory allocations may not show a great reduction in resources, in terms of storage capacity, in worst-case scenarios the authors put forward that fewer new labels will be assigned to seemingly disconnected components.

Algorithm Development

Bailey and Johnston's algorithm [56] is designed to handle any shape. While in general this is required by most conceivable implementations, it can be simplified considerably for robot soccer. Assumptions from the robot soccer system that can be used to simplify the algorithm are:

- All components are convex
- All components are solid with no holes inside them
- A small number of 'noise' components will be present and need to be removed
- Only a few components need to be identified (Most of the image is background)

Because all of the components are expected to be solid convex shapes, no mergers will be required. This removes the need for the merge table and logic. Since there is only a small

number of expected components, the data table that stores extracted information can be limited to a smaller size.

Figure 57 demonstrates the block diagram for the proposed connected component algorithm.

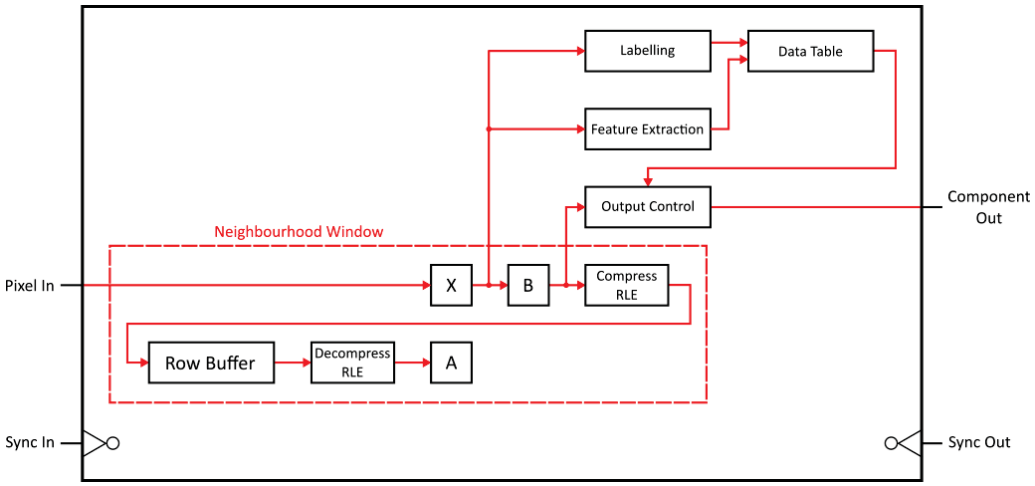


Figure 57 - Block diagram for connected component module

As demonstrated by Figure 57, the connected component module can be broken down into five separate sections:

- Neighbourhood Window – Controls the streaming of the pixels and control of the pixel window.
- Labelling – Controls the state transitions for the connectivity logic, applying labels, and controlling the module.
- Feature Extraction – Collects the relevant feature information about the current label run.
- Data Table – Stores the accumulated information for each identified component within the current frame.
- Output Control – Determines when a component is completed and synchronises the transmission of the completed component information from the data table to the next module.

The module is supplied with pixel information every clock cycle, in the form of colour labels, from the noise suppression filter. Therefore, this module must operate under strict one clock cycle timing requirements. However, the module will only transmit finished components, which means that there are no set timing requirements on the output. All modules after the connected component module must function asynchronously as each component is passed to them. Consequently, the synchronisation signal needs to be able to indicate the availability of asynchronous data.

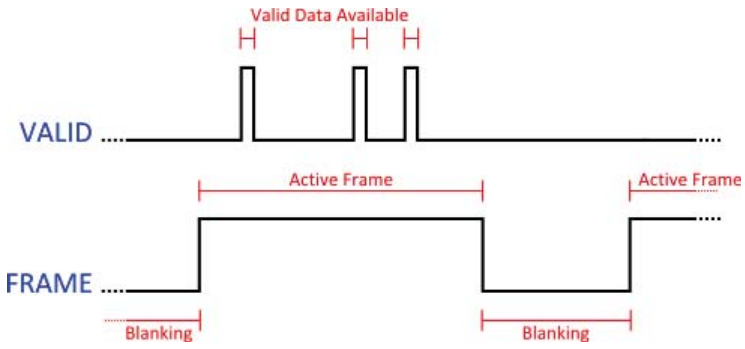


Figure 58 - Asynchronous synchronisation signal

Two indicators are required as shown in Figure 58. A '*VALID*' will go high for a single clock cycle when valid data is being output.

The second signal indicates the state of the active frame, denoted as '*FRAME*'. This signal goes high at the start of the frame, and remains high until all valid data is transmitted after the end of the frame. Downstream modules use this signal to determine when to commence end of frame processes, and reset for the next frame.

Neighbourhood Window

The input side the connected component module utilises a window, like many other modules before it. There are two approaches used to determine connectivity with previous pixels: 2-way, which considers only horizontal and vertical neighbours, and 4-way which also includes diagonal neighbours, as shown in Figure 59.

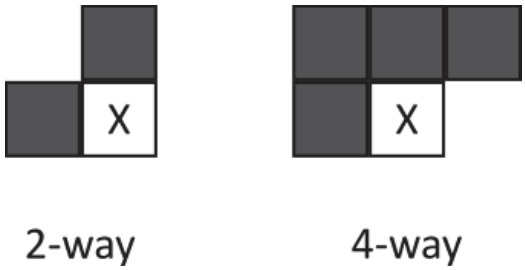


Figure 59 - 2-way and 4-way connectivity windows

Since purely diagonal connections, Figure 60, are unlikely to occur in normal operations, 2-way connectivity is used. This not only reduces the hardware resources, but also greatly simplifies the logic needed to determine connectivity.

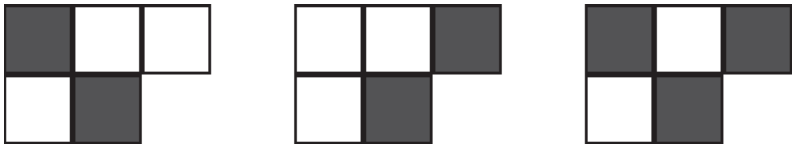


Figure 60 - Diagonal only connections in 4-way window

Figure 61 shows the design and window notation that will be implemented. Where *'A'* constitutes the pixel directly above *'X'* on the previous row. This position is provided by a row

buffer which holds one row of colour labels and associated component labels. It is important to keep the colour value during processing, as this is required to determine connectivity. *'B'* is the previously processed pixel on the same row.

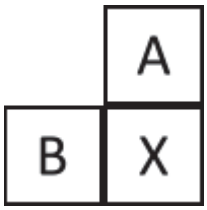


Figure 61 - Connected component analysis window notation

In order to further reduce memory for storing previous row information, run length encoding (RLE) can be used. RLE is simple to implement and can operate in real-time with minimal resource overhead.

Labelling

The labelling section controls both the assignment of labels to the pixels and the extraction of feature information. There are four different conditions based on the relationship between the current pixel and previous pixel. These can be illustrated using a state transition diagram, shown in Figure 62.

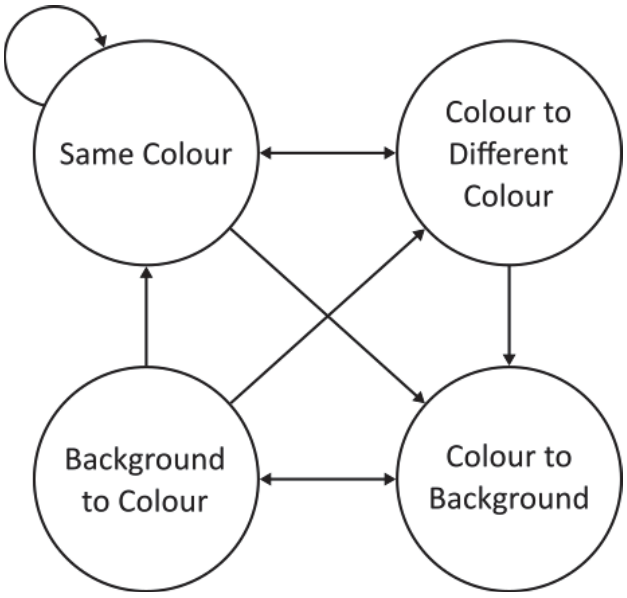


Figure 62 - 4-state transition diagram for connectivity logic

The functions within each state can be summarised as follows:

- Same colour – Assign the current pixel the same label as the previous pixel '*B*', and accumulate the feature extraction data (if it is a valid colour). If it is the background, assign the background label.
- Background to colour – Assign a new label to the current pixel. Record the first iteration of the feature extraction.

- Colour to background – Send the extracted feature data to the data table. Then label the current pixel as background.
- Colour to different colour – Send the extracted feature data to the data table to be accumulated with the current colour label. Assign a new label to the current pixel. Record the first iteration of the feature extraction.

Closer inspection reveals that each function deals with one of two scenarios. Either the function handles the continuation of the current run (same colour) or handles a transition from one run to another (the remaining three functions). This allows the control logic to be simplified to 2 states, as shown in Figure 63, allowing the resources to be significantly reduced.

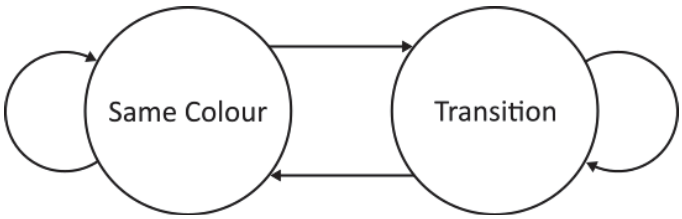


Figure 63 - Simplified 2-state transition diagram for connectivity logic

The new functions can now be defined as:

- Same colour – Add the current pixel to the existing run of labels, and update the features.
- Transition – Save the current runs features to memory. Update the RLE with the previous run. Start a new run, and update the features for the current pixel.

Initially it was assumed that all shapes will be convex, and therefore there should be no need to resolve merging conflicts because they shouldn't exist. In reality, small errors in Bayer demosaicing and segmentation can cause small faults resulting in concave shapes. Figure 64 illustrates an example where a small noise point causes a merging fault in a one-pass solution. Conceptually, this splits the single component into two separate components.

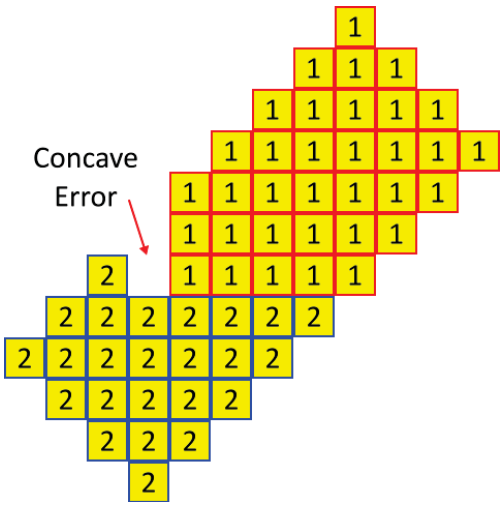


Figure 64 - Example of a concave merging error in a single-pass solution

This can be solved by deferring label assignment until the run is saved to memory. If a run encounters multiple labels, it makes sense to use the label with the largest area. However, this requires an additional memory access which will delay assignment, and cause synchronisation conflicts. It is not unreasonable to assume that the label with the most connections with the current run would also have the highest sum of pixels. Therefore, by collecting statistics on the number of connections made by individual labels, the label with the highest number of connections is applied to the current run. Figure 65 shows the results from this technique.

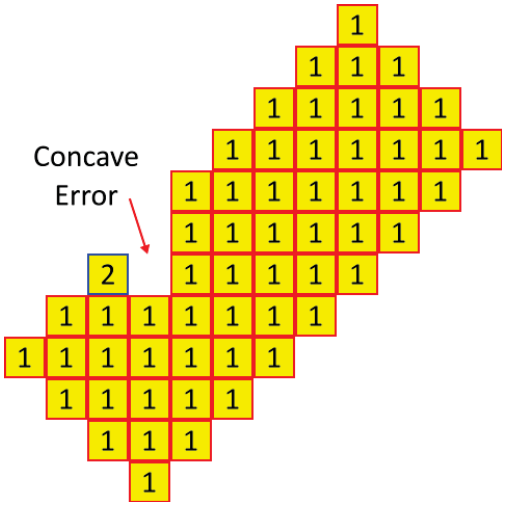


Figure 65 - Results for statistically gathered label assignment

This method is capable of processing convex shapes and concave shapes to a limited extent. The remaining parts of the component that were given a different label will simply be removed from the main component. This may cause a small error when calculating the shape or centre of gravity, but assuming the concave error is relatively small this should have little consequence overall.

Feature Extraction

The final features for each component are the accumulation of the features extracted from each individual run within the component. This process operates in parallel with the labelling process in order to accumulate the features for the current run. As the features are being accumulated, the values are stored in a register. This allows for simultaneous reads and writes in the same clock cycle. Once the run is finished, the features are then transferred and accumulated with the values in memory.

Because of the single-pass system, and the strict timing requirements, there is a limit to what types of features can be extracted. The features are restricted to ones that can be gathered on a pixel by pixel basis. The features that will be extracted are:

- Area
- Sum of X and Sum of Y
- Sum of X^2 and Sum of Y^2
- Perimeter

The area can be calculated relatively easily for a binary image by totalling the number of pixels present for each run, and accumulating that with the totals from previous runs. This is needed for the centre of gravity module and the recognition module.

The sum of *X* and sum of *Y* are gathered from the column and row positions respectively. This is needed for the centre of gravity module, to determine the robot's positions within the frame.

Sum of X^2 and sum of Y^2 are needed for the robot recognition module to test second moment calculations. These values can be accumulated in a similar fashion to the sum of *X* and sum of *Y*. The only difference is that each column or row value is first multiplied by itself before being accumulated with the rest of the run.

Calculating the perimeter for binary images is quite complex [59, 60]. The simplest approach would be to count the number of pixels on the edge of the object. This would be relatively simple to implement on the FPGA, by accumulating the sum of pixels that are found to be in contact with the background within each run. However, this method tends to under-estimate the actual perimeter length because it doesn't take into account the extra distance added by diagonal lines [59]. This could be compensated for by adding an additional weight to diagonal pixels, at the cost of slightly more algorithm complexity. Calculating the perimeter is needed as a part of the robot recognition module, therefore specific results comparing the two methods will be examined there. However, in order to provide the most information for development, both the weighted and unweighted perimeter values are accumulated for further testing.

Data Table

In order to store the large amount of feature data extracted for each component the FPGAs memory will be needed. However, the main limitation is that memory can only be read or written to once within the same clock cycle. Therefore, at least 2 clock cycles will be needed to read the previous component data, update it, and write it back. This is further complicated by the introduction of the output control needing to also read from memory in order to transfer the finished component, adding in further read/write conflicts.

By using a dual port read/write system, with a separate read and write port, the save process could be pipelined to read during the first clock cycle, and then write back during the second clock cycle, as demonstrated by Figure 66. However, with only a single read port, there is a high possibility that the output control and saving process will access the read port at the same time, negating one of the actions.

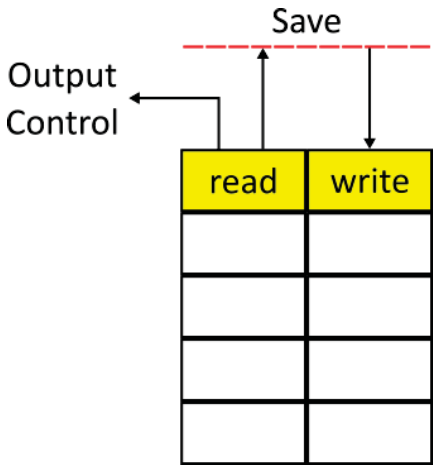


Figure 66 - Dual port read/write data table

Another option would be to use a dual port RAM system, where both ports have read/write access. One of the RAM ports is reserved for the saving pipeline, and the second for the output control access, Figure 67. This would allow the output control dedicated access using a port that will not conflict with the saving process.

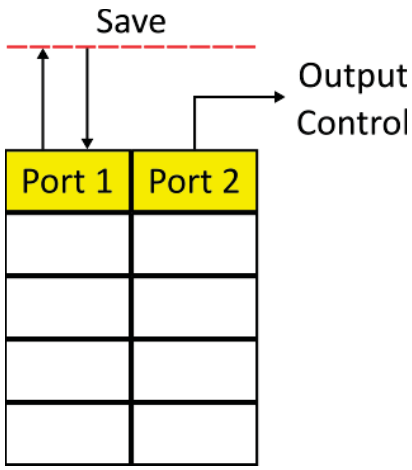


Figure 67 - Dual port RAM data table with dedicated ports

The dual RAM port method is preferable over the dual port read/write technique because it requires less logic and complex synchronisation to handle memory access conflicts.

Output Control

The output control is responsible for detecting when a component is finished, and transferring the data to the next module. The difficult part is detecting when a component is completed. To keep latency to a minimum the components should be output as soon as they are completed. The simplest way to determine if a component is complete is the absence of connected pixels in the following row.

One method of achieving this would be to keep a record of every component label encountered in both the current and previous row. Any label that is present in the previous row, but not in the current row, will have no more connecting pixels and would therefore be

completed. At the end of each row, the two records would need to be compared, and any labels matching this criterion can be labelled as complete. The disadvantage of this method is the difficulty and amount of time needed to compare the two records. Depending on how many components are encountered on each row, the size and processing time for comparing each record will change. Secondly, arrays would be needed to store the records, requiring a significant amount of resources, making this method unfeasible.

A more suitable approach is to keep track of the current and previous labels and pass on the completed components in real-time. This can be achieved by inspecting the labels from the previous row as they are decoded by the run length encoding. If a transition from component label to background, or to another colour, is detected and the same label hasn't been identified on the current row, then the component is finished. This is illustrated in Figure 68.

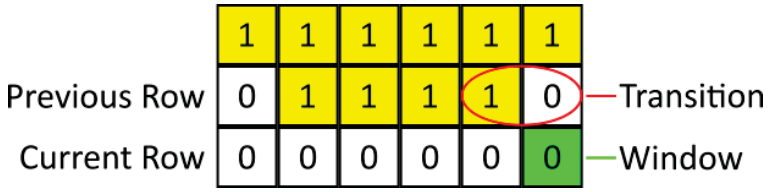


Figure 68 - Real-time finished component detection example

This method allows the component to be detected within the row, as opposed to at the end of the row. This has the fastest detection rate possible when using a raster scan approach without prior knowledge, accumulating only 1 row of latency in a worst-case scenario. It also reduces the need for expensive arrays for storing data, as all completed components are detected and transferred in real-time.

During the pre-processing and colour segmentation stage, all possible care is taken to remove as much noise as possible. Despite this, some noise will still get through to the connected component analysis module, where it manifests as small components. Figure 69 demonstrates a small amount of noise, in the form of isolated yellow lines, caused by colour bleeding of the white lines on the field.

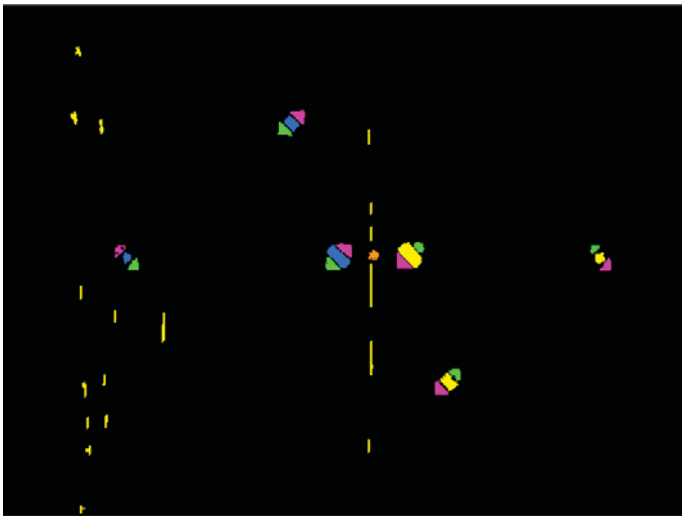


Figure 69 - Small components (noise) within connected component algorithm

Due to the limited area of the noise components, it is possible to filter them out. Before a component is transmitted to the next module, components with an area below a minimum threshold are discarded. This reduces much of the remaining noise, and goes a long way to ensuring that the later modules are only processing correct robot components. Figure 70 shows the results of the connected component algorithm, each component is represented by a separate colour signifying its colour group.

Figure 70 - Results from the connected component analysis algorithm

Output Destination

The pre-processing and feature extraction stages must be carried out in series, as the data is reliant on the output from the previous module. Due to the limited resources and strict timing requirements, saving an entire image is not feasible. However, this is not necessarily the case with the object processing stage, which can potentially operate in parallel. As such, the data would need to be accessible by all modules at once; this can be done by passing the data individually to each module or through globally accessible structures. The robot data would consist of the components centre of gravity, sum of pixels, sum of X^2 and Y^2, as well as the robots team identification, orientation, and coordinates within the frame. This data can be represented by 270 bits per robot, with a total of 1724 bits needed for 6 robots and the ball. To implement a globally accessible array, logic units would need to be configured to store data. The DE0 Development board has a total of 15,408 logic units, this means storing the robots and the ball in an array would utilise 11% of the FPGAs total logic resources.

While 11% may seem like an excessive amount, there are some advantages to storing the robot information globally. The FIFO buffer module design, illustrated in Figure 9, requires a section of memory to be allocated to store incoming information in order to avoid potential data loss. However, a globally accessible array would allow for a pipeline module to be used, removing the need for extra memory. This reduces the memory used by each module, at the expense of reserving a fixed amount of logic for the array.

On the other hand, processing each robot in a streamed manner, using FIFO buffer modules, would save logic units at the expense of added memory for each module. The DE0 development board has roughly 30x more memory than logic units, and all of the object processing modules combined would use approximately 3% of the DE0's memory. While it is

true that operating the object processing modules in series adds additional latency, in this example the extra latency is negligible. The recognition module adds 1 clock cycle, and the orientation module adds an additional 7 clock cycles. Compared to the parallel method this increases the latency by 3 clock cycles, or 81nS.

Logic units are required to implement the modules and perform the image processing operations. Limiting the number of available logic units limits the resources available to implement the image processing. Because the DE0 has significantly more memory than logic units it makes little sense to conserver memory at the cost of reduced image processing capabilities. Due to the inefficient use of resources, and negligible latency saved, the algorithm will continue to process in series, with the object processing stage using FIFO buffer modules where required.

3.15. Centre of Gravity

The centre of gravity (COG) module is used to find the location of each component. This is needed to help group the different component types into their respective robot, as well as identify the coordinate position for each robot and the ball. The centre of gravity (COG) is given by

$$COG = \left(\frac{S_x}{Area}, \frac{S_y}{Area} \right) \tag{16}$$

Where 'S_x' and 'S_y' are the sums of the columns and rows of the component respectively, and the area is the total size of the component [12]. However, the rounding caused by integer arithmetic and binary representation can cause the centre of gravity to be inaccurately rounded.

