NTNU
Det skapende universitet

Autonomt Ubemannet fartøy i søk og redningsoperasjoner - En Synsmodul

Tom Meland Pedersen

Master i kybernetikk og robotikk
Innlevert: juni 2016
Hovedveileder: Amund Skavhaug, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk

Autonomous Unmanned Aerial Vehicle in search and rescue - A vision module

Tom Meland Pedersen

May 2016

MASTER THESIS

Department of Engineering Cybernetics

Norwegian University of Science and Technology

Supervisor: Amund Skavhaug

Preface

This master thesis was done as a part of the fifth year at the study program Master of Science in Engineering Cybernetics at the Norwegian University of Science and Technology. It was carried out during the spring semester of 2016, and is a part of the effort to build an autonomous unmanned aerial vehicle (UAV) for use in search and rescue (SAR) operations. The effort to design and build such a system has been carried out at NTNU since autumn 2012, and this master thesis will partly be a continuation of the project thesis written on the subject during the autumn of 2015. The project thesis was by itself a continuation of a master thesis written during the spring of 2013, as well as a project and master thesis written during the fall of 2013 and the spring of 2014 respectively.

The motivation behind this project was a high interest in the field of computer vision, as well as the desire to produce a specialized and robust module to be used in UAV's. The thought of doing potentially lifesaving work is also a huge motivational factor.

Trondheim, 2016-10-05

Tom Meland Pedersen

Acknowledgment

I would like to give a big thanks to my supervisor Amund Skavhaug for his guidance and help during my last year at NTNU. As well as for letting me borrow literature and hardware needed during the work on the thesis.

Furthermore i would like to thank Simen Solihøgda and Eirik Wold Solnør for valuable feedback and input during the development phases of the project, as well as Øyvind Netland for helping by answering some valuable questions regarding the fields of computer vision.

T.M.P.

Summary and Conclusions

The main goal of this thesis was to design, prototype and test a vision module capable of detecting humans from the air while attached to a Unmanned Aerial Vehicle, so that it can aid human Search and Rescue crews on the ground in finding missing persons in the wilderness. Emphasis was put on using only color cameras and computer vision algorithms in order to reduce the overall cost of the system and attempt to push the limits of modern embedded hardware and the specialized circuits capable of hardware acceleration that are becoming common on these platforms.

A vision module to be used in order to detect humans from a long distance away was successfully designed. A prototype has been implemented as far as time allowed, but not fully completed. The completed submodules of the main module have been tested and verified, and recommendations for further work has been given.

The internal algorithms of the module have been chosen, and the module has been designed in a modular way in order to simplify future additions, changes and the addition of the remaining submodules that have not yet been fully implemented. The primary detection system consists of a object recognition algorithm capable of storing features detected in a reference image for later comparison with features in sample images, allowing detection of similar objects and shapes.

The tests on the completed submodules were quite promising, and the performance of the module so far seems capable of real time object recognition on high resolution images, witch is important in order to enable long range detection.

Sammendrag og konklusjon

Hovedmålet med denne avhandlingen var å designe, prototype og teste en synsmodul som skulle være i stand til å oppdage mennesker fra luften mens den er festet til et Ubemannet Luftfartøy, slik at det kan hjelpe mennesker som utfører søk og redningsoperasjoner på bakken i å finne savnede personer i villmarka. Det ble lagt vekt på å bruke bare fargekameraer og datasynsalgoritmer for å redusere den totale kostnaden av systemet og forsøke å presse grensene til moderne tilpassede datasystemer da spesialiserte kretser i stand til maskinvareakselerasjon stadig blir vanligere på disse plattformene.

En synsmodul som skal benyttes for å detektere mennesker fra lang avstand er blitt designet. En prototype er implementert så langt tiden tilot, men er ikke helt ferdig. De ferdige submodulene av hovedmodulen er testet og verifisert, og anbefalinger for videre arbeid er gitt.

De interne algoritmene i modulen har blitt valgt, og modulen er utformet på en modulær måte for å forenkle fremtidige utvidelser, endringer og ferdigutviklingen av de gjenværende submoduler som ennå ikke er fullt implementert. Det primære deteksjonssystemet består av en algoritme som benytter seg av utregninger for deteksjon og klassifikasjon av lokaliserte trekk i bilder. Den er i stand til å lagre detaljer som påvises