A one pixel offset corresponds to a worst-case offset of 3.2mm. With such a small error within a rapidly changing environment, such as robot soccer, it was determined that the binary rounding would not overly affect the results.

Hardware Realisation

The component stream comprises of a structure of elements. The simplicity of the equations allows the results to be calculated in a single clock cycle, and saved directly to the output stream, as shown by Figure 71.

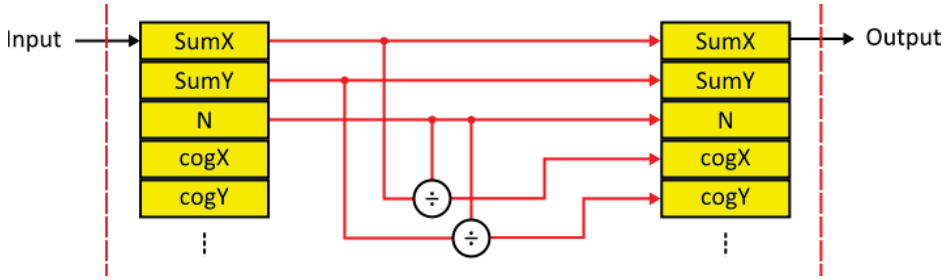


Figure 71 - Block diagram for centre of gravity module

Where '*SumX*' and '*SumY*' are 'S_x' and 'S_y' respectively, and '*N*' is the area. '*cogX*' and '*cogY*' are the X and Y coordinates for the calculated centre of gravity.

3.16. Robot Association

The robot association module takes the individual components and combines them together into complete robots. A robot consists of three separate components, a pink and green triangle (used to determine orientation) and a team patch (used to recognise the team and robot identity). Figure 72 shows the overhead view of the colour patch as it is seen from the camera.

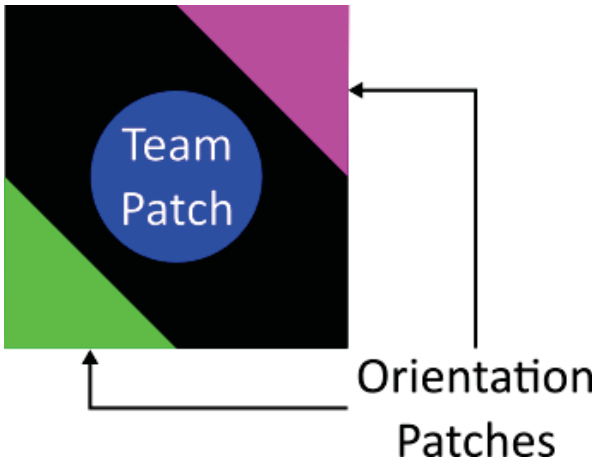


Figure 72 - Robot colour patch layout

The problem is to determine which components belong to each robot. Since the robots are constantly moving, any position and orientation is possible, therefore there is no clear order in which completed components will enter this module. Consequently, robots cannot be associated purely on a first come first serve basis.

Algorithm Development

The most basic description of what constitutes a robot is the three colour patches in close proximity to each other. A simple solution would be to determine the distance between each patch as they are discovered and combine them if they are sufficiently close together. However, in robot soccer it's not uncommon for multiple robots to be in close proximity to each other, which poses more difficulty for distinguishing different sets of very close components.

The positional information available is the components centre of gravity (COG) coordinates. Since every robot has a team patch in the middle, an orientation patch (pink or green) is associated with a robot if its COG is within a given distance of the team patch. The simplest distance measure to calculate is

$$D = \max\left(|x_1 - x_2|, |y_1 - y_2|\right) \tag{17}$$

Where '*D*' is the distance in 'pixels', and the coordinates of the two points are represented by (x_1, y_1) and (x_2, y_2). Unfortunately, this is not orientation independent.

Figure 73 shows the worst-case scenario, where two robots are next to each other, with one triangle close in distance to the other robot's team patch.

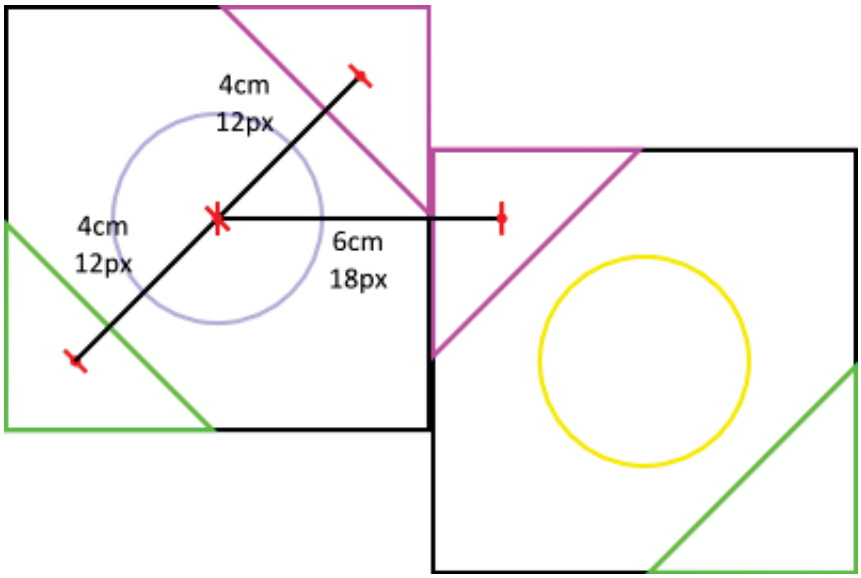


Figure 73 - Worst-case scenario for robot association

In practical situations, with the inaccuracies caused by the image processing included, it is very likely that this method could cause incorrect associations. In reality, this worst-case scenario can happen quite often, as the robots do bump into each other regularly.

The alternative is to use Euclidean distance:

$$D = \sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2} \tag{18}$$

Testing the validity of this theory is relatively simple in software. A few test images are collected that demonstrate various ideal and worst-case scenarios. In order for this theory to be confirmed, a single threshold value would need to be found that can be used across multiple images to distinguish individual robots with a high-level of reliability. Figure 74 shows 4 images, labelled a, b, c, d that demonstrate the various possible scenarios, ranging from ideal (a, b) to worst-case (c, d). The distance, calculated using equation (18), from the team patch to the correct orientation patches are shown in Table 1. Table 2 shows the calculated distance from the team patch to adjacent orientation patches from other robots, testing the worst-case scenario.

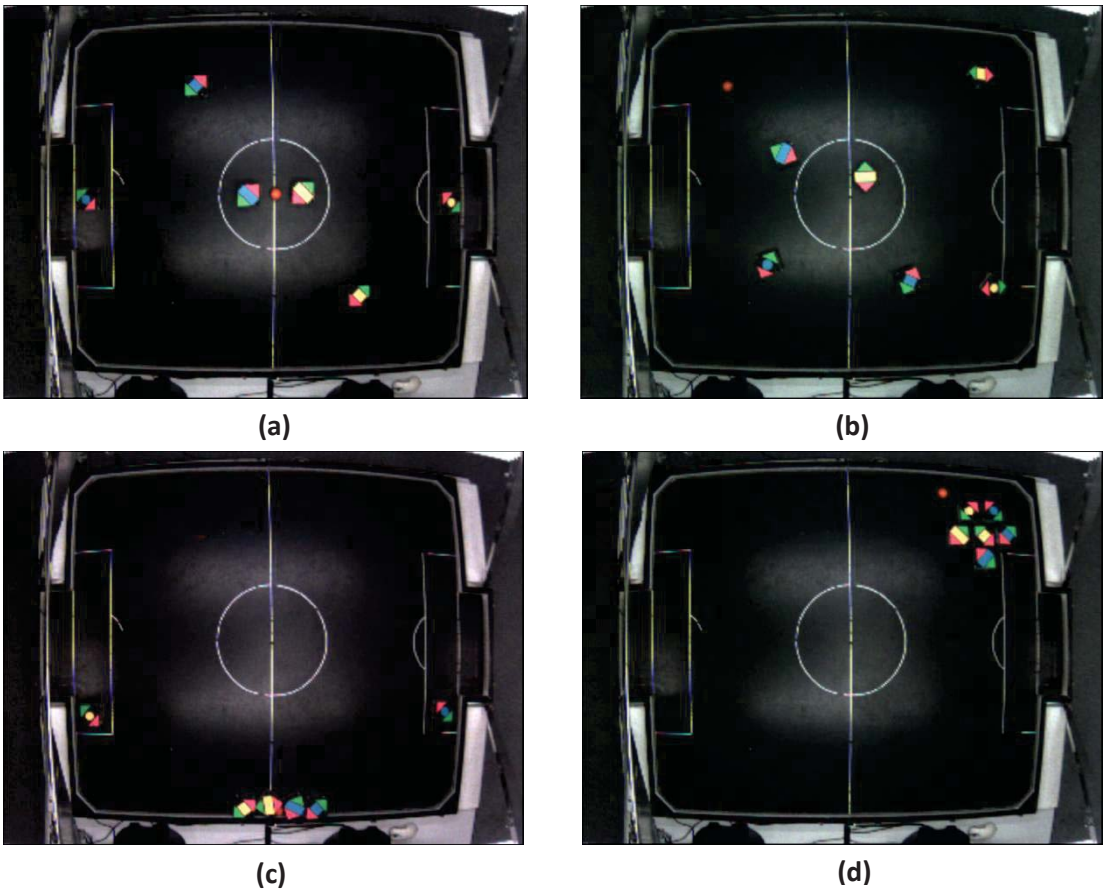


Figure 74 - Various scenarios used to test the robot association module. (a, b) Ideal, (c, d) Worst-case

	Y Circle		Y Square		Y Rectangle		B Circle		B Square		B Rectangle	
	Pink	Green	Pink	Green	Pink	Green	Pink	Green	Pink	Green	Pink	Green
(a)	9.4	9.9	9.9	11.3	10.8	12.2	9.2	9.9	10	9.2	9.9	9.2
(b)	8	10.7	10	9.2	11	11	11.1	9.2	10.4	10.8	10.2	10.4
(c)	9.2	10.8	10	10.6	11	9.5	9.4	8.9	10.6	9.9	8.9	9.8
(d)	10.6	8.5	10.2	10.6	9.2	9.9	9.9	10	9.8	8.5	9.4	10.2

Table 1 - Distance (measured in pixels) between the team patch and correct orientation patch

	Y Circle	Y Circle	Y Circle	Y Square	Y Square	Y Square	Y Rectangle	Y Rectangle	Y Rectangle	B Circle	B Square	B Rectangle	B Rectangle	B Rectangle
	Pink	Green	Green	Pink	Pink	Green	Pink	Green	Green	Pink	Pink	Pink	Pink	Green
(c)						21.1	22					21.5	21.9	
(d)	25.3	22.6	25.1	21.6	26.2	25.3		23.8	25	25	20.9	21.8		26

Table 2 - Distance (measured in pixels) between the team patch and incorrect orientation patch (worst-case)

Based on the information from Table 1 and Table 2, an average of 10 and 23 pixels can be calculated for the distance between the correct and incorrect orientation patches respectively. These values are similar to the physical measurements taken from the robots, shown in Figure

73. From this information, a threshold of 15 can be estimated. This value is larger than each value from Table 1, but smaller than the values from Table 2. This will allow each robot to be properly associated even in a worst-case scenario. Figure 75 shows the results of this test, using a red bounding box to surround the paired triangles and their team patch to symbolise a detected robot.

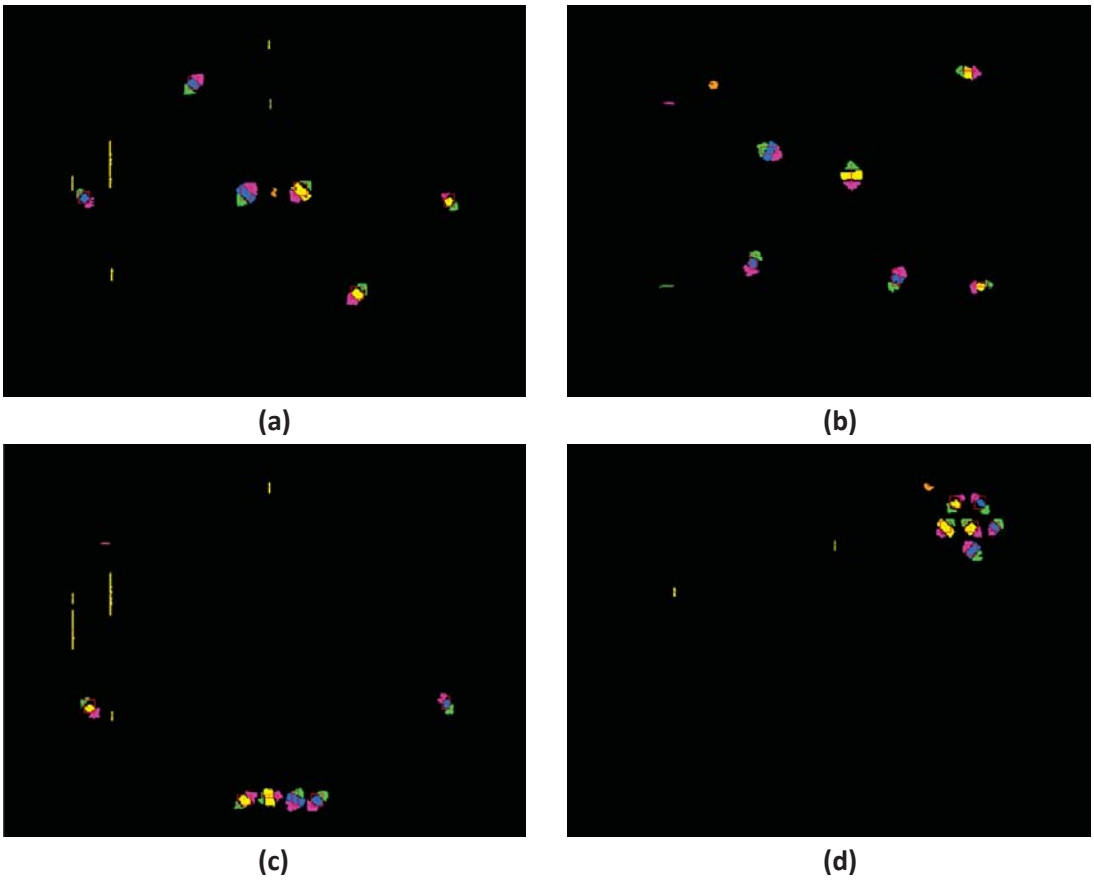


Figure 75 - Results of applying the threshold to each image, showing the identified robots within bounding boxes

These images show that all robots were successfully identified. Even in Figure 75 (d), with all of the robots in close proximity in the corner, the algorithm was able to clearly pair the orientation indicators to the correct team patch. This validates the Euclidian distance, and provides a stable test for the processing pipeline. However, directly calculating the square root of the sum of the squares can be resource intensive. An alternative approach would be to use a linear approximation of the Euclidean distance [61]. From the approximations tested by Filip [61] the one that could be implemented within the FPGA with minimal resources is.

$$D = \max\left(\|x_1 - x_2\|, \|y_1 - y_2\|\right) + \frac{\min\left(\|x_1 - x_2\|, \|y_1 - y_2\|\right)}{2} \tag{19}$$

Equation (19) only requires a few more resources than equation (17). The only extra resource that is needed is the single addition unit to add the two values together. The division is handled using bit shifting operation, which is effectively free.

Figure 74 (d) can be used to test the effectiveness of the approximations using equation (19), as it is the worst-case scenario. The Euclidean distances from Table 1 and Table 2 can be used as a reference to evaluate the approximations. Ideally, a linear approximation should return a near identical result to the non-approximated value. However, as long as a single threshold separating the correct and incorrect orientation patches can be found, the linear approximation approach can be used successfully.

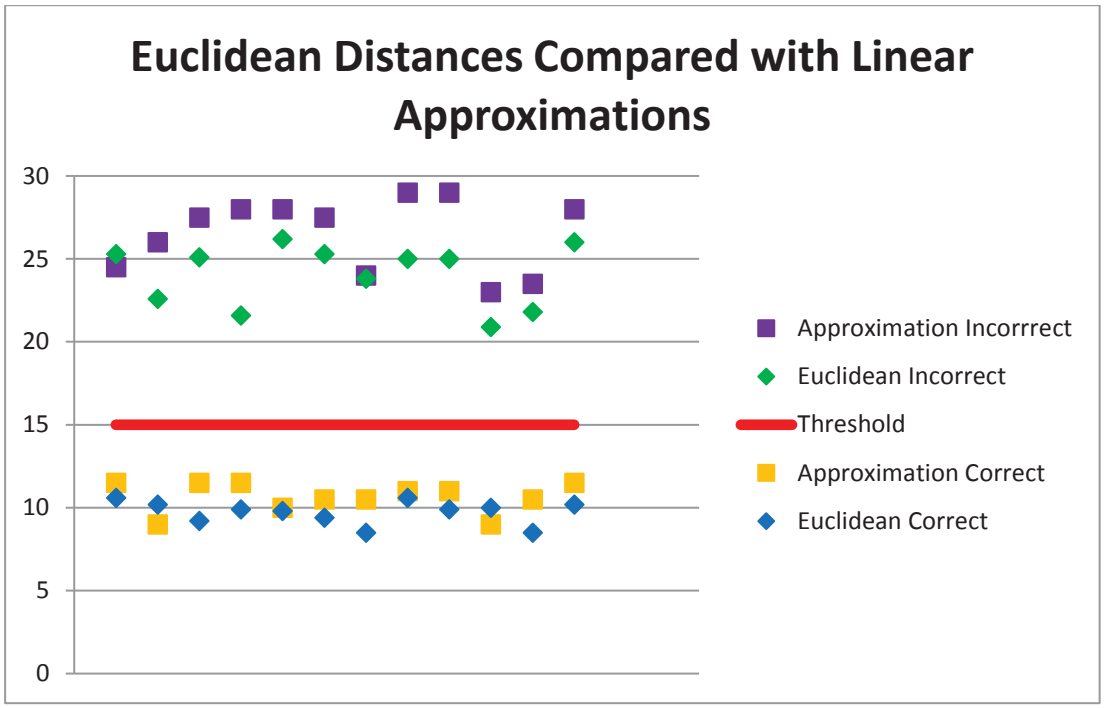


Figure 76 - Euclidean distances compared with linear approximations

Figure 76 shows that Equation (19) generally gives an over approximation to the Euclidean distance for both correct and incorrect triangles, but overall provides a close approximation. Figure 76 also shows a good separation between the correct and incorrect orientation patches during the worst-case scenario. From these results a clear threshold can be found in order to properly associate each robot. Therefore, equation (19) will be used to measure the distances for the association module.

Hardware Realisation

The robot association module processes asynchronous input data. Therefore, the basic structure of this module will be different compared to the previous modules. As described in Chapter 2, the structure for this module will use a FIFO buffer design, similar to Figure 9. Figure 77 shows the layout needed for this module, and can be separated into 4 sections: image processing logic, input buffer, storage tables, and output control.

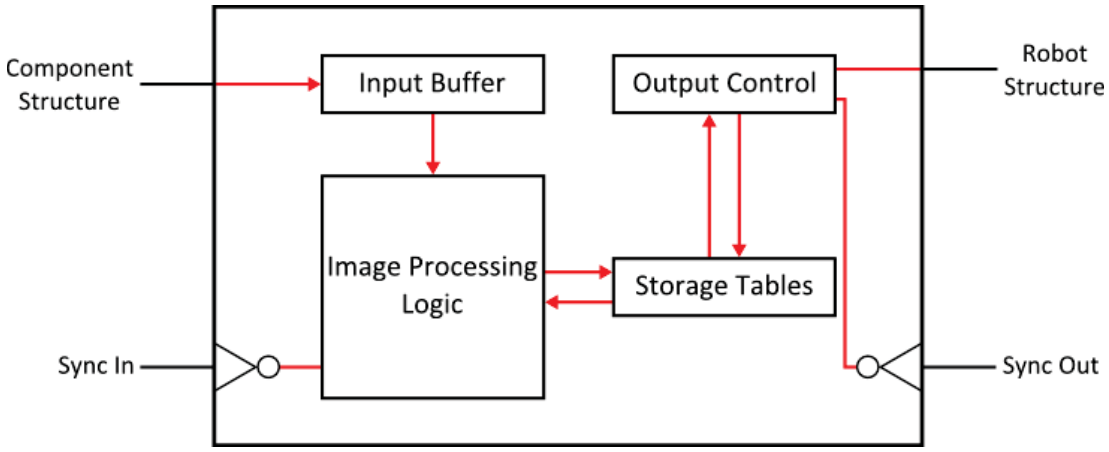


Figure 77 - Block diagram of robot association module

Image Processing Logic

The Matlab algorithm operates using two arrays, one containing team patches and the other containing triangle indicators. Each team patch is compared to each triangle until two matches are found, then the algorithm moves on to the next team patch. This continues until all robots are found or all team patches or triangles are exhausted. Although this method is the easiest way to model the algorithm in software, where all of the patch data is available, the FPGA streaming environment will behave differently.

In the hardware environment, components are passed in as they are discovered. To keep the latency down, it is faster to process the components individually as they enter the association module rather than wait for all the components at the end of the frame.

Figure 78 shows a state transition diagram for the processing logic within the hardware environment.

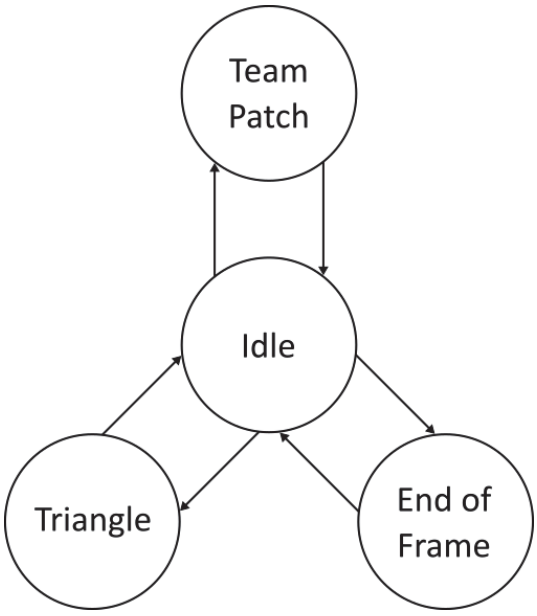


Figure 78 - State transition diagram for the image processing logic, within the robot association module

- Idle – The idle state holds the algorithm until a valid component enters the module.
- End of Frame – Resets memory and counters at the end of the frame. Performing various clean up procedures to start again for the next frame.
- Team Patch – Adds the team patch to a new robot index within storage. Then all unpaired triangles are sequentially compared with the new team patch. Any below the distance threshold are paired with the new team patch.
- Triangle – Compares the new triangle to all incomplete team patches. If the distance is below the threshold, then it is paired with the corresponding team patch. If no team is found, then it is added to a separate array for storage until it is paired.

No robot can be established without first finding the team patch. Therefore, a separate array will be needed to store triangles if they arrive before their respective team patch. Storing these in a memory limits the processing rate to one distance comparison per clock cycle, but also reduces the resources required by the module. In a worst-case scenario, when all triangles are stored at once and no team patches are present, the memory must store at least 12 triangles, and some extra overflow for noise. As new team patches are detected the processing time will vary, depending on the location of the correct triangle indicators. However, the maximum processing latency would be 12 clock cycles plus the amount of extra noise overflow. Similarly, the processing latency of the triangle state can vary depending on how many team patches have been discovered. The maximum processing latency for the triangle state is 6 clock cycles, plus one clock cycle if no pair is found and it is saved to the unpaired triangle array.

With a larger FPGA, it would be possible to use logic resources to temporarily store the data instead of memory. This would allow the distance calculations to be conducted in parallel, performing all comparisons in one clock cycle. This effectively creates a trade-off between faster processing or reduced resources. However, given the small number of robots and limited resources within the DE0, the time-consuming sequential approach is more appropriate here.

Input Buffer

Because the processing can take several clock cycles, especially as more team patches are found, a FIFO buffer will be needed to store incoming components to avoid data loss. In the worst-case, the FIFO will need to be large enough to hold all components at once. Therefore, it will need 31 entries, which includes a safety buffer. This requires a negligible amount of memory from the FPGA. During normal operations, the component will pass straight through the buffer with only a single clock cycle of latency.

Storage Tables

A storage table is required to hold partly associated robots, and unpaired indicator triangles. A register array would occupy roughly 10% of the FPGAs total logic resources, which is a poor use of resources. Therefore, the table will be stored in memory. The main limitation of using memory is that the algorithm is limited to one access of the table per clock cycle, which increases the processing time of the algorithm. To compensate, a small register array is used to

track which memory indexes are populated, and also stores the necessary processing information needed from that index. This reduces the number of reads needed by highlighting important data, allowing faster processing speeds during optimal conditions.

Output Control

The robot association module passes on completed robots as soon as they are found. The tracker array enables detection of completed robots, in real-time, without accessing memory. This allows robots to be transferred as soon as they are completed, and reduces the necessary memory bandwidth for checking. When a completed robot is detected the output control logic will stop image processing for 3 clock cycles, while it collects the robot information from memory, and passes the robot onto the next module. This processing hold is needed to stop memory access collisions, and subsequent data loss.

The output control will also have to deal with incomplete robots. At the end of the frame, due to inaccurate image processing or improper thresholds, some robots may only have one or no triangles associated with them. An advantage to the robot patch design is that it is still possible to calculate the robot's identity and orientation from a single triangle. This allows even partially detected robots to produce valid information, but likely with reduced accuracy. Therefore, at the end of each frame all incomplete robots, with a single triangle, are passed on sequentially until the memory is empty. Any robots with only a team patch are ignored as they cannot be processed further.

3.17. Robot Recognition

After the robot location has been found, its unique identity, and team, needs to be recognised. The identities of each robot are determined by the team patch in the centre of the robot, as demonstrated by Figure 79.

Figure 79 - Robot team and identity patch

Algorithm Development

The three robots on each team are distinguished by different shaped team patches. These shapes are:

- Circle
- Square (with aspect ratio of approximately 1:1)
- Rectangle (with aspect ratio of approximately 2.4:1)

Determining the robots team is straightforward, the colour of the team patch can be extracted from the colour label. The difficult part is distinguishing the shapes of the team patch. Due to streaming, and a single-pass limitation, many feature based approaches cannot be used. For example, convex hull, minimum area enclosing triangles, and Fourier descriptors, would all require multiple passes, and are therefore unsuitable. Similarly, region extracting methods will need to be limited to those that can be gathered in a single pass, such as:

- Area
- Perimeter
- Sum of X and Sum of Y
- Sum of X^2 and Sum of Y^2

Because of the highly unpredictable nature of the field environment, the recognition algorithm must be very robust. The robots must be recognised in any orientation, meaning the algorithm must be rotation invariant. The algorithm must also be insensitive to lens distortions, particularly with the reduction of magnification from the centre of the lens. This will require the algorithm to be scale invariant as well. Lastly, the algorithm must be able to contend with minor shape defects and errors caused by the colour segmentation and connected component analysis modules. These defects are unavoidable and can cause flaws within the shapes.

With these requirements in mind, three methods were chosen and compared in Matlab to determine their viability and robustness before implementation on the FPGA.

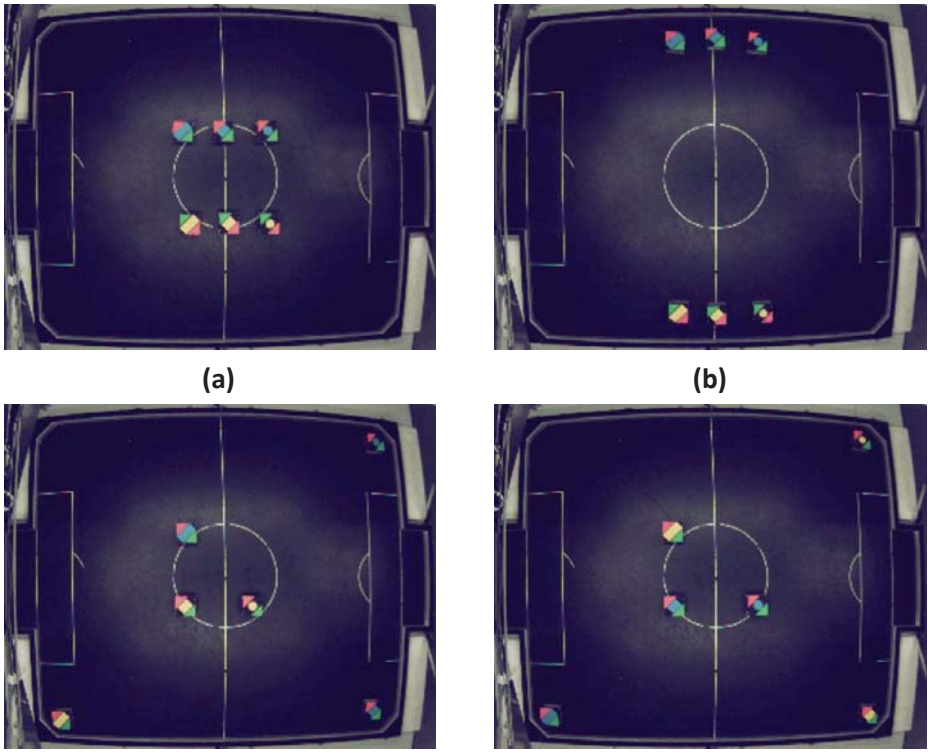


Figure 80 - Test images for the recognition module

Each method will be tested using the same images shown in Figure 80.

Compactness or Complexity

Compactness is a relatively simple and widely used shape descriptor [62, 63]. It uses the perimeter (*P*) and area (*A*) of a binary shape and produces a dimensionless value, irrespective of size and orientation.

$$Compactness = \frac{4\pi A}{P^2} \tag{20}$$

Compactness has a maximum value of 1 for circles, and approaches zero as the shape becomes more elongated and irregular.

$$Complexity = \frac{P^2}{A} \tag{21}$$

Inversely, complexity increases in size as shapes become more irregular. From a computation point of view, complexity is more convenient to calculate on an FPGA as it results in, or can be rounded to, whole numbers. Ideally, the different shapes can be distinguished from their theoretical resulting values:

$$Complexity_{Circle} = \frac{(2\pi r)^2}{\pi r^2} = 4\pi \approx 12.6 \tag{22}$$

$$Complexity_{Square} = \frac{(4h)^2}{h} = 16 \tag{23}$$

$$Complexity_{Rectangle} = \left(\frac{2w + 2h}{wh}\right)^2 \approx 19.3 \tag{24}$$

Where *r* is the radius of the circle, and *w* and *h* are the width and height of the rectangle (*w* = 2.4*h*).

The area can be calculated relatively easily, however accurately measuring the perimeter is more complex. Two methods for calculating the perimeter are tested and implemented into the connected component module, weighted and un-weighted. The weighted perimeter adds a weight of 1.5 to diagonal edge pixels. Whereas the un-weighted perimeter simply accumulates the total number of edge pixels in contact with a background pixel. Both perimeter values were tested using equation (21) in order to determine the most accurate results. See Table 3 for the results acquired from the four test images.

	Un-weighted Perimeter			Weighted Perimeter		
	Complexity	μ	σ	Complexity	μ	σ
Circle	9.60 - 11.08	10.54	0.52	14.06 - 15.94	14.93	0.69
Square	8.65 - 12.60	10.70	1.61	15.88 - 19.56	17.59	1.41
Rectangle	10.64 - 14.40	12.56	1.42	18.55 - 23.03	20.57	1.56
Misclassification Rate	38%			21%		

Table 3 - Results for complexity descriptors from the test images in Figure 80

The un-weighted perimeter produced significant overlap between the shapes, particularly with the circle and square. This meant that the recognition process was not very robust, resulting in a 38% misclassification rate. All of the descriptor values were under-estimated compared to the theoretical values because diagonal lines are considered to have a length of 1. The weighted perimeter value resolves this, and produces values much closer to the theoretical values. These results are significantly improved compared to the un-weighted perimeter values. However, due to the large range and relative closeness of the values, the weighted perimeter produced a 21% misclassification rate. This is still too high to be practical.

Moments

Moments are another common method of characterising shape [62], with the spread of a shape being characterised by its second moment (the variance about the centre of gravity). Objects which are more spread out would have a higher second moment. In two dimensions,

$$\sigma_x^2 = \frac{\Sigma\left(x-\bar{x}\right)^2}{A}, \quad \sigma_y^2 = \frac{\Sigma\left(y-\bar{y}\right)^2}{A} \tag{25}$$

The sum of horizontal and vertical variances is invariant to rotation, but has units of distance squared. Therefore, to derive a scale invariant spread descriptor, the moment invariant can be normalised by the area:

$$Spread = \frac{\sigma_x^2 + \sigma_y^2}{A} \tag{26}$$

Assuming that the origin is at the centre of gravity, then the ideal theoretical descriptor values for the different shapes are then:

$$Spread_{Circle} = \frac{\iint\left(x^2+y^2\right)dxdy}{A^2} = \frac{\frac{1}{2}\pi r^2}{\left(\pi r^2\right)^2} \approx 0.159 \tag{27}$$

$$Spread_{Square} = \frac{\frac{1}{12}\left(w^3h + wh^3\right)}{\left(wh\right)^2} \approx 0.167 \tag{28}$$

$$Spread_{Rectangle} = \frac{w^2+h^2}{12wh} \approx 0.235 \tag{29}$$

Where *w* and *h* are the width and height of the rectangle respectively (and *w* = 2.4*h*).

Central moments cannot be accumulated directly from the connected component module, as this requires knowing the centre of gravity in advance. However,

$$\sigma_x^2 = \frac{\Sigma\left(x-\bar{x}\right)^2}{A} = \frac{\Sigma x^2}{A} - \frac{2\bar{x}\Sigma x}{A} + \bar{x}^2 = \frac{\Sigma x^2}{A} - \left(\frac{\Sigma x}{A}\right)^2 \tag{30}$$

And similarly, for σ_y^2, allows central moments to be derived from four incremental accumulators:

$$Spread = \frac{\frac{\Sigma\left(x^2+y^2\right)}{A} - \left(\frac{\Sigma x}{A}\right)^2 - \left(\frac{\Sigma y}{A}\right)^2}{A} \tag{31}$$

Table 4 shows the results for the spread calculated from the test images in Figure 80.

	Spread	μ	σ
Circle	0.1625 - 0.1681	0.1645	0.0019
Square	0.1634 - 0.1904	0.1743	0.0096
Rectangle	0.1904 - 0.2299	0.2075	0.0134
Misclassification Rate	13%		

Table 4 - Results for moment spread descriptors from the test images in Figure 80

The values for the circle and square are slightly higher than expected, and have some overlap, while the values for the rectangle are lower than expected. However, after choosing appropriate thresholds it is possible to achieve a 13% misclassification rate among all four images, usually misidentifying a circle or square. While this is a significant improvement over the complexity descriptor, it is still too high to be practical.

One of the main reasons for the similarity in values between circles and squares, is that squares, especially those orientated diagonally, tend to develop more rounded corners as a result of segmentation errors, making them more circular. Because the corners increase the spread descriptor of squares over circles, losing the corners makes the values more similar. Conversely, circles tend to become more square-like as a result of distortions introduced from the Bayer pattern demosaicing. For physically larger circles, this effect would be less noticeable, but the small size of the circle patch amplifies this effect. Examples of the ambiguous circles and squares (with overlapping spread descriptors) are shown in Figure 81.

Figure 81 - Ambiguous circles and squares, (left) two circles, (right) two squares

Nevertheless, because these errors are caused by lens distortions, segmentation, and Bayer pattern demosaicing inaccuracies, they are inevitable.

Area

Each of the shapes has a different area. However, this on its own is insufficient for distinguishing between the different identities for three reasons. First, the lens distortion has a significant effect on the area of the patch, with the biggest reduction in the corners of the playing field. Second, the illumination is lower around the edges of the playing field, and the dominant effect of this is also a reduction in the area detected. Third, Bayer pattern demosaicing introduces an uncertainty in the area measurement, increasing the range of values within each of the shapes. This results in significant overlap between the shapes.

However, in the vicinity of the team patch, there are two orientation patches. These will be affected similarly by lighting, and undergo similar distortions. This should enable the area of the orientation patches to be used to normalise the area of the team patch. There are a few ways this can be achieved. Normalising with respect to the average of the orientation patches:

$$A_{normalised} = \frac{2A_{team}}{A_{green} + A_{pink}} \tag{32}$$

should provide minor compensation for different sized orientation patches, assuming both patches are relatively equal. This should allow for a more accurate normalisation assuming both orientation patches distort equally.

However, in some cases due to incorrect segmentation or poor illumination, some orientation patches will be smaller than their pair, which could result in an inaccurate normalisation. Another option would be to normalise against the largest triangle. This would require fewer resources than the average and should provide more accurate results in situations where one orientation patch is significantly smaller than the other.

$$A_{normalised} = \frac{A_{team}}{\max\left(A_{green}, A_{pink}\right)} \tag{33}$$

Table 5 shows the results for the un-normalised area, normalised area by the average of both orientation patches, and normalised area by the max orientation patch. As before, these results are accumulated using the test images from Figure 80.

	Un-normalised			Equation (32)			Equation (33)		
	Area	μ	σ	Area	μ	σ	Area	μ	σ
Circle	36-135	86	34	0.73-1.54	1.06	0.31	0.65-1.25	0.93	0.26
Square	89-204	158	36	1.32-1.77	1.51	0.16	1.19-1.52	1.31	0.14
Rectangle	179-309	260	45	2.30-2.78	2.55	0.18	2.01-2.65	2.28	0.22
Misclassification Rate	13%			4%			13%		

Table 5 - Results for area descriptors from the test images in Figure 80

As expected, the un-normalised area contained a great deal of overlap with the area values for each shape, although surprisingly only lead to a 13% misclassification rate. Both normalising methods have a relatively small range, allowing them to be calculated accurately. However, the max normalisation method produced a greater misclassification rate, at 13%. This is due to the averages being closer together, making thresholding difficult. It is observed that the orientation patches do scale down in size with the team patches in the corners and edges due to the lens distortion. However, in some cases one or more of the orientation patches do not scale equally with the team patch. This can cause the largest orientation patch to be a similar size to the team patch, resulting in reduced separation between shapes. This doesn't seem to be a problem with the average normalisation, which performed the best, producing a 4% misclassification rate. The average normalisation, from equation (32), increases the separation between values allowing for easier thresholding. This is likely caused by the two orientation patches providing a more even value to normalise against, allowing for a more accurate representation.

Of the three methods, the normalised area is the easiest to calculate, requiring only a single area accumulator for each detected component. The normalised average also gave the best results of the three methods tested, and can be considered robust. Therefore, this will be the method that will be implemented on the FPGA.

Hardware Realisation

The processing logic is relatively simple and can be completed in a single clock cycle, as illustrated in Figure 82.

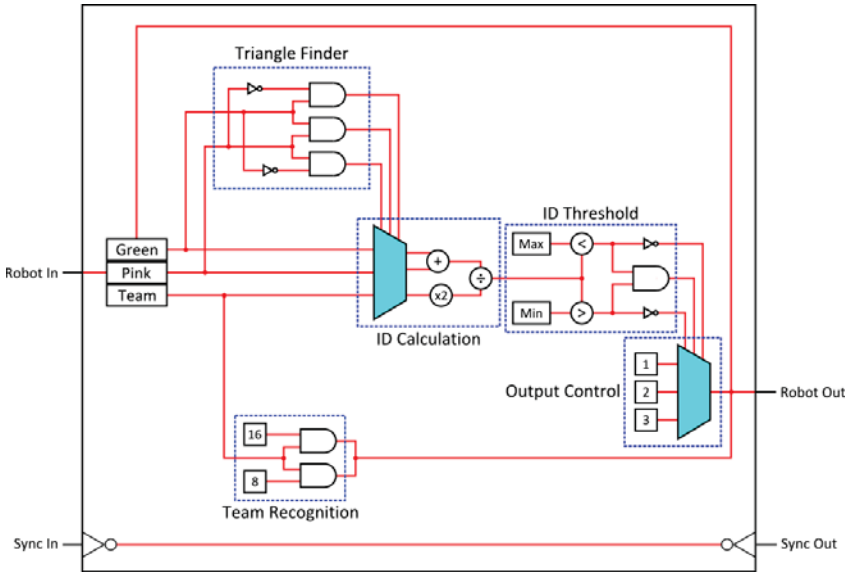


Figure 82 - Block diagram for the recognition module

The normalised area approach cannot be directly implemented as is. The results attained from this method are floating point values ranging from 0 to 3. To use integer arithmetic, the area of the team patch will need to be scaled up by a power of 2 before normalising with the average.

Furthermore, if only one orientation patch is detected, its area is also used for that of the missing patch.

$$Area_{normalised} = \frac{2A_{team}}{A_{single\ patch} + A_{single\ patch}} \tag{34}$$

Table 6 shows the results gathered after testing the recognition module using the test images from Figure 80. The green areas show where a successful recognition was found, and the red areas show fails.

	Y Circle	Y Square	Y Rectangle	B Circle	B Square	B Rectangle
Figure 80 (a)						
Figure 80 (b)						
Figure 80 (c)						
Figure 80 (d)						

Table 6 - Results from testing average area normalised recognition method on the FPGA

These results coincide with the ones calculated in the software development. This demonstrates that the scaling used to increase the values worked without affecting the behaviour or separation of the results.

Since none of the test images have only a single orientation patch, the thresholds were tuned to artificially remove one orientation patch from every robot. This was done to test the concept and pipeline in a non-ideal situation. The results are shown in Table 7.

	Y Circle	Y Square	Y Rectangle	B Circle	B Square	B Rectangle
Figure 80 (a)						
Figure 80 (b)						
Figure 80 (c)						
Figure 80 (d)						

Table 7 - Results for processing only one orientation patch using the same averaging processing pipeline

Even though it is unlikely each robot will lose one orientation patch, the misclassification rate was calculated at 13%. During an ideal scenario, if only one or two robots lose an orientation patch, there is still a high probability that all robots will be classified correctly.

3.18. Orientation and Position

The final module in the image processing chain is the orientation module. During this stage the orientation of the robots is calculated, and the position of the robots, and the ball, are passed to the communication module for transmission.

The position of a robot can be determined using the centre of gravity of the team patch. This is because the team patch is designed to be at the centre of the robot. However, this only provides the coordinates of the robots within the frame of the camera. In order to control the robots, it is necessary to first convert the frame coordinates to real-world coordinates. To

achieve this, the camera would need to localise the playing field within its field of vision, as well as correcting for all lens distortions. Since the camera doesn't move after it has been placed above the field, a procedure would only need to operate once in order to calculate the required conversion factors. Due to the complexity of such an algorithm, and because it only needs to operate once, it is more efficient to perform this calibration in software, at start-up, rather than on the FPGA itself. After calculating the values, the coordinate and orientation conversions can be performed on either the FPGA or the computer. However, since this calibration would initially be conducted in software, it can be considered outside of the scope for this thesis. Therefore, this section will only concentrate on the calculation of the coordinates, and orientations, with respect to the camera frame.

Algorithm Development

The orientation can be calculated from the angle between the centres of the two orientation patches as shown in Figure 83 (a). An offset of 45° is added to find the orientation of the robot relative to the front face, as demonstrated by Figure 83 (b).

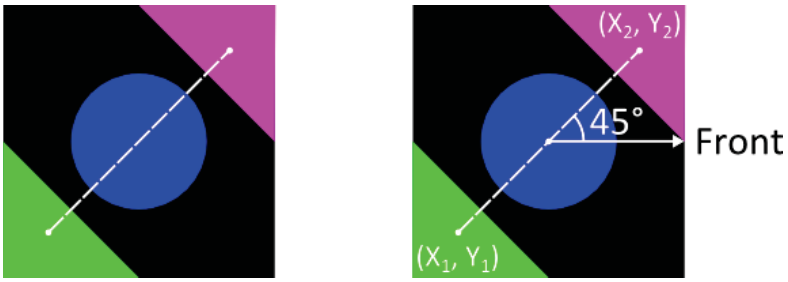


Figure 83 - (a) orientation baseline, (b) orientation offset

In software, the angle can be calculated as

$$Angle_{degrees} = \arctan\left(\left(x_1 - x_2\right),\ \left(y_1 - y_2\right)\right) \times 180 / \pi + 45 \tag{35}$$

An efficient way of calculating angles in hardware is to implement a CORDIC function [64]. CORDIC, short for 'coordinate rotation digital computer' is an efficient algorithm for calculating trigonometric functions. The CORDIC function can be applied using only addition, subtraction, bit shifting and table lookup, making it a relatively simple and resource efficient process. The function works by applying a series of rotations to a vector in an attempt to reduce the angle to 0. If any of the rotations causes the vector to become negative, then this rotation is not applied. After a series of *K* rotations, an approximation of the angle for the original vector is the sum of the angles of the individual rotations. The key to an efficient implementation is to arrange all multiplications to be a power of 2 (a bit shift). This can be achieved using angles $\tan^{-1}2^{-k}$. The accuracy of the approximation can be increased by increasing the number of rotations.

A challenge with CORDIC, in hardware, is how to efficiently represent the angle. When digital variables go below 0, integer overflow or 'wraparound' occurs. This requires special handling in

order to compensate for the wraparound for values between 0-360°. Bailey [12] outlines a solution, “With a binary system, the use of degrees (or even radians) is inconvenient. An alternative is to normalise the angle so that a complete cycle goes from zero to one, which maximises the hue resolution for a given number of bits and avoids the need to manage wraparound of values outside of the range of 0-360°.

Hardware Realisation

The processing stage consists of the CORDIC algorithm. In order to implement the algorithm a pipeline series of addition, subtraction and multiplexors are used, as shown in Figure 84.

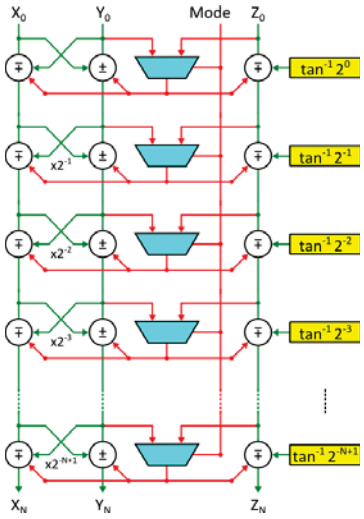


Figure 84 - Block diagram for CORDIC function

Each level in Figure 84 represents an individual rotation. This would require *K* levels for *K* iterations. Two signals (*x* and *y*) are used to control the rotation of the vector, while a third (*z*) is used to accumulate the sum of the rotated vectors. A standard CORDIC algorithm would use logic to check the rotation vector and use this to accumulate the approximation. In order to take advantage of pipelining, the vector approximation will always perform either an addition or subtraction operation, depending if the approximation exceeds its target value. This derives the same result as the standard approach, but is more optimised for a hardware implementation.

The angles are represented using 11-bit values, this allows for 0.17° of accuracy for each bit. Using 7 rotations, the angle can be approximated to within 1° of accuracy, which is enough for this case study. A closer approximation can be achieved by increasing the number of rotations, however doing so adds extra latency and resources.

Each of the rotation levels can in theory be performed as a cascade, allowing a result to be determined within a single clock cycle. This assumes the propagation delay for the calculations in series is smaller than one clock cycle. However, when operating at higher clock speeds, the total propagation delay is longer than a clock cycle. This can be overcome by performing only one rotation per clock cycle and using pipelining to maintain the throughput. Therefore, new

angles can be calculated every clock cycle, with a 7 clock cycle latency needed for the result to propagate through the pipeline.

To allow multiple operations to be calculated sequentially, a 4th signal '*T*' will be used to track valid data within the pipeline, as shown in Figure 85. When an operation is started, a 1 is injected at 'T_0' which propagates at the same rate as the processing. On the output, the 'T_N' signal indicates the presence of completed calculations.

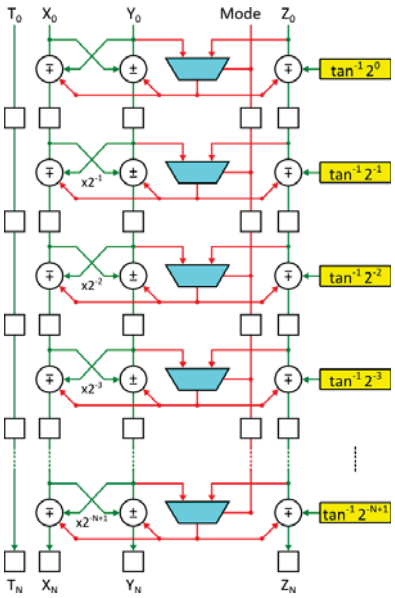


Figure 85 - Block diagram for orientation processing algorithm

By utilising a continuous algorithm, the module can be simplified to a pass-through design, with a latency of 7 clock cycles. Figure 86 shows the black box diagram for the module.

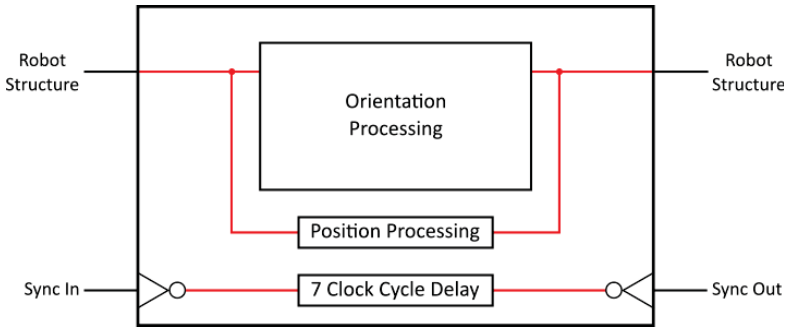


Figure 86 - Black box diagram for the orientation and position module

Testing

When testing the results, a problem was encountered when comparing the processed angles from the FPGA with the angles calculated using Matlab. Matlab calculates the angles in degrees, whereas the CORDIC function in the FPGA calculates a binary representation of the angle. To correct this, the FPGA angles are transferred to a computer first, and then converted to degrees by equation (36), for comparison with the 'gold standard'.

$$Angle_{degrees} = \frac{x \times 360}{2048} \quad (36)$$

Where *x* represents the binary angle.

3.19. Camera Performance

To fully test the robustness of the camera, two separate tests must be completed. The first tests each module against the 'gold standard' images within software. This ensures the individual algorithms function as they were designed. The second test will determine the robustness of all of the algorithms as a whole, and how well they function on a live video stream. The algorithm may function exactly how it was designed for individual images, but may not function correctly with multiple successive images in a video stream. Therefore, the modules need to be tested together to ensure they communicate and synchronise with each other properly, and the modules can work effectively at the required frame rate.

During the algorithm development, the individual modules were already tested against the 'gold standard'. Moreover, the 'gold standard' testing scheme already tests the modules ability to process repetitive images, to a limited extent. By inserting the test image into the module at the beginning of a random frame the modules ability to reset itself between frames is tested. Therefore, this section will concentrate on testing and analysing the performance of the smart camera as a whole.

To test the camera, four scenarios are examined with different lighting conditions. Each scenario consists of a series of 5 images containing all 6 robots and the ball in a stationary position. In order to test the stability between frames it is important to test a variety of images taken of the unaltered field, to ensure the same results are gathered each time. The four scenarios consist of one low lighting condition, two ideal lighting conditions, and one saturated lighting condition. During the ideal lighting scenarios one series of images was captured in sequential frame order and the other at random intervals. The sequential images are meant to test the algorithms capabilities through normal situations, during a short timeframe, at a constant frame rate. Whereas the random interval images (taken equidistantly within a 10-minute interval) are meant to test the capabilities over a longer time frame, at random stages. This is to attempt to simulate the camera over a long length of time since video streams are not able to be captured for testing, due to memory and communication bandwidth limitations. The low and high lighting condition scenarios are also tested using random intervals, to test against a longer time period.

The scenarios will be used to test the values of:

- Area of robot components
- Centre of gravity
- Correctly associate all three robot components
- Correctly identify each robot
- Robot orientation

In theory, since all of the images are of the same stationary scene the calculated values should be identical. Therefore, any deviations in the calculated results should be caused by random noise or changes in the environment (mostly light) affecting the performance of the algorithm as a whole.

For a robust system, it would be ideal if the colour thresholds didn't change between different lighting conditions. This was the main reason for utilising the YC_1C_2 colour space, as it is less susceptible to changes in light. However, during the course of the camera testing it was found that colour thresholds needed to be recalibrated between the separate scenarios with different light levels. Though, this was not the case when testing the separate images within each scenario, or between the two 'ideal lighting condition' scenarios. This shows that while the YC_1C_2 colour space is less susceptible to small changes in light, compared to RGB, it is not impervious and still needs to be recalibrated after significant changes. Having to change the thresholds manually is tedious and time-consuming, and would be better suited as an automated process. However, in terms of testing, changing the thresholds between test scenarios is not necessarily a bad thing, as it provides the best thresholds for testing. This way any errors or anomalies found in the results can be spotted more easily, and are unlikely to be caused by poor colour segmentation.

Another interesting occurrence was found when testing within each series of images. It was observed that there was significant noise fluctuation within the image sets for the random interval scenarios. Some images where segmented ideally, while others had large amounts of noise, shown in Figure 87. This was not observed during the testing of the sequential images, whose noise levels remained relatively consistent.

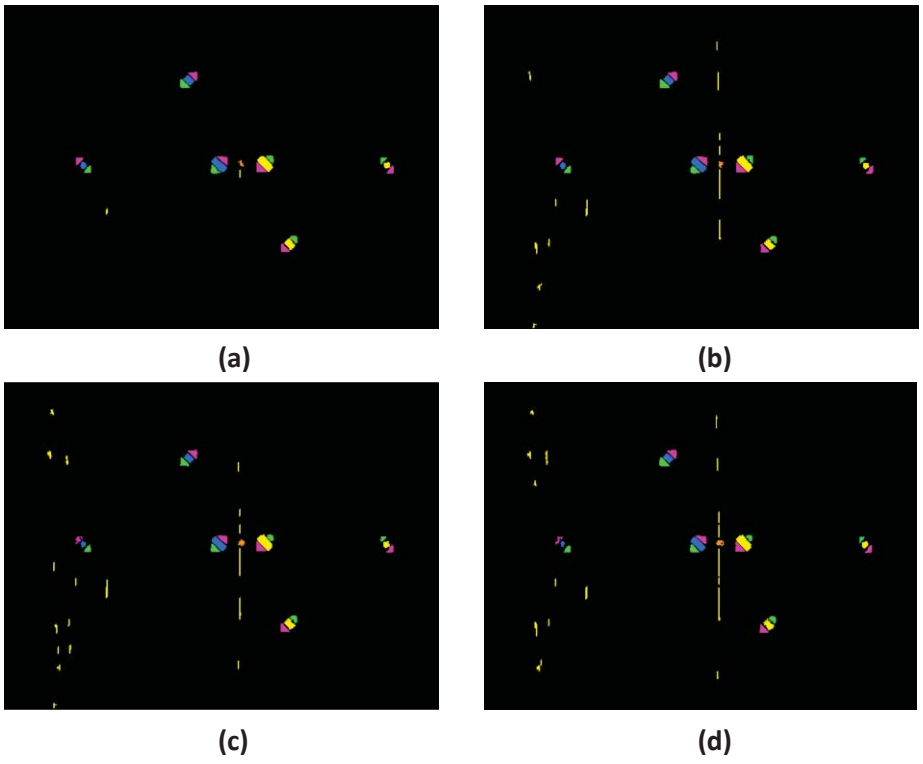


Figure 87 - (a, b) images from random interval scenario, (c, d) images from sequential scenario

Because the noise fluctuation is not seen in the sequential test images it is reasonable to assume that the lighting conditions have changed while testing at random intervals. Since the overall perceived light levels have not changed, a logical guess could be made that the florescent lights used to light the board and room caused the difference. The florescent lights oscillate at 50 Hz; this is not synchronised with the camera (operating at 86 Hz). Therefore, all images will be captured at random intervals along the florescent lights oscillating waveform, effectively capturing the flicker.

This could be fixed in a number of ways. One method is to develop a colour segmentation algorithm that is less susceptible to noise, or is able to compensate for the light flicker. Another would be to synchronise the camera frame rate with the oscillation of the flicker, to ensure images are taken during the same lighting conditions each time. Or, the lights could be changed to a type that doesn't produce a flicker, such as LED or incandescent bulbs.

3.19.1. Area

During the individual module testing, the 'gold standard' images were repetitively tested in order to ensure the image processing could produce an identical result. This was to verify the algorithm functioned as designed on individual images. However, during this round of testing, the results from the series of images will be compared to the 'gold standard'. Since each image will be slightly dissimilar, due to the fluctuation of light and noise, the values are likely to vary. Table 8 shows the results for the four scenarios tested.

	Yellow						Blue					
	Circle		Square		Rectangle		Circle		Square		Rectangle	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Low Light	48	8	113	14	270	11	48	9	119	11	268	15
Ideal (Sequential)	57	5	120	8	300	2	55	4	130	4	280	2
Ideal (Random)	69	7	151	16	293	6	65	30	140	22	280	27
High Light	72	13	152	20	300	29	69	24	143	20	287	21

Table 8 - Results for the detected areas for each robot patch within 4 different lighting scenarios

As expected the averages and ranges differ depending on the light level used, with higher averages found in images with brighter light, and oppositely, low averages in images with low light levels. Yet, generally all ranges are relatively small and the averages for the shapes are distinct and do not encroach on each other. This ensures that each shape is represented with a unique area that is easier to distinguish during the robot recognition module.

Unsurprisingly, the series of images taken sequentially under ideal lighting conditions produced the lowest range, likely due to the relatively consistent lighting conditions. In contrast, the other three scenarios showed wider ranges, likely caused by the difference in environment. Interestingly, the average areas also scaled with the light levels. Since the ideal colour thresholds were recalibrated for each scenario, it was expected that the averages would stay relatively similar. Even though this was not the case, the ranges still manage to remain fairly constant.

With the difference in area detected among the different lighting levels, even under optimised thresholds, special care will be needed for processes requiring a specified area. Obviously, an automatic thresholding algorithm would be useful for keeping the ranges low, although doing so will not stop the averages from drifting between light levels. This could be fixed by investigating different colour segmentation methods that are less susceptible to changes in light. However, since it does not appear that the difference in area has caused any serious limitations within the algorithm there is no need to redesign it for this case study. Although, this does highlight an area for further research in order to improve the reliability and functionality of the robot soccer algorithm.

3.19.2. Centre of Gravity

It is possible that the variations in the area could adversely affect the calculated centre of gravity for each robot. Large ranges in the coordinates would obviously call into question the accuracy for the positions of the robots, as well as potentially causing problems with the association module. Table 9 and Table 10 show the results for the centre of gravity for the X and Y coordinates respectively within the frame.

	Yellow						Blue					
	Circle		Square		Rectangle		Circle		Square		Rectangle	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Low Light	563	2	419	0	385	0	117	2	272	1	316	0
Ideal (Sequential)	562	1	420	1	385	1	117	0	272	0	316	0
Ideal (Random)	562	2	420	0	384	0	117	1	272	0	316	0
High Light	562	1	420	1	384	0	117	1	273	1	316	0

Table 9 - Centre of gravity results for X coordinates

	Yellow						Blue					
	Circle		Square		Rectangle		Circle		Square		Rectangle	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Low Light	236	1	353	0	234	1	236	2	109	1	235	0
Ideal (Sequential)	236	2	353	1	234	0	236	0	110	0	235	1
Ideal (Random)	237	0	353	1	234	0	236	2	110	0	236	1
High Light	236	1	354	0	234	0	237	1	110	0	236	0

Table 10 - Centre of gravity results for Y coordinates

From these results, it is clear that the difference in the areas does not overly affect the accuracy of the positioning coordinates. In a worst-case scenario, the robot coordinates seem to be accurate to within 2 pixels, which equates to roughly 6mm in physical dimensions. Furthermore, the calculated averages are all within 1 pixel of each other. What is most surprising is that the values are so similar across all of the different lighting conditions and time intervals, despite the difference in area and noise level within the image. This suggests that the calculated positions are rather robust, and even though the colour segmentation is somewhat susceptible to light changes this doesn't greatly affect the calculated positions.

An observation can be made that the circles show the greatest range compared to the other shapes. Circles have the smallest area and are therefore slightly more sensitive to noise. This is amplified by their positions on the board. Each circle is placed in the far edges of the field where the light levels are lower and the lens distortion is greater. Based on the area differences, and the known colour segmentation problems with components around the edges of the field, this larger range is not surprising.

The question that stands out however is how such accurate positions are able to be extracted based on the widely varying areas. The most logical assumption is that the difference in area occurs equally around the edges of the shape. By doing so, the value of the area will change but the centre will still remain the same. This holds true to the observations made while manually tuning the colour thresholds. When the thresholds are incorrect the edges are the first to degrade, while the centre of the object remains relatively intact. This is a good sign for the resilience of the algorithm as a whole, as a number of the object processing stages are reliant on the accuracy of the extracted centre of gravities for each component.

The main limitation with these results is there is no way of verifying how close the averages within the frame are to the physical coordinates, and therefore how accurate the averages are. The coordinates calculated are with respect to the image frame. However, since the conversion to physical coordinates is outside of the scope of this case study, knowing the accuracy of the coordinates within the frame is more important.

3.19.3. Robot Association and Recognition

Robot association is heavily reliant on the accuracy of the centre of gravities for each robot component. If the centre of gravities have too great of an error, then the association module will struggle to correctly distinguish all the robot components. Although, due to the accuracy of the centre of gravity module, all of the robots were able to be properly associated with all three components. With these results, the calculated values for the robot recognition and orientation modules have a greater chance of producing robust results, due to both orientation patches being located.

The robot recognition module uses the averages of both orientation patches combined with the team patch to calculate an un-parameterised ID value for the shape. This ID value is then thresholded between 2 values in order to identify each shape. This means that the robot recognition module is susceptible to the area differences found during the area performance test. It would be ideal if a set of threshold values could be used to identify each shape within different lighting conditions. However, due to the area changing with different lighting condition, despite the optimal colour thresholds being used, this would be very difficult. During the robot recognition test, special care was taken to try and determine a set of thresholds that would work with every lighting condition, however this proved impossible. New values had to be determined for each lighting condition in order to ensure the best results were obtained. Table 11 shows the calculated recognition values, as well as a colour code system to show if the robots were identified correctly. Green states that all 5 robots were

correctly identified, yellow shows that more than half of the robots were identified correctly, and red shows that less than half or none of the robots were correctly identified.

	Yellow						Blue					
	Circle		Square		Rectangle		Circle		Square		Rectangle	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Low Light	18	10	36	8	67	10	18	14	29	11	52	9
Ideal (Sequential)	23	6	40	2	75	6	20	8	30	6	69	7
Ideal (Random)	21	8	29	12	70	15	20	8	35	8	54	12
High Light	32	8	49	12	89	13	29	10	42	10	95	11

Table 11 - Results for the robot recognition test (calculated values and colour coded pass or fail)

From these result, 3 robots in total were misclassified. This equates to a 15% misclassification rate. This is higher than the predicted 4% misclassification rate; however, the new rate is calculated using more samples, across a wider range of lighting conditions. If only considering the values gathered during the ideal conditions, which is the standard operating conditions for the smart camera, then that misclassification rate can be recalculated to 5%.

The wide distances between the averages are caused by the differences of the areas between lighting conditions. Surprisingly, it appears the square shapes are the hardest to identify even with the large area ranges of the circle. The smaller area values found with the circle shape tend to lower the recognition values for the circle, improving the separation of the thresholds. However, the square value is sandwiched between two threshold limits, whereas the circle and rectangle only have one threshold to constrain them. This makes the square value more sensitive to the thresholds than the other two shapes, making it more likely to be misclassified.

The fact that new shape thresholds needed to be chosen for different lighting conditions shows that the values, because of the areas, are not light invariant. This means that when the lighting conditions change significantly, the recognition thresholds will have to be adjusted along with the colour thresholds. As demonstrated by the results, this shouldn't be necessary on a short-term basis. However, it will likely need to be done every time the smart camera is initialised and calibrated.

3.19.4. Robot Orientation

Like the centre of gravity test, it is difficult to determine the accuracy of the physical orientation of the robots compared to the orientation calculated in respect to the frame. Although, with a degree of care it is possible to closely approximate the real-world angles with the frame angles, assuming the camera frame is parallel with the field. In order to test the accuracy of the orientation, the robot's angles were measured with respect to the field, as shown in Figure 88.

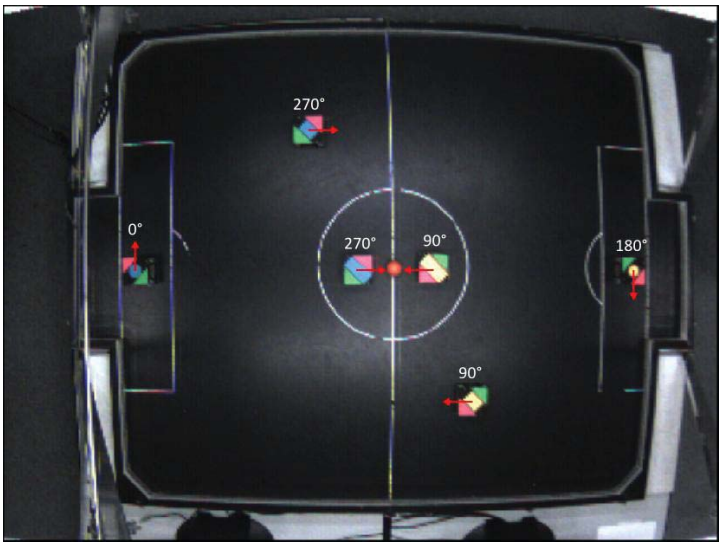


Figure 88 - Measured real-world orientations of robots within test scenarios

These angles were then compared with the orientations extracted from the smart camera. While some deviation in angle is expected, due to inaccuracies with aligning the frame and the field, lens distortions, and slight inaccuracies while measuring the angle, the values should remain fairly similar. As described in section 3.18. , the angle calculated in the robot soccer environment uses a binary scale rather than degrees. For simplicity sake when comparing results, the FPGA orientations are converted into degrees and displayed in Table 12.

	Yellow						Blue					
	Circle		Square		Rectangle		Circle		Square		Rectangle	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Low Light	181	1	89	1	91	0	1	1	269	0	270	0
Ideal (Sequential)	180	0	90	0	91	0	1	0	269	0	270	0
Ideal (Random)	180	1	90	1	91	0	1	1	269	0	270	1
High Light	180	1	90	0	91	0	2	0	268	1	270	0

Table 12 - Orientation results for robots extracted by the smart camera. Converted to degrees from FPGA binary values

The first observation is that the calculated angles are very similar to the measured physical angles. This helps validate the accuracy of the calculated results, and demonstrates that an automatic conversion algorithm can be established. Secondly, these results show very little variation within the calculated angles. The ranges for all results were accurate to within one degree. The variations were likely caused by the minor differences in the centre of gravities for each orientation indicator.

Even though it is not as apparent, as with the other tests, the orientation does still appear to have a wider range in the two circle robots over the other shapes. This again alludes that the results are less accurate on the edges of the playing area, where lens distortion is most severe.

3.19.5. Robot Soccer Algorithm

In total, the robot soccer algorithm proves to be rather robust and accurate. The robot soccer algorithm contains 10 modules in total, and provides the low-latency and high frame rate solution that was aimed for in the beginning of the case study. While the total latency is difficult to measure, due to the robot association module taking a variable amount of time, a total latency can be approximated. The pixel processing stage takes 5 rows and 6 clock cycles of latency; feature extraction requires 6 rows and 8 clock cycles of latency; and the object processing takes approximately 11 clock cycles of latency. This accumulates to a total of 11 rows and 25 clock cycles.

During the testing and development stage, the VGA screen was needed for a number of processes, such as threshold calibration and visual confirmation. Because of this, the smart camera clock speed, and the resulting frame rate, were limited to enable the VGA screen to operate. With the screen in use, the camera operates at 86 frames per second. At this clock rate, the latency is 77 µS. However, if the VGA screen is removed, the clock rate can be increased to 96 MHz, which would operate the camera at 120 frames per second and reduce the latency to 55 µS.

The robot soccer algorithm requires 83% of the resources on the DE0 FPGA development board. While the DE0 can utilise the robot soccer algorithm as it stands now, a more powerful FPGA development board may be needed if further modules or changes are made to the robot soccer algorithm. Given that the DE0 has fewer resources than the DE0-Nano, the algorithm could also be run on the DE0-Nano, allowing the smart camera to take up less physical space. However, doing so would require new drivers to be developed for the DE0-Nano development board, and remove the ability to use a VGA screen. Regardless, this would be an area for further research, and not in the boundaries for this case study.

3.20. Conclusion

In conclusion, this case study demonstrates the development process used to create a functional smart camera. Furthermore, the robot soccer algorithm was implemented successfully and shown to provide accurate results.

The smart camera achieved all of the goals set out by implementing a high frame rate, low-latency, streamed algorithm. The smart camera was developed and tested at 68.75 MHz, which allows the camera to operate at 86 frames per second with only 77 µS of latency. However, the camera is capable of increasing the frame rate to 120 frames per second once there is no need for the VGA screen, reducing the latency to 55 µS.

The colour segmentation was capable of discriminating colours in various lighting conditions. However, the need to manually recalibrate the thresholds proved to be a laborious and tedious task. Moreover, the algorithm struggled to retain the shape and size of many of the components, especially around the edges and corners of the board. While this is in part caused by the lens distortion, most of the fault is with the YC_1C_2 colour space, as it is still somewhat susceptible to lighting variations.

The algorithm utilises 83% of the DE0 FPGA development board's resources. This FPGA uses an older and relatively small chipset compared to other development boards produced by Terasic and its competitors. While this is enough to run the robot soccer algorithm, there is not a lot of resources left to implement other modules on the current DE0 board. In order to expand the robot soccer algorithm, it may be necessary to use a more powerful FPGA.

Chapter 4 Case Study 2: Improved Colour Segmentation

4.1. Introduction

The aim of this case study is to demonstrate how the techniques used to initially develop an FPGA-based smart camera can also aid the modification or improvement of the implementation. To demonstrate this, the image segmentation for the robot soccer algorithm will be further developed in order to improve the usefulness and reliability of the robot soccer algorithm.

The main limitation of the robot soccer algorithm is its susceptibility to illumination changes on the playing field. This makes colour segmentation using the current thresholding method more difficult, and the calibration/setup process tedious and time-consuming. At first glance, it seems logical to simply upgrade the lighting to ideal standards. However, during competition play, teams have no control over the lighting conditions and must use what is provided. Therefore, the system must be robust to lighting changes.

One of the most challenging tasks associated with colour segmentation is finding the optimum threshold values for each colour. This is made harder by changes to the environment, such as uneven light patterns and global intensity changes over time. It is a tedious task to manually calibrate thresholds, especially if the task must be done at the start of every session. The smart camera implemented in Chapter 3 performs adequately after it has been properly calibrated. However, performance drops considerably as the lighting conditions change, for example if regions of the board are darkened due to the shadows of spectators. To this end, measures need to be taken to improve the robustness of the smart camera implementation from variations in illumination.

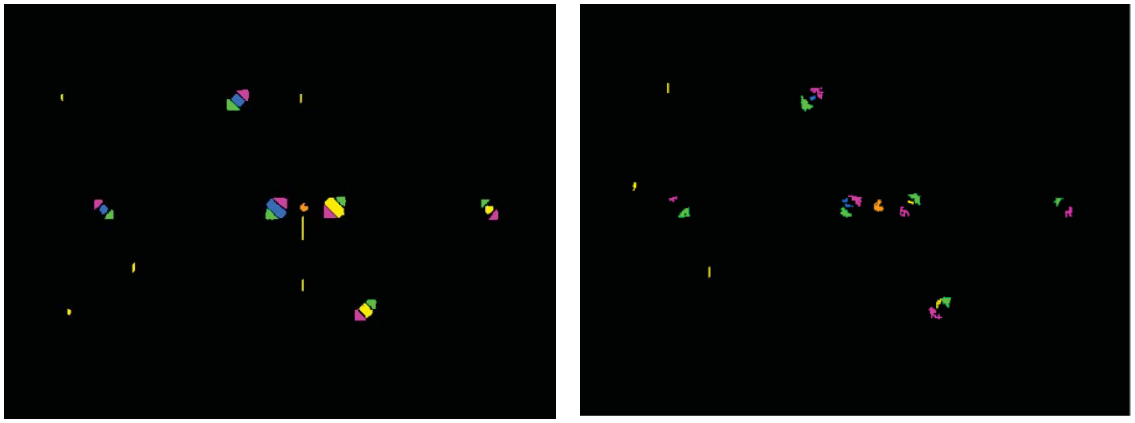
4.2. Context

There are two main ways illumination can affect the camera, global variations and uneven lighting patterns. Global variations can be caused by many things, such as artificial lights flickering, altering weather conditions, etc. This affects most of the cameras field of vision. Similarly, uneven lighting conditions can greatly change the sensed colour values between light and dark areas.

It was observed that changing the lighting conditions altered the thresholded areas significantly. Furthermore, the darker areas on the field, specifically the edges and corner of the board, tended to make the robot patch area smaller. In order to increase the area of the colours in the dark corners the thresholds needed to be widened. However, this introduced a lot of noise and misclassifications throughout the image.

Figure 89 illustrates the changes to the YC_1C_2 colour space by adjusting the global light level. Figure 89 (a) shows the field under ideal lighting conditions with both the field and room lights on. Using the same thresholds, Figure 89 (b) shows the field with the room lights on and the

field lights turned off. In the darker conditions most of the patches are misshappen and decreased in area.



(a) – Ideal Lighting **(b) – Low Lighting**

Figure 89 - The susceptibility of YC_1C_2 to global lighting changes

Figure 90 demonstrates the uneven lighting pattern caused by the lights above the field. Both images were processed using ideal thresholds and under normal lighting conditions.

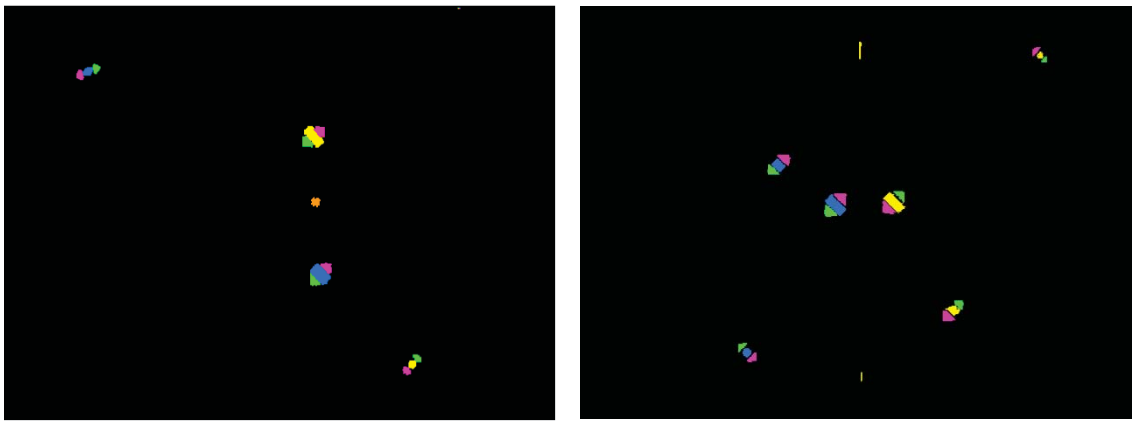


Figure 90 - The susceptibility of YC_1C_2 to uneven lighting patterns

The simplest option for possibly improving the colour segmentation is to consider a different colour space. Many have described the HSV colour space as a reliable way of identifying colour chrominance values irrespective of light [50, 51, 65, 66]. This approach was considered during the development of the robot soccer algorithm in Chapter 3; however, it was dismissed due to the added complexity compared to YC_1C_2. The HSV colour space is well documented and should be relatively simple to implement in an FPGA environment without having to significantly alter the robot soccer algorithm.

4.3. Algorithm Overview

The changes to improve the colour segmentation all take place in the pre-processing stage of the robot soccer algorithm. In order to achieve the best results with HSV based thresholding, the colour space will first need to be white balanced. This will centre the colour space on

white, allowing the hues to be equally distributed around the origin, and reduce the magnitude of some of the background noise. White balancing can be achieved by manually adjusting the individual gains on the red, green, and blue channels. However, manual tuning can be tedious and susceptible to changes in light. A more reliable option would be to automatically adjust the gain and exposure values to produce optimal segmentation conditions. This will require adding a separate module for analysing and adjusting the camera levels or pixel stream. Automatically adjusting the gain values every frame allows the camera to change with the lighting conditions, allowing the algorithm to be more resistant to global illumination changes.

To transform the RGB pixels into the HSV colour space the YC_1C_2 transformation module will need to be modified or replaced. Because of the modularity of the robot soccer algorithm, this can be achieved relatively simply without having to retest the entire smart camera algorithm or each module individually.

Furthermore, with a new colour space, the colour thresholding module will need to be modified to work with HSV. Because parameterisation is used to control the internal module logic, as well as to pass data between modules, making the adjustments to the thresholding module should be simple, and will not affect the internal timing of the thresholding module.

Development and Testing Scheme

The same development and testing scheme outlined in section 2.3. can be applied for this case study. The development of a modified algorithm will still require the use of software languages, like Matlab, in order to quickly prototype the algorithms and create 'gold standard' test images. The algorithm will then be realised on the FPGA, and test images will be used to compare the results between the software and hardware implementations.

When developing new algorithms for an existing smart camera, it may be better to modify an existing module rather than replace it. Generally speaking, it takes less development time to perform minor changes to existing modules than design new ones. However, when making changes to modules, special care should be taken to not inadvertently alter the performance or functionality of the module. With HDLs, it can be relatively easy to accidentally affect timing or synchronisation by adding more operations to a module. Due to modularity, and the serial processing order of the algorithm, it is unlikely that modifying modules will affect the algorithm synchronisation; however, it will likely change the algorithms total latency.

4.4. Colour Correction & Exposure Control

In order to not lose the advantages of stream processing, the corrections will need to be performed on a pixel by pixel basis, without storing the frame or implementing multiple processing passes. This could be achieved by developing a filter, or series of filters, that will modify the image stream in real-time based on previous calculated values. A filtered approach is ideal for situations where direct control of the camera gain values is not possible, or only a single image is processed. A simple example, illustrated in Figure 91, would be to calculate the offset for each colour channel during the frame, these values would then be used to modify

each colour channel during the following frame. This could be implemented using a synchronous pipeline module, and would introduce only a few clock cycles of latency. However, this approach would require some memory in order to accumulate the results and store the offset values. Furthermore, it would take a full frame in order to calculate the initial colour offsets.

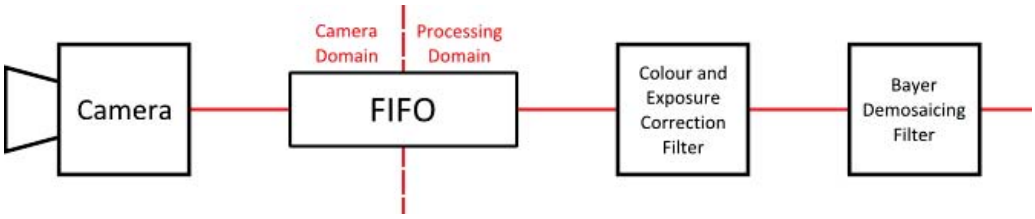


Figure 91 - Colour correction and exposure control as a pre-processing filter

However, with the TRDM-D5M camera, direct control over the pixel output is possible through I^2C. This allows the individual gain values for each colour channel, and the global exposure level, to be adjusted in real-time. For this reason, it makes little sense to implement separate filters after the pixels have been captured, as this utilises unnecessary resources and introduces extra latency. By moving the colour balancing and exposure control to within the sensor readout algorithm, Figure 92, it is possible to analyse the pixel stream from the camera and directly control the colour gains for the next frame before the pixels even make it to the image processing algorithms. This reduces the resources and logic needed to implement the process.

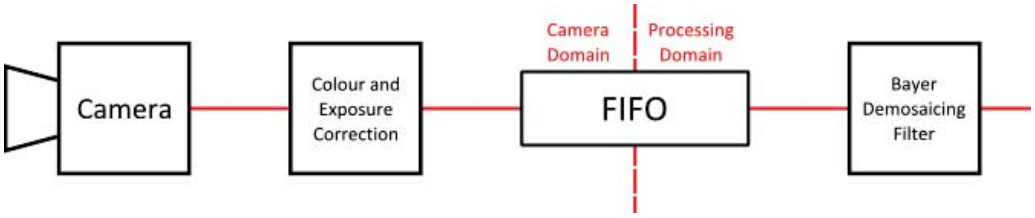


Figure 92 - Colour correction and exposure control in-line with the sensor readout

Since the camera levels are being directly changed, all adjustments will need to be made during the vertical blanking period (between frames) so that enough time is available for the changes to take place and not interrupt the capture of new pixels. Since the frame rate is so high, it can be assumed that the lighting and colour values of the current frame will be very similar to the following frame. This allows small fixed adjustments to be made in order to correct the colour balance and exposure.

Exposure Correction

The dynamic range can be described as the range of available values that can be assigned to each pixel. A low dynamic range would limit the differences between colour values, making colour segmentation harder. Inversely, a too much dynamic range can lead to image saturation. Saturation in the image can have a negative effect, as the actual value of the saturated pixel is lost. The best dynamic range of colour pixels occurs when the brightest pixels are on the verge of saturation. Knowing this, automatic exposure correction is relatively simple

to implement. By collecting a tally of the total number of saturated pixels, the exposure level can be increased or decreased accordingly to conform to an acceptable range.

However, calculating the exact exposure level needed based solely on how many saturated pixels are present is unnecessarily complicated. Therefore, to further simplify the process, an incremental stepping system can be implemented. This way the logic only needs to decide whether too much or too little saturation has occurred, and adjust the exposure level in the appropriate direction. While this method reduces the resources and logic needed it does have some possible disadvantages. Firstly, this method would take longer to effectively balance the exposure levels, especially if the initialisation point is set many steps away from the final level. But with a high frame rate of 120 frames per second, it is conceivable that balancing can be established within a few seconds.

Another potential disadvantage would be possible oscillation of the exposure levels. This would occur if the desired saturation range is much smaller than the exposure level steps. However, this can be solved by applying an adequate hysteresis, and ensuring correct values are chosen for both saturation range and incremental step.

Lastly, flickering caused by artificial lighting, in particular florescent tubes, could be problematic. Florescent tubes in New Zealand operate at 50Hz. This oscillation would require the exposure to constantly change depending on where the frame is sampled on the waveform. Synchronising the camera operations to 50 or 100 frames per second can effectively cancel out the flicker caused by the lights and observe the colours at the same effective light level between pulses. Or at least allow for a gradual adjustment as the relative phase drifts, instead of the algorithm attempting to adjust for a constantly changing value every frame.

Adjusting the exposure does, by definition, affect the timing of the output pixel stream. There is an upper and lower limit to what the exposure can be set to without affecting the operation of the camera. If the exposure reaches its ceiling, and adjustments are still needed, then all of the gain values can be raised equally. Raising the gain values equally effectively brightens the image in a similar way to raising the exposure. This effectively increases the dynamic range of the exposure without desynchronising the camera. Similarly, the gain values can be lowered in instances where the image is too bright, and exposure is at the lower limit. This is useful for widening the variety of lighting conditions the camera can operate in.

Using the incremental adjustment approach, the maximum number of saturated pixels allowed is 1500 out of 307,200. This allows for roughly 0.5% of the image to be saturated, leaving a very large dynamic range. However, the fluorescent lights present in the camera frame made adjustments based on the number of saturated pixels difficult, because the difference in light magnitude relative to the board values is so great. Reducing the exposure to the point where the light pixels were no longer saturated would reduce the dynamic range for the playing area to near 0. By redefining what is considered a saturated pixel, it is possible to effectively cancel out the effects of the lights. For 12-bit pixels to be considered ‘saturated’ an upper limit of 3820 (out of 4095) and a lower limit of 3200 can be defined. This effectively ignores the

saturated pixels from the fluorescent lights, and creates a hysteresis to buffer against small fluctuations between frames.

Colour Correction

White balancing is very important in a HSV colour space, more so than in RGB or YUV. Balancing the colours allows for more even separation between the colour values.

Because colour correction is being performed in the camera clock domain, on raw pixels, the easiest action for altering the colours is to adjust the camera gain levels. However, the gains cannot be adjusted until the end of the frame, or the pixel stream will be interrupted by incorrect values.

The background is mostly made up of black and dark grey pixels, with the colours making up an insignificant number of the total frame pixels. Finlayson et al. [67] states that by assuming the average pixel value across the entire image is a neutral grey, a "grey world assumption" can be made. By this reasoning, the average value for the red, green, and blue pixels should be roughly equal. Consequently, a simple method for applying colour correction would be to merely adjust the colour channels until the averages are equal.

Because there are twice as many green pixels in a Bayer pattern, the green pixels will represent a larger amount of the image's light information. Therefore, the green pixel should be used as the middle point to which the red and blue channels are averaged to, since green pixels make up half of the entire image.

$$G_{red} = \frac{\frac{1}{N}\sum Green}{\frac{1}{N}\sum Red} \tag{37}$$

$$G_{blue} = \frac{\frac{1}{N}\sum Green}{\frac{1}{N}\sum Blue} \tag{38}$$

These equations give the offsets of the red and blue averages with respect to the green average. Given these calculations, it can be concluded that once the balancing has been completed all values will be roughly equal.

$$\sum Red \approx \sum Green \approx \sum Blue \tag{39}$$

Therefore, simple accumulative equations can be derived to calculate the averages, without the need for complex equations. The Bayer pattern alternates with red and green on one row, and blue and green on the next row. By effectively splitting the green pixels in half by rows, the values for the red and blue offset can be accumulated by equation (40) and (41).

$$S_{R-G} = \sum R - G_R \tag{40}$$

$$S_{B-G} = \sum B - G_B \tag{41}$$

Using the Bayer pattern notation seen in Figure 30 from section 3.9.

At the end of the frame, if the offsets are greater than a max threshold, then the respective gain must be lowered. Conversely, if the offsets are lower than a minimum threshold, then the gain is increased. By centring the thresholds around 0, a hysteresis is formed to compensate for any fluctuations in light between frames. Figure 93 shows the block diagram for the colour correction implementation.

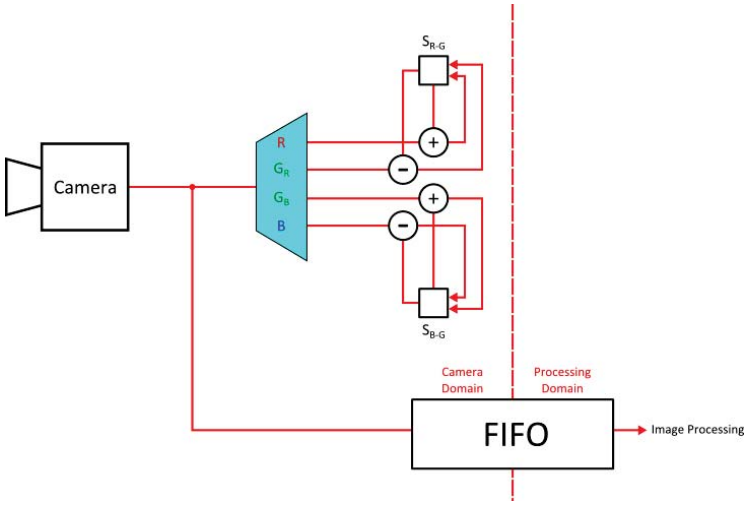


Figure 93 - Block diagram for colour correction logic

A single demultiplexor can be utilised to distinguish which type of Bayer pixel is currently selected. This information is then used to either add or subtract the value from the respective row register. At the end of the frame, an accumulated offset value will have been calculated for the colour correction. Like the exposure levels, the gain intensities are incrementally controlled. Therefore, adjusting the colour balancing will take some time to normalise, but will not require complex logic to implement.

Once the exposure correction has equalised, the colour correction routines will begin. It is important to not adjust the exposure and colour correction during the same frame. This could result in an unpredictable clash, with the gain values being changed separately by each routine.

Figure 94 shows the effects of the exposure and colour correction algorithms on an un-optimised frame.

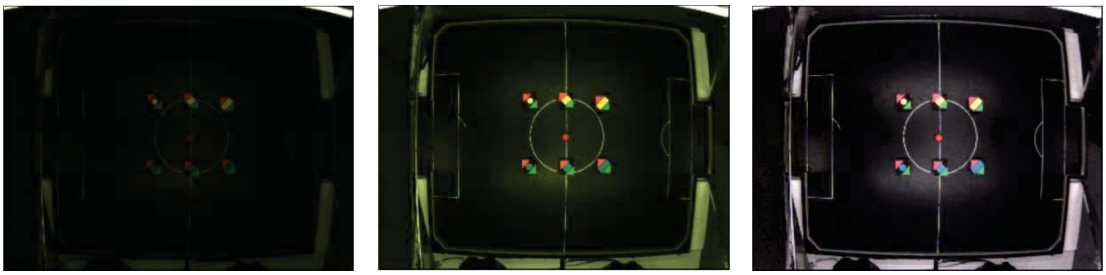


Figure 94 - (left) Un-optimised frame, (centre) Exposure control, (right) Exposure control and colour correction

4.5. Colour Space

One limitation that was discovered using the YC_1C_2 box thresholding method is the edge of shapes is often lost, as C_1 and C_2 values differ greatly from the centre pixels. Similarly, the colours of patches in darker corners of the field also struggle to be identified for the same reason. This could be partially compensated for by increasing the threshold limits, thereby increasing the area within the threshold box. However, this often introduced unwanted noise or created conflict with thresholds for other colours.

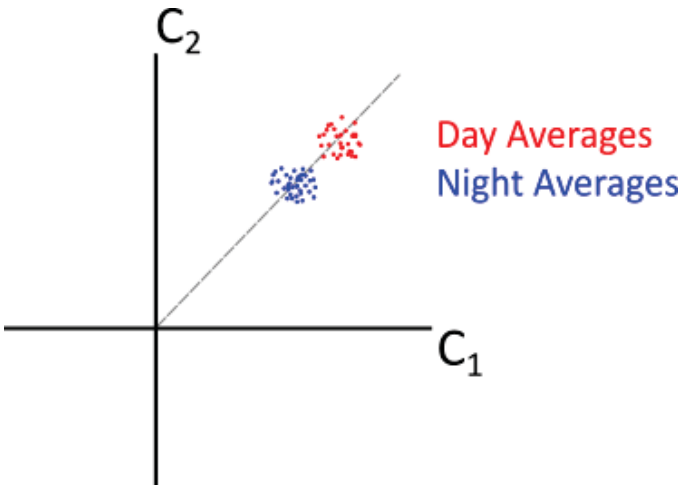


Figure 95 - Averages of different light intensities for a single colour

As illustrated in Figure 95, it is apparent that the fluctuation of light causes a movement towards the origin. This would indicate that in the HSV colour space these colours would be very similar in hue angle. Thus, it stands to reason that a hue based thresholding method would provide better results in uneven lighting conditions, because the threshold parameters allow for variations without having to increase the threshold area. This also reduces the risk of colours overlapping, assuming they are spaced far enough apart. By implementing a polar coordinate system like that used by the HSV colour space, and subsequently a hue based thresholding scheme, it should be possible to more accurately threshold the colours over a wider range of light intensities.

HSV

The HSV colour space models a more intuitive way in which humans describe colours. People can easily identify a base colour *(H)* and define its pureness *(S)* and how light the colour was *(V)*. While this may not seem necessary for a computer using machine vision, the HSV colour space does present some interesting advantages.

Unlike the Cartesian coordinate system utilised by RGB and YUV colour spaces, the HSV colour space employs a colour wheel of sorts. A typical RGB colour space is shown in Figure 96.

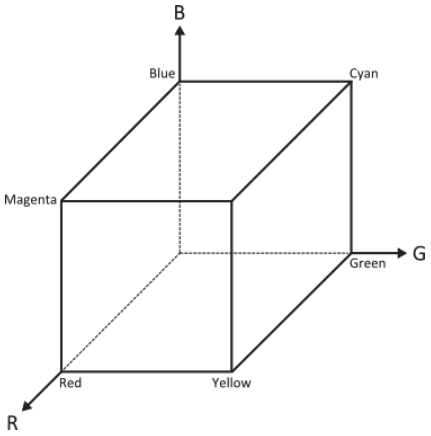


Figure 96 - RGB colour space geometry

The HSV colour space is arranged within a cone, rather than a cube, with all colours represented inside of it by a point in space. As shown in Figure 97, the coordinates for a single point can be determined by its angle *(H)* from a given origin, its radius *(S)* from the intensity axis, and the level of the intensity *(V)*.

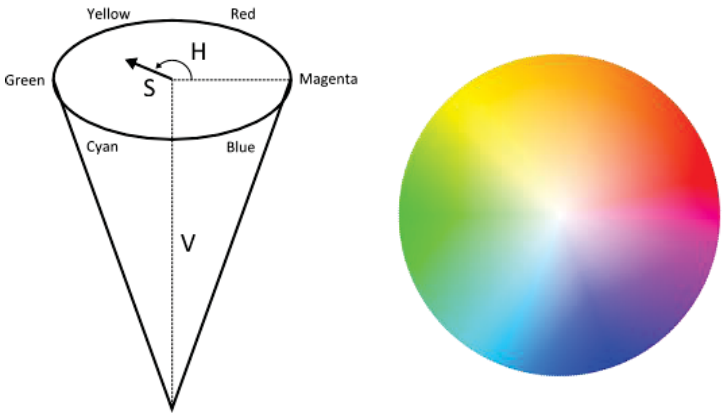


Figure 97 - HSV colour space geometry

By balancing the colour around a white scale, a standardised intensity is assigned to all colours around the colour space's midpoint. This allows the colour space to be simplified to a 2D circle, or colour wheel, comprised of a hue angle and a saturation vector.

HSY

Converting to HSV from RGB is relatively easy, and is outlined by many sources [12, 68]. However, this was initially dismissed as the RGB colour space geometry would need to be transformed and a new coordinate system would need to be established. This would require a more intensive conversion than the YUV-like matrix transformation. However, by converting the chrominance axes of the YC_1C_2 colour space to polar coordinates, similar to HSV, a new colour space can be defined as HSY (Hue, Saturation, Intensity, within the YC_1C_2 colour space), with similar properties to HSV.

Due to the colour balancing performed on the image, only the hue *(H)* and saturation *(S)* need to be converted.

$$H = \arctan\left(\frac{C_1}{C_2}\right) \tag{42}$$

$$S = \sqrt{C_1^2 + C_2^2} \tag{43}$$

Figure 98 shows the distribution of the pixels after converting to the HSY colour space.

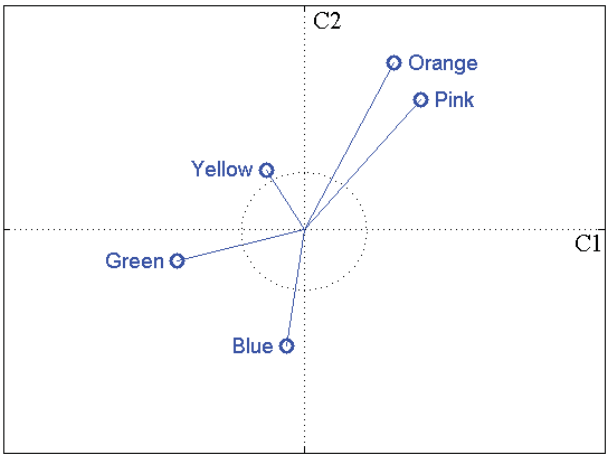


Figure 98 - Colour pixel distribution in the HSY colour space

Even though these colours are well separated to the human eye, this is not always the case for computer vision. Figure 98 shows clear separation between the yellow, green and blue pixels, while the orange and pink are very close together. In order to increase separation, new colours can be selected and placed on the robots. This is relatively simple, but can be a tedious process to find a correct colour combination that works for the computer vision. Another option is to modify the YC_1C_2 transform to virtually adjust the colour separation. This is the preferred method as it can be completed relatively quickly by altering the transform weightings.

This demonstrates that the HSY colour space should be capable of differentiating all of the colours. Furthermore, with increased separation between colours, thresholding could be more reliable and simpler to initialise.

Hardware Realisation

Implementing equation (42) and equation (43) is not very efficient on an FPGA. An efficient way of calculating both the hue angle and colour magnitude is to implement a CORDIC function. As demonstrated in section 3.18. the CORDIC function is a relatively simple process that operates well in a hardware environment, and will approximate accurate results. During the initial software algorithm development, it is better to calculate the angle using the basic arctangent function, and the magnitude using Pythagoras theorem. This allows for any errors to be found when realising the CORDIC function, as both answers should return the same results.

To implement the HSY transformation a separate module is added to the robot soccer algorithm, as shown by Figure 99. It is better to separate the YC_1C_2 transform from the HSY conversion as it is an already realised and tested algorithm. Furthermore, separating the two algorithms allows the HSY conversion to be tested and developed individually.

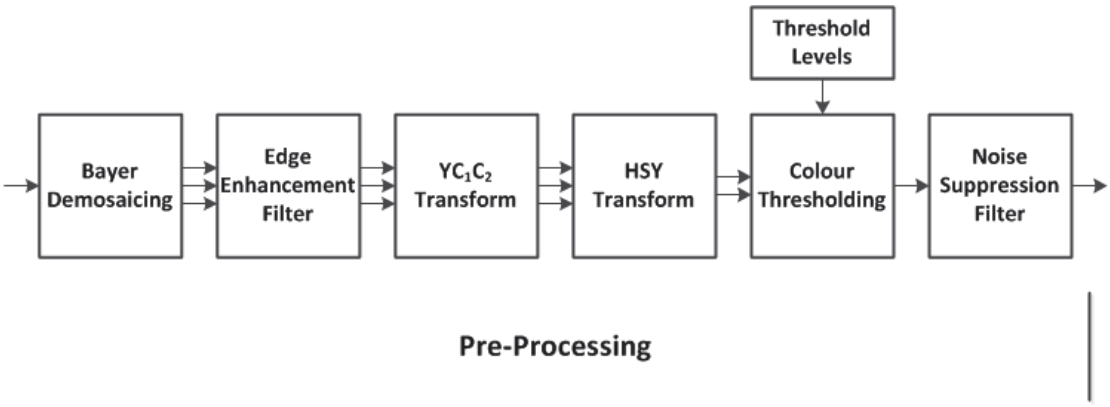


Figure 99 - HSY module location in robot soccer algorithm

Because robot soccer uses modularity, a new module can be entered into the processing pipeline without affecting the synchronisation of the other modules. The HSY transformation will introduce seven clock cycles of latency into the robot soccer algorithm.

4.6. Modified YC_1C_2

Based on the initial results from the hue conversion, the average colour locations show a 47° separation between yellow and orange and only 12° separation between orange and pink, shown in Figure 100. While it is still possible to threshold these values manually, this causes some problems with misclassifications.

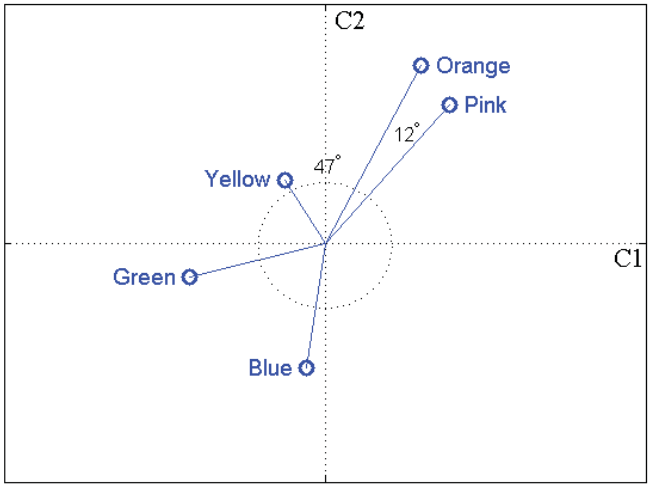


Figure 100 - Colour pixel distribution using original YC_1C_2 transform (11)

Modifying the weightings used to calculate the YC_1C_2 chrominance values should enable a better separation of these colours. This will of course not derive a true representation of the colour hues; however, this does not matter for machine vision. This approach would require

very little change to the overall YC_1C_2 module and fundamentally no extra resources. Making this option the most desirable.

In order to not lose the efficiency of the YC_1C_2 conversion module, the new weights will remain power of 2 variations. To increase the separation between the orange and the pink, new weightings were chosen to rotate the colour space in order to place each colour on separate sides of the axis, as demonstrated by Figure 101.

$$\begin{bmatrix} Y \\ C_1 \\ C_2 \end{bmatrix}_2 = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{44}$$

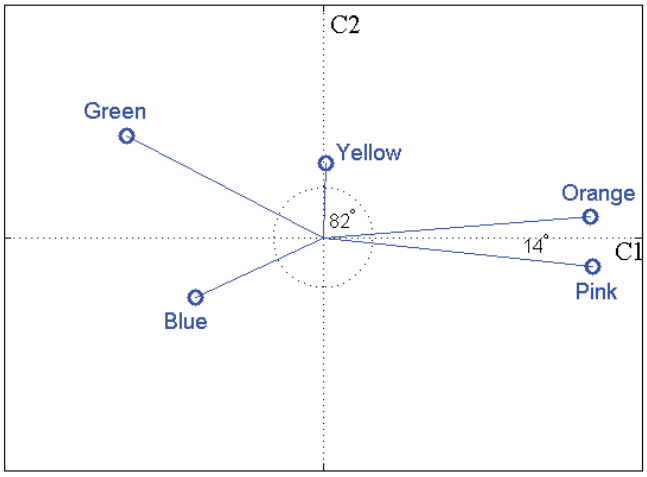


Figure 101 - Colour pixel distribution using equation (44)

Figure 101 shows a lot of improvement compared with the original transform shown in Figure 100. The yellow cluster has been further separated by an additional 35°, without compromising the separation between blue, green, and yellow. The orange and pink however have only separated by a further 2°.

$$\begin{bmatrix} Y \\ C_1 \\ C_2 \end{bmatrix}_3 = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & 0 \\ 0 & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{45}$$

By doubling the C_1 scaling, equation (45) manages to further separate the values along the C_2 axis by an order of 2. This allows the orange and pink values to be further separated, although this does also decrease the angle between green and yellow, illustrated in Figure 102. Even though the colours are not evenly distributed throughout the colour space, equation (45) manages to segregate the clusters more effectively than equation (11). This should allow more reliable thresholding for the colours and remove the pixel misclassifications.

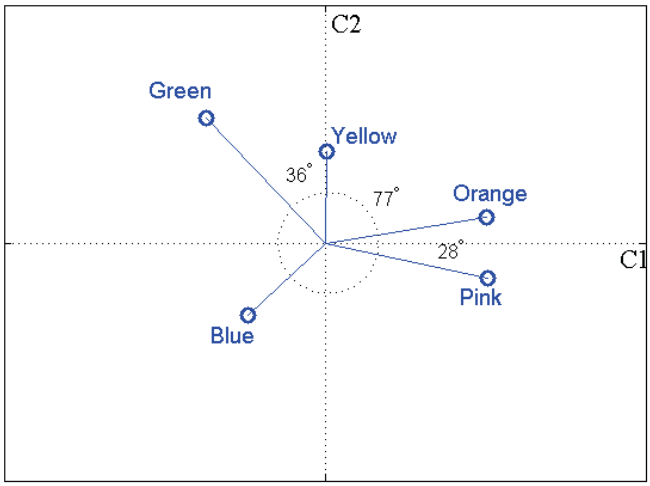


Figure 102 - Colour pixel distribution using equation (45)

These changes will not affect the timing within the module or synchronisation between modules. This example shows the strength of using parameterisation, allowing changes to be made without having to redesign the entire module. After which, only a single image download is required to test the module to ensure the results from the FPGA match with the 'gold standard' from Matlab.

4.7. Hue Thresholding

Once the values have been converted into the HSY colour space they will need to be thresholded into the single binary value. The hue thresholding method requires a maximum and minimum hue angle for each colour class. All of the neutral background pixels are located close to the origin, with very small colour magnitude (Saturation). By implementing a lower saturation threshold, all background pixels can be removed from the image. An upper colour saturation threshold is not required. This should allow for better segmentation in differing light levels on the field. Therefore, each colour can be thresholded using only three limits, as illustrated in Figure 103. With only two hue thresholds per colour class in the HSY colour space, it is far simpler to detect and manage potential overlap between colour classes. This will reduce the probability of misclassifying colour pixels.

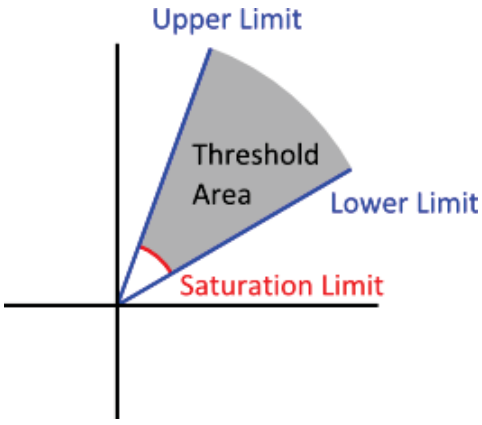


Figure 103 - Hue thresholding limits

Hardware Realisation

To implement the colour thresholding, simple comparators are applied to threshold angles between the two angle limits, and magnitudes above a lower limit, as shown in Figure 104.

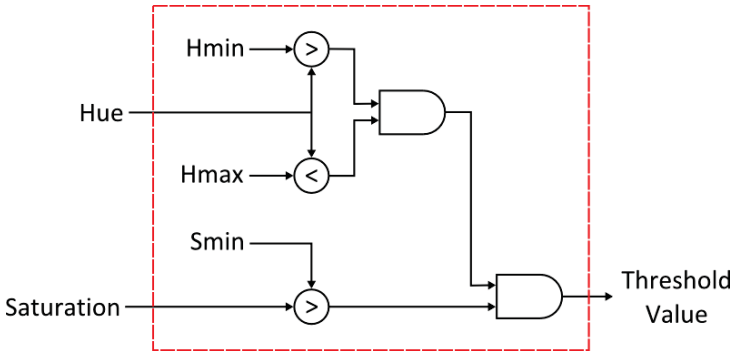


Figure 104 - Individual colour thresholding logic for HSY thresholding

The hue thresholding process functions fundamentally the same as the colour thresholding module implemented in section 3.12. The only difference being that hue thresholding only requires three comparisons per colour class, whereas the YC_1C_2 colour space required six. This allows the previous colour thresholding module to be modified in order to operate with the new colour space. This is a simple procedure and does not affect the internal timing of the module or algorithm synchronisation.

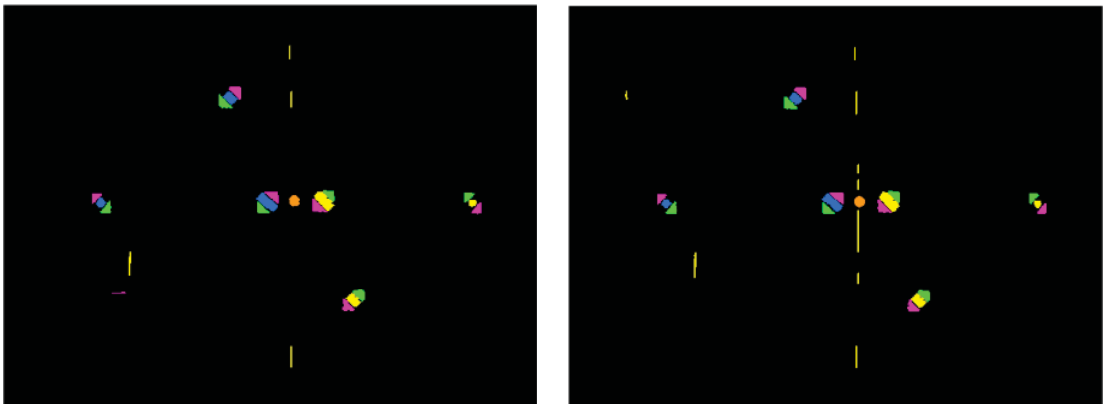
4.8. Discussion

This discussion will test and compare the performance of the HSY colour segmentation method (hue thresholding) with the unaltered YC_1C_2 thresholding method outlined in Chapter 3 (box thresholding). This comparison is made using automatic exposure and white balancing only on the hue thresholding approach, as it is required for thresholding the image properly. While automatic exposure and white balancing would likely make the images more consistent for box thresholding as well, it is not necessary for its basic operation.

The main disadvantages of the box thresholding method implemented in Chapter 3 are:

- Susceptibility to global light changes
- Susceptibility to uneven lighting patterns
- Tedious manual thresholding

Figure 105 demonstrates the same global lighting test, performed in Figure 89, on the HSY colour space. Figure 105 (a) shows the field under ideal lighting conditions, and using the same thresholds Figure 105 (b) shows a lower illumination level with the room lights on and field lights off.



(a) – Ideal Lighting **(b) – Low Lighting**

Figure 105 - HSY global lighting improvement

Both images look very similar and visually there does not appear to be significant difference in terms of colour patch size or shape. The darker image, Figure 105 (b), does appear to have more noise. This is likely caused by the longer exposure needed to compensate for the darker image. However, visually this approach appears to be less susceptible to global light changes compared with the previous approach.

Figure 106 demonstrates the reliability of the hue thresholding approach to operate in uneven light conditions. Both images were captured using similar room lighting conditions with the same threshold levels.

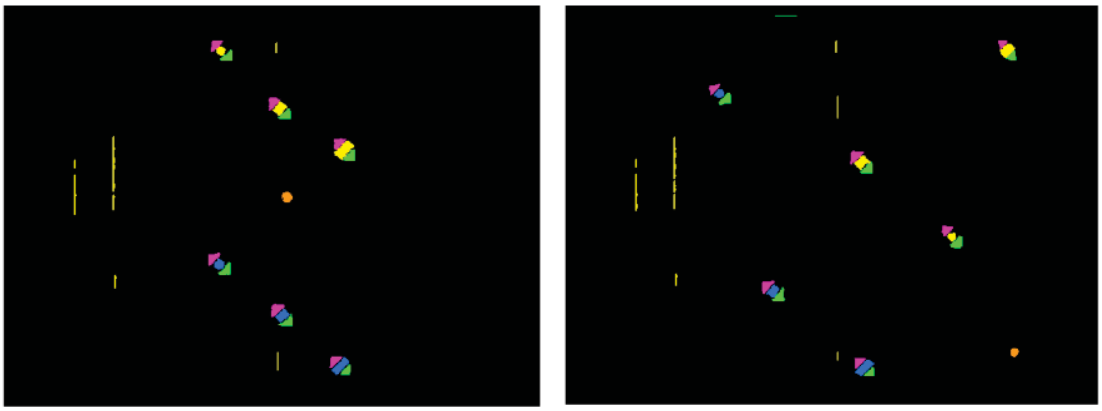
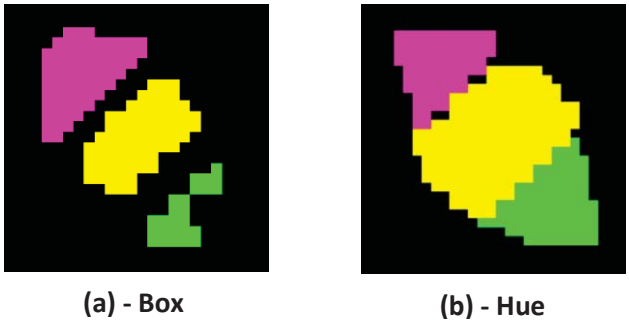


Figure 106 - HSY uneven lighting pattern improvement

From a visual stand point, it would appear that hue thresholding surpasses the performance of the previous box approach. There doesn't seem to be as much shape distortion in low light areas of the field, and there are no missing shapes. Furthermore, the thresholds do not need to be recalibrated between different light levels. This makes the initialisation and configuration relatively easy compared to the box thresholding approach.

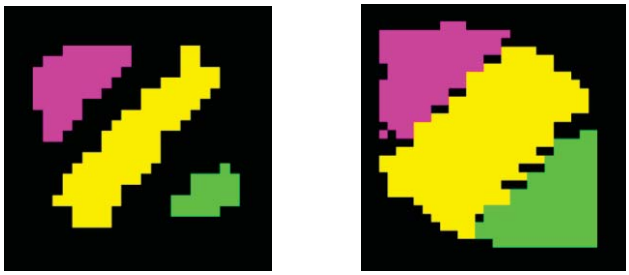
Figure 107 and Figure 108 demonstrates two examples of the improvement hue thresholding makes for reducing distortions compared to the box approach. The left images (a) show the results from the box thresholding method, and (b) show the same robot thresholded using the

hue approach. Both examples used robots located in the corner (where the illumination is lowest) in order to test the capability to compensate for uneven lighting patterns.



(a) - Box (b) - Hue

Figure 107 - Shape distortion from colour segmentation (example 1)



(a) - Box (b) - Hue

Figure 108 - Shape distortion from colour segmentation (example 2)

A big difference can be seen from the green triangle, which retains its shape well using the hue method. Overall, the images processed using the hue approach show better shape consistency for each colour patch, even in dark areas.

Similarly, there does not appear to be as much component size reduction in the edges or corners due to low illumination. Figure 109 (a) shows a robot, thresholded with the box method, located on the edge of the playing field, Figure 109 (b) shows the same robot thresholded with the hue approach. This shows a large improvement for size as well, using the HSY colour space.

(a) - Box (b) - Hue

Figure 109 - Size reduction in low illumination from colour segmentation

From a visual perspective, the shapes in the test images shown in Figure 105 and Figure 106 appear much clearer than shapes attained from the previous method. Analytically, after

comparing the shape recognition values for each robot using HSY thresholding, it is apparent that many of the shapes share a close correlation to each other, independent of their location. This is not observed when comparing shape values using the box thresholding method, shown in Table 13. On average the hue thresholding method has a range of 0.9, while the box method has 2.27. Using hue thresholding, the recognition module is able to cluster the calculated shapes into tighter groups, which in theory makes thresholding easier by decreasing the range. However, the separation between the circle and square groups can be problematic due to the values similarities. With a threshold limit of 'circle < 2.6', only one robot was misclassified across five images. This allows the hue approach to define a single set of thresholds to identify each shape, irrespective of the robot's location on the board. In contrast, this is very difficult to achieve with the box method without increased misclassifications.

	Blue Circle	Blue Square	Blue Rectangle	Yellow Circle	Yellow Square	Yellow Rectangle
Hue Test 1	2.53	2.85	4.67	2.1	3.85	7.25
Hue Test 2	2.26	2.76	4.82	2.76	4.47	7.25
Hue Test 3	2.62	3.13	5.66	2.27	3.79	6.68
Hue Test 4	2.2	3.13	6.85	2.02	3.93	6.83
Hue Test 5	2.31	2.67	5.62	1.82	3.69	6.68
Box Test 1	3.88	4.38	7.87	2.29	3.72	2.29
Box Test 2	3.76	4.78	7.96	1.94	4.15	6.72
Box Test 3	5.86	6.82	7.5	2.47	4.76	6.95
Box Test 4	3.77	4.36	9.54	1.94	4.68	7.45
Box Test 5	4.08	4.44	8.56	2.06	4.87	7.64
Hue Range	0.42	0.46	2.18	0.94	0.78	0.57
Box Range	2.1	2.46	2.04	0.53	1.15	5.35

Table 13 - Comparison of shape values between hue and box thresholding methods

Another advantage of the HSY colour spaces is the reduction of small noise clusters after thresholding. However, this is substituted by large yellow lines that appear over the white field markers, shown in Figure 110.

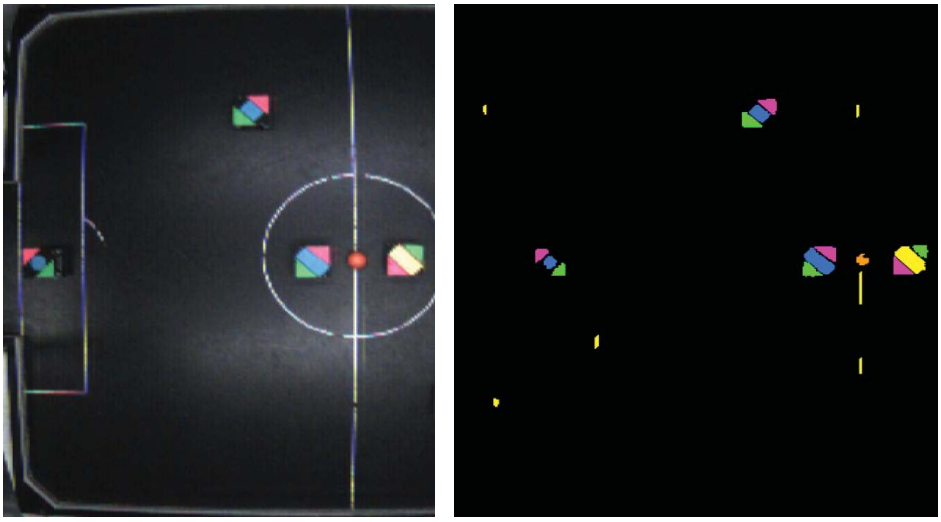


Figure 110 - Colour bleeding introduced by Bayer interpolation

The yellow lines appear to be caused by colour bleeding artefacts introduced by the Bayer interpolation. Simply reducing the thresholding limits does not remove them, as this also removes much of the area within the yellow identification patches.

This could be resolved by applying a Bayer interpolation that is less prone to colour bleeding. As investigated by Bailey et al. [46], there are other Bayer interpolations that can be used that are less prone to colour bleeding, such as altered weights for the current bilinear interpolation. However, these methods require more resources than the current Bayer module.

Another solution could be to adjust or implement a noise filter to specifically remove this type of noise. The lines are relatively thin and vertical so developing a morphological filter to remove them would be relatively simple. In terms of development time, a modification to an already implemented noise filter would be the best course of action, provided the changes effectively removed the noise without altering the colour patches. It would be possible to increase the window area on the noise suppression filter to a greater width than the noise lines. This would enable the filter to remove the lines without greatly affecting the robot shapes, because the robot patches are much larger than the noise.

However, both methods would require more resources or introduce more latency. During the algorithm testing stage, even though the noise is unsightly to human vision, it did not affect the accuracy or operation of the algorithm. Therefore, it is not necessary to add extra latency or processes to remove it.

The final test for the accuracy of the HSY thresholding method is to examine the centre of gravity results. The reliability of the algorithm can be determined by processing two separate images of the same field configuration using identical thresholds for the hue approach. The accuracy can be further measured by comparing the HSY thresholding results with the YC_1C_2 thresholding results.

The results in Table 14 show the calculated positions for the centre of gravity of each robot. Figure 111 was used as the YC_1C_2 comparison image and Figure 105 (a) and (b) were used for the HSY thresholding approach.

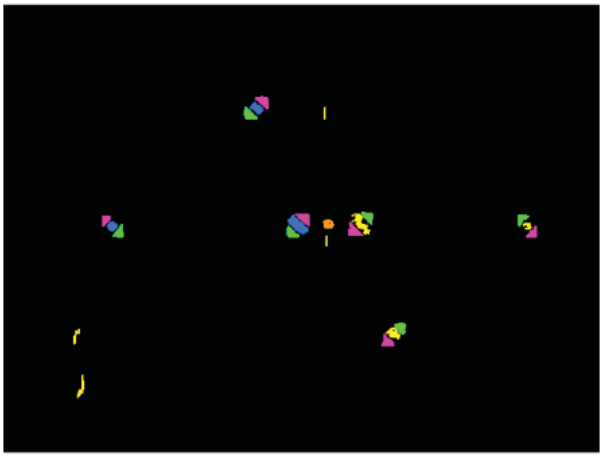


Figure 111 - Box thresholding image for centre of gravity comparison

	Ball	Yellow 1	Yellow 2	Yellow 3	Blue 1	Blue 2	Blue 3
Figure 111	349, 234	562, 237	419, 353	384, 234	117, 236	272, 111	316, 236
Figure 105 (a)	349, 234	562, 237	420, 353	384, 234	117, 237	272, 110	316, 235
Figure 105 (b)	348, 233	562, 235	419, 352	385, 234	116, 235	271, 109	316, 235

Table 14 - Analysis of the centre of gravity results between the hue and box thresholding methods

Table 14 demonstrates that all tests produce very similar results, being at most two pixels apart. This determines that not only is the hue approach reliable compared to other images within the HSY colour space, but also accurate compared with the YC_1C_2 box method.

Surprisingly, the threshold widths needed to attain the HSY results are relatively small compared to the previous method. With an average width of 40°, it is possible to segment each colour and only utilise roughly 55% of the colour space. If each colour can be evenly distributed further, with generous margins between each colour, it would be easier to threshold. However, as it is, the HSY colour space is less tedious to threshold than YC_1C_2.

Adding the hue thresholding modules to the robot soccer algorithm, along with the colour correction and automatic exposure module, introduces 8 clock cycles of additional latency. In total, the algorithm utilises 96% of the FPGA's resources. This significantly increases the compilation time for the HDL algorithm, as more resources require optimisation. However, these modifications can be implemented within the smart camera design without requiring a more powerful FPGA.

These results demonstrate that the hue based thresholding approach within the HSY colour space is very robust. Without having to alter the threshold limits it is possible to accurately threshold colours in the brightest and darkest areas on the field, with relatively small threshold widths. From a visual analysis, the shapes are easier to identify with the human eye. From an analytical perspective, the results from the hue approach were more reliable when identifying shapes, and equally reliable for determining positions, compared with the previous box approach. Furthermore, because only three thresholds need to be adjusted, and the colour classes are relatively separated, hue thresholding is easier to manually calibrate than the box approach. This demonstrates that the HSY colour space and hue thresholding method are far more resilient, and less susceptible to light changes, than the previous box threshold approach within the YC_1C_2 colour space.

4.9. Conclusion

The aim of this case study was to improve the colour segmentation implemented in the robot soccer algorithm. To achieve this, an HSV-like colour space was implemented using a modified YC_1C_2 transform and a polar coordinate system to define the HSY colour space. The results showed that the shape recognition values became independent of light changes, and subsequently where easier and more reliable to threshold. Furthermore, the calculated positions for each robot were equal to the positions calculated using the original YC_1C_2 approach. In terms of accuracy, this makes the new thresholding method using the HSY colour space as reliable as the YC_1C_2 approach.

The value of this case study comes from its demonstration on how to modify an already implemented smart camera. The development time to implement the changes was decreased due to techniques used to implement the algorithm design in the first place. Because of the modular design, the internals for each module can effectively be isolated from the rest of the algorithm. This allows changes to be made within each module, and more modules to be added, without affecting the synchronisation of the camera as a whole. This was necessary as the HSY transform and automatic colour correction and exposure adjustment introduced 8 additional clock cycles of latency into the robot soccer algorithm. In total, this utilises 96% of the Terasic DE0's resources.

In conclusion, hue based thresholding using the HSY colour space was found to be a very resilient form of colour segmentation. Combined with automatic exposure and white balancing, the hue approach is more resistant to changes in light and easier to threshold compared to the previous YC_1C_2 box thresholding method. This makes the robot soccer algorithm more reliable.

Chapter 5 Discussion and Conclusion

5.1. Discussion

In moving the vision system, from a conventional camera and video processor to a smart camera, a number of benefits have been achieved. Parallelism can maximise the resolution/frame rate product. This allows an increase in the resolution, frame rate, or both, over a conventional camera. Being able to directly control the image sensor parameters also gives increased flexibility, because in a conventional camera, many of the low-level features are not user accessible. Processing the data with a synchronous streamed pipeline approach also minimises the latency. On an FPGA, each stage of the processing pipeline is built with separate hardware, so all can operate in parallel. This allows the clock rate of the system to be reduced to the native rate of the pixels being streamed from the camera. Careful transformation of the algorithm allows the potential for data to be extracted, even before the frame has completed loading into the FPGA. While in a conventional system, the processing will not begin until the end of the frame.

Many methods have been explored during this thesis, and the important ones were highlighted and demonstrated using two case studies. While the design methodology was the same throughout both case studies, a selection of unique methods was shown during the development of each module. This was done in order to show a wider range of solutions and design challenges, rather than demonstrating the same method with minor differences.

The development of the first case study was split into 3 different sections, the pre-processing, feature extraction, and object processing. The development of the pre-processing section allowed for the use of many well documented filters in order to capture and convert the pixels into a format suitable for feature extraction. The pre-processing stage demonstrates the large amount of mathematical calculations needed in order to implement image processing. Similar to the pre-processing stage, the feature extraction modules are based on well documented algorithms. However, because they had to be modified to fit a stream-based processing mode, more rigorous testing was needed to ensure they successfully handled the wide variety of scenarios. Lastly the object processing stage was completely custom designed for this role. As such, various methods were initially tested in order to determine the best results. From there, the final designs were implemented and tested in the same way as the previous algorithms. Consequently, the object processing modules contain more testing data than the other modules.

The second case study demonstrates the processes needed to modify an already developed smart camera algorithm designed using the techniques from this thesis. The aim is to demonstrate how these techniques not only simplify the initial development of an FPGA-based smart camera, but also simplify future development or modification. Due to modularity and parameterisation, it is a relatively simple process to change the functionality of the algorithm. To determine if the modifications perform better than the original implementation, extensive testing is used throughout the development process. Many of the algorithms that were considered are specifically designed for this implementation, in order to increase the

robustness of the smart camera in changing light conditions. As such, a more detailed and systematic analysis was needed to develop and test each potential algorithm, to compare the results with the previous implementation.

This chapter will discuss and analyse each of the main techniques demonstrated throughout this thesis. For each technique, a few examples will be highlighted in order to show why they are useful and how they aim to simplify the development of FPGA-based smart cameras.

5.1.1. Development and Testing Scheme

A solid development plan is essential for the development of any project. This is especially true for hardware approaches, due to the non-straightforward debugging tools available, compared to software development. Developing an all-in-one testing scheme is not easy, as each program and situation is unique. For this reason, each testing scheme should be tailored to the appropriate situation. While this isn't likely to change drastically from the examples shown, some changes will likely be needed in order to make development easier. This thesis provides good examples and broad overviews for good testing practices.

Case Study 1: Robot Soccer

A few considerations had to be taken into account when developing the scheme for the first case study. The image processing algorithm needed to be developed in software first in order to achieve the goals for the case study, as image processing development is not feasible within hardware orientated compilers. Fortunately, the software algorithm also provides a 'gold standard' for testing the FPGA algorithm against.

Utilising a software package such as Matlab allows more control and feedback during the initial stage of the image processing development. However, its role is not limited solely to the initial development stage. It is important to have a platform to test the hardware optimisation performance of the algorithm in order to initially adapt it for an FPGA environment. As well as, having another working copy of the algorithm, to aid debugging hardware-based problems.

This was particularly useful during the pre-processing and the object processing stage. Many different approaches could be quickly prototyped and analysed at the pixel level in order to compare the results for each method. It was sometimes immediately evident through visual inspection that a module is unsuitable (such as artefacts in the Bayer interpolation) or not implemented correctly (due to programming errors). For more detailed inspection and comparison, pixel-based analysis can determine many image quality metrics, such as PSNR for different Bayer interpolation weights.

A debate can be made whether or not to make the software algorithm similar, or identical, to the hardware algorithm. A disadvantage of making the two algorithms identical is that this makes finding design flaws difficult. This was discovered during the development of the connected component module, where the concave merging error wasn't discovered until the hardware testing stage. Having independent implementations of the same algorithm can make design or coding errors easier to detect since it is less likely that an identical error will be

implemented in both. Another disadvantage of using similar implementations is the longer prototyping time needed to develop parallel algorithms during the initial software development stage. Software is not efficient at executing parallel algorithms, therefore testing and simulations can take longer to complete. Furthermore, parallel algorithms tend to be more complex, so greater time is needed to write and develop them. Both of these disadvantages can greatly impact the development time during the initial prototyping stage. On the other hand, imitating the functionality and structure of the algorithm in software can save a lot of time when converting the software algorithm to hardware.

For example, the Bayer interpolation could be implemented in software using a series of matrix transformations. Matrix transformations are efficient in software because the entire image is already available, and can be quickly prototyped in Matlab. However, this does not outline how the algorithm would operate in a hardware environment. Therefore, more time will be needed to convert for the hardware approach. On the other hand, by designing the algorithm using a windowed design, and keeping the hardware limitations in mind, the algorithm can be converted relatively easily.

There is no definitive answer as to which methodology is best because both methods have their advantages. For example, when designing the YC_1C_2 filter, it is easier to design a quick transform in software, and utilise the pre-designed window module template during the hardware conversion. This allows for quick prototyping and relatively quick hardware development. On the other hand, for the object processing modules it is better to emulate the hardware programming approach during the initial development, as these modules were specifically designed for this case study. This adds more time to the prototyping stage, but decreases hardware conversion time.

A compromise of sorts can be made, by adding an extra stage to the development of the software algorithm. This method was employed when designing most of the modules in the first case study. An intermediate stage can be added between the software prototyping and hardware conversion stage that is used to optimise the software algorithm to function similarly to how the hardware approach will. This allows for quick prototyping and fast examination, in software, of a potential hardware implementation. The development time for the hardware algorithm can thereby be greatly reduced. This also resolves the issue of simply cloning algorithms, as the initial prototype and hardware emulation in software are two different implementations of the same algorithm. Therefore, this method allows for fast initial prototyping and slightly faster hardware development.

Once the algorithm has been developed within software, and implemented in hardware, the software approach provides a ‘gold standard’ for the hardware algorithm. Having the ‘gold standard’ is helpful for isolating logic errors. During the hardware conversion of the Bayer module, the FPGA results were compared to the ‘gold standard’ and an error was found where three quarters of the pixels were correct but the rest showed seemingly random values. After inspecting the FPGA image, a pattern was found for the incorrect pixels which identified that the problem lay with the blue pixel interpolation. Upon closer examination of the logic and

timing in the hardware algorithm, a propagation delay which violated the timing constraints was found and corrected, fixing the results.

Case Study 2: Improved Colour Segmentation

In procedural terms, the process for modifying a smart camera algorithm or module is similar to designing a new module. Though, during the FPGA conversion and subsequent testing stage the process must be tested at a local and global-level. At the local-level, the new or modified modules will need to be tested to ensure the conversion was successful, and functions as expected. While the global-level will need to ensure the modifications did not adversely affect the operations of other modules. During the initial development, the modifications would have been tested to ensure the correct image processing was achieved. However, during the FPGA conversion, timing and resource allocation must also be tested to ensure the algorithm as a whole functions correctly and optimally.

5.1.2. Modularity

One of the easiest techniques to conceptualise from this thesis is algorithm modularity. Splitting the algorithm into separate modules has many advantages. In hardware, modularity allows simpler design and development, integrated synchronisation and timing control, and algorithm reusability.

During the initial development stage, it is simple to think of the entire image processing algorithm as a collection of different filters and functions. This provides clear boundaries for breaking up the algorithm into smaller modules. By doing so, the programming and development of the hardware smart camera is simplified, as each module becomes a self-contained black box which implements the individual filters or functions. Any coding or design errors can be isolated to a single module. Therefore, once an individual module is complete and tested there is little chance that it can directly interfere with the operation of other modules.

Integrating the synchronisation and timing controls within each module makes it easier to change the smart camera algorithm without needing to adjust each individual module. For example, based on the findings and assumptions from the first case study, it was theorised that a new method of colour thresholding would give improved colour segmentation. Because of the modular design of the overall smart camera algorithm, it is relatively easy to replace the colour thresholding module with an improved one. Additionally, because of the internal synchronisation within each module, no extra adjustments to the smart camera algorithm were required. This allowed for quick and seamless modifications to the smart camera design, and limits the chance of causing synchronisation errors.

With hardware programming, great emphasis is placed on reusability. This can be as low-level as reusing resources or algorithm pipelines, or as high-level as reusing filters or drivers. This is largely due to the fact that HDL implementations are so complex to develop. Because of the clear boundaries surrounding the module algorithms, it is possible to reuse them within other smart camera projects. Furthermore, because HDLs are generic, it is also possible to use

modules across different FPGA chipsets and manufacturers, assuming there is enough resources. This would be very challenging if the code for a filter was fragmented across multiple areas within the smart camera program.

While reusing image processing modules is not clearly demonstrated in this thesis, the reuse of camera drivers is. The drivers for the DE0 development board, such as camera pixel capture, DRAM, I^2C, buttons and switches, are all contained within a common library. Each driver or module can be reused in any number of FPGA projects, and is commonly used as the initial foundation. This significantly reduces the time needed to develop future projects.

Module Designs

Setting standardised module designs can greatly reduce the time needed to develop new modules, as well as simplify the debugging process. This is achieved by creating a set of standardised designs that allow for quick deployment of a module skeleton, with relatively minor changes for individual module functionality. During the course of the two case studies, 3 designs were explored that covered a wide range of processing functionality for the smart camera.

- Window Design
- FIFO Buffer Design
- Pipeline Design

Window Design

Primarily, this is used to implement image filtering that requires neighbourhood information to perform transformation or morphological operations. The design streams pixel information through a cluster of registers forming a neighbourhood region, or 'window'. During the case study in Chapter 3, this design was used to implement Bayer demosaicing, edge enhancement, and the noise suppression filter. The standardised design of this module allowed the dimensions of the window to be adjusted to suit the filter. As the window size increases, the resources allocated increase as well. However, it is possible to reduce the required resources by removing unnecessary parts of the window. This was demonstrated in the design of the noise suppression filter. When implementing a 5x5 window, only data on rows 1, 3, and 5 were needed, therefore the registers on rows 2 and 4 could be removed. However, the row buffers are still required to preserve the frame data and synchronisation.

FIFO Buffer Design

The FIFO buffer design was devised from the need to implement more complex processing methods over a variable period of time. In order to achieve this, results would need to be stored in order to avoid data loss. The resulting module design is comprised of a memory block (FIFO) which is separate from the processing logic. Like the windowed module, this design applies parameterisation in order to adjust the memory width. Doing so allows the FIFO buffer to be easily scaled to suit the requirements, without affecting the functionality of the processing logic.

For small amounts of data, FIFOs can be used within the FPGA for temporary storage. These resources allow for fast access and manipulation of the data, at the expense of limited storage space. However, for larger data volumes, it is better to use external memory modules, such as DRAM. The FIFO buffer design utilised on-chip memory modules. While this allowed faster and simpler management of data, it does restrict access to one read/write operation per clock cycle. The limited access requirements for the FIFO buffer make this approach more suited for asynchronous processing. However, synchronous operations using FIFO buffers can be accomplished using pipelining techniques.

The smart camera design, in Chapter 3, uses asynchronous processing between the camera and display clock domains, in order to synchronise the pixel data across two different clock speeds. FIFO buffer modules are also used to store components and robots in the object processing stage, as the processing techniques were more complex and required a variable completion time.

Pipeline Design

The pipeline design utilises in-line processing techniques to allow processing to occur over a fixed latency without data loss. Pipeline designs are ideal for modules requiring continuous processing or transformation. A synchronous pipeline was used to implement the YC_1C_2 transformation and associated filtering. An asynchronous design was used to implement the centre of gravity module, and the orientation and position module. The difference between these two pipeline designs is how often processed information is output. The synchronous design outputs data every clock cycle, and can be thought of as operating 'continuously'. Whereas, the asynchronous design only outputs data as needed, and can be thought of as operating 'intermittently'. In reality, both designs operate continuously, and function every clock cycle. Because resources are permanently allocated at compile time, in terms of reducing resources or processing bandwidth, there is no advantage to operating a module truly asynchronously.

5.1.3. Parameterisation

Parameterisation, in this context, is the process of defining values that are globally referenceable. This was used extensively throughout the thesis to set global thresholds that could be referenced and altered at either compile time or run-time. For example, setting the colour threshold values and the image pixel width would be a form of parameterisation. This has a number of advantages which are especially useful in a hardware environment. First, defining fixed values in a centralised location allows any changes made to propagate throughout the entire design. As an example, this was particularly useful when changing the raw camera pixel width from 8-bits to 12-bits. This change only increases the width of data being processed, and does not affect the underlying functionality of any of the modules. Without parameterisation each module would need to be altered in order to account for the change in pixel widths, as this must be explicitly defined. This would require a lot of time to modify and retest each module individually. However, with parameterisation only a single test

is required to make sure all modules are operating correctly, as all modifications would have been completed automatically.

Macros can also be used as a form of parameterisation. Macros can be used to derive a global value which is formulated by the result of other fixed or variable values. This form of parameterisation was used in the object processing modules, to compress and decompress data into the storage tables. The bit widths of the components being stored were used to calculate the block size within memory. This allowed the amount of memory required to be automatically calculated at compile time, despite any changes that were made during development.

Parameterisation can be used to decrease the development time by reducing the amount of retesting. Furthermore, it can improve the readability of the algorithm by providing a macro name, rather than a lengthy equation. While it does require a small amount of extra resources and time to develop the algorithm, this outweighs the amount of time saved modifying hard coded values and retesting modules.

When paired with communication to an external computer, control registers can be used to change algorithm parameters at run-time. This allows key parameters such as colour thresholds to be adjusted, without the need to recompile the algorithm. This reduces the amount of down time spent compiling, and allows the algorithm to react faster to changing conditions.

5.1.4. Stream Processing

Stream processing is the best method, at this time, for creating a low-latency image processing application. While it limits the image processing algorithms to pixel based and small neighbourhood approaches, these types of algorithms are ideal for an FPGA environment. Processing each pixel on-the-fly, without temporarily storing the image in memory first, significantly reduces the latency, compared to conventional software means.

5.1.5. Pipelining

Pipelining is a technique for spreading processing over multiple clock cycles, while allowing new data to be input every clock cycle without losing information. This is very useful for stream processing, as it allows more complex and higher latency algorithms to be used without disrupting synchronisation or incurring data loss.

Pipelining can be used with any of the proposed module designs. The Bayer interpolation module utilises pipelining with a window design. The robot association module demonstrates an example of pipelining within a FIFO buffer module. As well as, both the centre of gravity and the hue transformation modules that solely use a pipeline module design.

While pipelining is usually thought of as a low-level technique for implementing algorithms within modules, it can also be used in more complex situations. Each algorithm module can be thought of as a stage in a pipeline. Therefore, the techniques for pipelining can be applied to entire modules, by allowing resources to temporarily store or pass on information at the

beginning and end of each module. This allows the module pipeline to operate in both serial and parallel.

The downside of pipelining is that it adds latency in order to store or pass on information between clock cycles. The smart camera design in Chapter 3 only introduced 25 clock cycles and 11 rows of latency. The inclusion of the hue thresholding from Chapter 4 increases the latency to 33 clock cycles and 11 rows. In both cases, this is negligible and adds only a few microseconds delay, from the time a pixel is read from the camera to when the corresponding processed pixel is output. While this delay may be larger for more complex algorithms, this is acceptable for the robot soccer algorithm.

5.2. Conclusion

Smart cameras can be useful within many different fields, such as surveillance, robotics, and automation. Due to the FPGA's massively customisable and parallel nature, it is well suited to the highly repetitive and high-volume tasks of signal or image processing. Moving the image processing to within the camera can lead to a reduction, or elimination, of backend computing power. As a result, the resource and operating costs can be reduced. As demonstrated in this thesis, an FPGA is an ideal platform for implementing smart cameras. Furthermore, FPGAs provide a smaller and more power efficient platform, compared to conventional computers. This is especially useful for robotics projects that require a small and lightweight form factor for image processing, as space and power are often at a premium.

Despite the advantages that smart cameras offer, they are not often used due to the complexity and time needed to develop them. To address these issues, this thesis set out to explore and demonstrate techniques that can reduce and simplify the development process.

A solid development plan and testing scheme can help reduce the time needed to develop the image processing algorithm. When developing a hardware project, it is much simpler to first design the image processing within a software environment, and then convert the algorithm to a hardware description language. This provides a fast software environment for rapidly prototyping different approaches, as well as a test bench to compare the converted hardware algorithm against.

Parallel systems can be complex to design and implement. By utilising modules, parameterisation, and a solid testing scheme, the development and readability of the algorithm can be simplified. Modules provide a simple way of separating and isolating smaller sections of the overall algorithm. By breaking the algorithm into small isolated pieces, they can be individually developed and tested, simplifying development. Furthermore, isolating the modules from each other can streamline debugging, as each module acts as a self-contained unit and cannot be easily affected by other modules.

Compiling HDL algorithms can be very time-consuming, especially as the algorithm becomes more complex or more FPGA resources are utilised. Having to recompile the algorithm multiple times for testing can add a lot of downtime for development. An efficient development and testing strategy, as well as effective parameterisation and communication,

can reduce the number of times the algorithm needs to be recompiled, greatly reducing the development time. Parameterisation can be used to provide addressable locations to store thresholds or static values. With communication, these parameters can even be modified while the FPGA is running, allowing changes to be made to the operation of the algorithm without altering the core functionality. Thus, the camera can be modified without having to be recompiled, allowing the algorithm to react faster to changing environments.

Finally, in order to get the most accuracy for the robot soccer implementation, the camera needed to have as high a frame rate as possible. This would allow the smallest amount of change within the system between captured frames. To achieve this, stream processing was used to increase processing rates, without relying on relatively slow memory storage resources. However, stream processing requires strict timing guidelines that can restrict the type of image processing algorithms employed. To resolve this, pipelining can be used to break up an algorithm into smaller processes, and distribute the processing over multiple clock cycles. This allows more complex algorithms to be used, while adhering to the guidelines required by stream processing so that no data is lost. The main disadvantage is that pipelining adds extra latency to the system, however, within these case studies the latency was negligible with only 11 rows and 33 clock cycles.

In conclusion, this thesis successfully utilised these techniques in order to develop a working smart camera for robot soccer, which operates at 86 frames per second with only 77µs of latency. Furthermore, by removing the VGA display the smart camera can be increased to 120 frames per second with only 55µs of latency. The techniques demonstrated in this thesis are meant as a guide to help others new to the field to quickly and efficiently develop their own projects. With proper usage, these techniques can be used to simplify the development process and reduce the development time for future FPGA-based smart cameras.

5.3. Future Development

When it comes to future work with respect to smart camera design, there is always room for improvement. Whether it is techniques for shorter development time or more efficient designs, the subject of smart camera development is only in its infancy, especially with respect to FPGA-based designs.

This section is more aimed at the future development for the two case studies that were investigated within this thesis. While they were first conceived as a means of demonstrating and testing the capabilities of an FPGA-based smart camera, they do propose areas for further research. Thereby further exploring the capabilities of smart camera design.

5.3.1. Case Study 1: Robot Soccer

- Automatic camera calibration (perspective error, lens distortion)
- Automatic threshold calibration
- More complex processing

At the completion of this thesis, the smart camera developed in Chapter 3 can distinguish and process the robots and ball within a robot soccer environment. However, more work can still be done to improve the accuracy of the results. Due to the position of the single global camera, the robot's positions can be affected by perspective errors. This can potentially be corrected in real-time by adding a perspective correction module within the FPGA design. Correcting the perspective error would improve the robot position and orientation results, as it will give a truer representation of the real-world values.

More errors that are introduced to the image are caused by lens distortion. This was caused by using a wide-angle lens in order to view the entire field. The most significant error introduced is a relatively large amount of barrel distortion, skewing the calculated robot coordinates and orientations. Since this distortion is fixed (the camera and lens are stationary) the detected robot coordinates can easily be amended after analysing and correcting the barrelling effect.

The smart camera design utilises many manually obtained threshold values, such as colour threshold, shape sizes, component distances, etc. This was originally done because manual values are faster and simpler to develop, and often require fewer resources. However, this does make the design less useable in a changing environment, and requires time-consuming manual initialisation whenever the camera is started. Further research should be completed in order to automate the acquisition of these values. This would allow for faster initialisation, as well as increasing the accuracy of the results from minor changes in the environment (lighting, positioning, colour changes, etc.).

The aim of this thesis is to investigate the design techniques for implementing image processing and control functions within an FPGA. However, in doing so, a platform has been created that can be used as a test bench for a wider area of research. During the robot soccer example, the smart camera only extracts the robot's identity and positional coordinates, while a separate computer is required to process strategy and control the robots. However, with a sufficiently powerful FPGA, and optimised strategy algorithm, it is possible to implement the strategy and robot control from within the FPGA as well. This would offer the strategy algorithm the same advantages as the image processing. By doing so, further latency within the system could be removed, allowing faster reaction speeds and removing the need for external computing power. Other than the advantages towards robot soccer, this type of implementation would further demonstrate the capabilities of a smart camera, by requiring it to make more complex decisions and control the environment around it. This could lead to control systems being integrated within the vision systems of mainstream productions. This would greatly affect the robotics field, by requiring fewer electronic components, as well as allowing smaller and more power efficient designs.

5.3.2. Case Study 2: Improved Colour Segmentation

At the completion of this thesis, an investigation was made that resulted in hue thresholding providing more reliable colour segmentation. The colour thresholding method, as well as the automatic camera exposure and gain control, was implemented within the robot soccer smart

camera design. This greatly improved the accuracy of the calculated results across varying light conditions.

However, the colour threshold values still have to be manually tuned. Further research can be completed in order to develop an automatic colour thresholding approach. This would improve the reliability of the robot soccer algorithm, allowing it to automatically adapt to changing conditions. Furthermore, this investigation could help demonstrate the versatility of smart cameras. By further isolating the camera from human input, the design may become more reliable or at the very least more independent. This would greatly benefit many fields requiring smart cameras, by allowing the cameras to calibrate themselves to changing conditions in real-time, rather than requiring human interaction or manual adjustment.

References

[1] G. Wolfe, "Use of array processors in image processing," in *Design of Digital Image Processing Systems*, vol. 43, 1982, pp. 43-47.

[2] J. Wittenburg, M. Ohmacht, J. Kneip, W. Hinrichs, and P. Pirsch, "HiPAR-DSP: a parallel VLIW RISC processor for real time image processing applications," in *3rd International Conference on Algorithms and Architectures for Parallel Processing (ICAPP '97)*, Melbourne, Australia, 1997, pp. 155-162.

[3] Y. Jie, S. Cong, C. Zhongxiang, H. Ye, L. Liyuan, and W. Nanjian, "Smart image sensing system," in *IEEE SENSORS*, 2013, pp. 1-4.

[4] Z. Nan, C. Yun-shan, and J. Wang, "Image parallel processing based on GPU," in *2nd International Conference on Advanced Computer Control (ICACC '10)* Shenyang, China, 2010, vol. 3, pp. 367-370.

[5] M. Bramberger, A. Doblander, A. Maier, B. Rinner, and H. Schwabach, "Distributed embedded smart cameras for surveillance applications," *Computer,* vol. 39, no. 2, pp. 68-75, 2006.

[6] G. Novak and S. Mahlknecht, "TINYPHOON A Tiny Autonomous Mobile Robot," in *IEEE International Symposium on Industrial Electronics (ISIE '05)* Dubrovnik, Croatia, 2005, vol. 4, pp. 1533-1538.

[7] M. Bramberger, J. Brunner, B. Rinner, and H. Schwabach, "Real-time video analysis on an embedded smart camera for traffic surveillance," in *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '04)* Toronto, Canada, 2004, pp. 174-181.

[8] A. Downton and D. Crookes, "Parallel architectures for image processing," *Electronics & Communication Engineering Journal,* vol. 10, no. 3, pp. 139-151, 1998.

[9] R. Kleihorst *et al.*, "Xetal: a low-power high-performance smart camera processor," in *IEEE International Symposium on Circuits and Systems (ISCAS '01)* Sydney, Australia, 2001, vol. 5, pp. 215-218.

[10] H. Fatemi, "Processor architecture design for smart cameras," PhD Thesis, Technische Universiteit Eindhoven, 2007.

[11] F. Dias, F. Berry, J. Serot, and F. Marmoiton, "Hardware, Design and Implementation Issues on a FPGA-Based Smart Camera," in *First ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC '07)* Vienna, Austria, 2007, pp. 20-26.

[12] D. Bailey, *Design for embedded image processing on FPGAs*, 1st ed. John Wiley & Sons, 2011.

[13] M. Leeser, S. Miller, and Y. Haiqian, "Smart Camera Based on Reconfigurable Hardware Enables Diverse Real-time Applications," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, 2004, pp. 147-155.

[14] S. Kilts, *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007.

[15] B. Mealy and F. Tappero, *Free Range VHDL*. 2012.

[16] W. Caarls, "Automatied Design of Application-Specific Smart Camera Architectures," Applied Sciences, Delft University of Technology, TU Delft, 2008.

[17] P. Wilson, *Design Recipes for FPGAs*, 1st ed. Newnes, 2011.

[18] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2002 (Revision of IEEE Std 1076, 2002 Edn),* pp. 1-300, 2002.

[19] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001),* pp. 1-560, 2006.

[20] M. Asada, M. Veloso, G. Kraetzschmar, and H. Kitano, "RoboCup: Today and Tomorrow," *Experimental Robotics VI,* vol. 250, p. 369, 1999.

[21] D. Bailey, G. Sen Gupta, and M. Contreras, "Intelligent Camera for Object Identification and Tracking," in *Robot Intelligence Technology and Applications 2012*, vol. 208(Advances in Intelligent Systems and Computing: Springer International Publishing, 2012, pp. 1003-1013.

[22] M. Brezak, I. Petrović, and E. Ivanjko, "Robust and accurate global vision system for real time tracking of multiple mobile robots," *Robotics and Autonomous Systems,* vol. 56, no. 3, pp. 213-230, 3/31/ 2008.

[23] N. Zabawi and K. Omar, "Robot soccer vision: An overview for new learner," in *International Conference on Pattern Analysis and Intelligent Robotics (ICPAIR '11)*, Kuala Lumpur, Malaysia, 2011, vol. 1, pp. 125-130.

[24] FIRA Executive Committee, "FIRA small league MiroSot game rules," ed, 2006.

[25] L. DongHun, K. DoEun, H. KyungHun, C. ChaeWook, and K. TaeYong, "A novel color patch system for the large league MIROSOT," in *International Joint Conference SICE-ICASE*, 2006, pp. 576-580.

[26] G. Sen Gupta and D. Bailey, "Fast Image Capture and Vision Processing For Robotic Applications," in *Sensors*: Springer, 2008, pp. 329-352.

[27] J. Brusey and L. Padgham, "Techniques for obtaining robust, real-time, colour-based vision for robotics," in *RoboCup-99: Robot Soccer World Cup III*: Springer, 2000, pp. 243-253.

[28] C. Amoroso, E. Ardizzone, V. Morreale, and P. Storniolo, "A new technique for color image segmentation," in *Proceedings of International Conference on Image Analysis and Processing, (ICIAP '99)*, Venice, Italy, 1999, pp. 352-357.

[29] S. Dhanapanichkul and P. Chongstitvatana, "Shadow compensation for computer vision in a robot soccer team," in *IEEE International Symposium on Communications and Information Technology (ISCIT '05)*, Beijing, China, 2005, vol. 1, pp. 14-17.

[30] P. Bender, J. Beekman, and T. Williams, "Demo: Rapid deployment multi-camera ball tracking system for robotic soccer," in *Seventh International Conference on Distributed Smart Cameras (ICDSC '13)*, 2013, pp. 1-2.

[31] H. Broers, W. Caarls, P. Jonker, and R. Kleihorst, "Architecture Study for Smart Cameras," in *EOS Conference on Idustrial Imaging and Machine Vision*, ed. Munich, Germany: European Optical Society, 2005, pp. 39-49.

[32] F. Tong, Z. Chong, and M. Meng, "Sensor Fusion and Play Strategy Programming for Micro Soccer Robots," in *IEEE International Conference on Robotics and Biomimetics (ROBIO '04)* Shenyang, China, 2004, pp. 833-837.

[33] J. Bruce, T. Balch, and M. Veloso, "Fast and Inexpensive Color Image Segmentation for Interactive Robots," in *Intelligent Robots and Systems (IROS 2000)*, 2000, vol. 3, pp. 2061-2066 vol.3.

[34] J. Baltes, "Practical camera and colour calibration for large rooms," in *RoboCup-99: Robot Soccer World Cup III*: Springer, 2000, pp. 148-161.

[35] G. Sen Gupta and D. Bailey, "Discrete YUV look-up tables for fast colour segmentation for robotic applications," in *Canadian Conference on Electrical and Computer Engineering (CCECE '08)*, Ontario Canada, 2008, pp. 000963-000968.

[36] G. Sen Gupta, D. Bailey, and C. Messom, "A new colour space for efficient and robust segmentation," in *Image and Vision conference New Zealand (IVCNZ '04)*, Christchurch, New Zealand, 2004, pp. 315-320.

[37] D. Bailey, M. Contreras, and G. Sen Gupta, "Towards automatic colour segmentation for robot soccer," in *6th International Conference on Automation, Robotics and Applications (ICARA '15)*, Queenstown, New Zealand, 2015, pp. 478-483.

[38] C. Johnston and D. Bailey, "FPGA implementation of a single pass connected components algorithm," in *4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA'08)* 2008, pp. 228-231: IEEE.
[39] K. Parulski, "Color filters and processing alternatives for one-chip cameras," *Electron Devices, IEEE Transactions on,* vol. 32, no. 8, pp. 1381-1389, 1985.
[40] B. Bayer, "Color imaging array," USA Patent 3971065, 1976.
[41] E. Fossum, "CMOS image sensors: electronic camera-on-a-chip," *Electron Devices, IEEE Transactions on,* vol. 44, no. 10, pp. 1689-1698, 1997.
[42] Terasic, "Terasic TRDB-D5M Hardware Specification," vol. Version 0.2, ed: Terasic Technologies, 2009.
[43] D. Bailey, "Streamed high dynamic range imaging," in *International Conference on Field Programmable Technology (FPT '12)*, Seoul, Korea, 2012, pp. 305-308.
[44] U. Stevanovic *et al.*, "High-speed camera with embedded FPGA processing," in *Conference on Design and Architectures for Signal and Image Processing (DASIP '12),* , Karlsruhe, Germany, 2012, pp. 1-2.
[45] R. Jean, "Demosaicing with The Bayer Pattern," vol. 33, ed. Department of Computer Science, University of North Carolina, 2010, pp. 1-5.
[46] D. Bailey, M. Contreras, and G. Sen Gupta, "Bayer interpolation with skip mode," in *Irish Machine Vision and Image Processing (IMVIP '15)*, Dublin, Ireland, 2015, pp. 68-75.
[47] R. Ramanath, W. Snyder, G. Bilbro, and W. Sander, "Demosaicking methods for Bayer color arrays," *Journal of Electronic imaging,* vol. 11, no. 3, pp. 306-315, 2002.
[48] K. Gribbon, D. Bailey, and C. Johnston, "Colour Edge Enhancement," in *Image and Vision conference New Zealand (IVCNZ '04)*, 2004, pp. 297-302.
[49] M. Comer and E. Delp, "Morphological operations for color image processing," *Journal of electronic imaging,* vol. 8, no. 3, pp. 279-289, 1999.
[50] P. Sebastian, Y. Voon, and R. Comley, "The effect of colour space on tracking robustness," in *Third IEEE Conference on Industrial Electronics and Applications, 2008. ICIEA 2008.*, 2008, pp. 2512-2516.
[51] L. Yang and J. Shu, "Design and Realization of Group ID Image Segmentation for RoboCup Small Size League," in *Third International Symposium on Intelligent Information Technology Application, 2009. IITA 2009.* , 2009, vol. 3, pp. 155-159.
[52] D. Jiwen, L. Jing, F. Aifang, and L. Huiming, "Automatic Segmentation for Ovarian Cancer Immunohistochemical Image Based on YUV Color Space," in *International Conference on Biomedical Engineering and Computer Science (ICBECS), 2010* 2010, pp. 1-4.
[53] A. Rosenfeld and J. Pfaltz, "Sequential Operations in Digital Picture Processing," *J. ACM,* vol. 13, no. 4, pp. 471-494, 1966.
[54] E. Mandler and M. Oberlander, "One-pass encoding of connected components in multivalued images," in *10th International Conference on Pattern Recognition*, 1990, vol. 2, pp. 64-69 vol.2.
[55] P. Zingaretti, M. Gasparroni, and L. Vecci, "Fast chain coding of region boundaries," *IEEE Transactions on Pattern Analysis & Machine Intelligence,* no. 4, pp. 407-415, 1998.
[56] D. Bailey and C. Johnston, "Single pass connected components analysis," in *Proceedings of image and vision computing New Zealand (ICVNZ'07)*, Hamilton, New Zealand, 2007, pp. 282-287: Citeseer.
[57] M. Ni, D. Bailey, and C. Johnston, "Optimised single pass connected components analysis," in *International Conference on ICECE Technology*, 2008, pp. 185-192.

[58] R. Walczyk, A. Armitage, and T. Binnie, "FPGA implementation of hot spot detection in Infrared video," in *IET Irish Signals and Systems Conference (ISSC'10)*, 2010, pp. 233-238.

[59] T. Ellis, D. Proffitt, D. Rosen, and W. Rutkowski, "Measurement of the Lengths of Digitized Curved Lines," *Computer Graphics and Image Processing,* vol. 10, no. 4, pp. 333-347, 1979.

[60] Z. Kulpa, "Area and Perimeter Measurement of Blobs in Discrete Binary Pictures," *Computer Graphics and Image Processing,* vol. 6, no. 5, pp. 434-451, 1977.

[61] A. Filip, "A baker's dozen magnitude approximations and their detection statistics," *IEEE Transactions on Aerospace and Electronic Systems,* vol. AES-12, no. 1, pp. 86-89, 1976.

[62] E. Davies, *Machine Vision: Theory, Algorithms, Practicalities*, 3rd ed. San Francisco, USA: Morgan Kauffmann, 2005.

[63] J. Russ, "The image processing handbook," 2nd ed. Boca Raton, Florida: CRC Press, 2002.

[64] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers,* vol. EC-8, no. 3, pp. 330-334, 1959.

[65] L. Zhu and Z. Zhang, "Auto-classification of insect images based on color histogram and GLCM," in *Seventh International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2010*, 2010, vol. 6, pp. 2589-2593.

[66] X. Zhu, J. Yang, and A. Waibel, "Segmenting hands of arbitrary color," in *Fourth IEEE International Conference on Automatic Face and Gesture Recognition, 2000. Proceedings.*, 2000, pp. 446-453.

[67] G. Finlayson, S. Hordley, and P. HubeL, "Color by correlation: a simple, unifying framework for color constancy," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 23, no. 11, pp. 1209-1221, 2001.

[68] R. Gonzalez and R. Woods, *Digital Image Processing*, 3rd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2008.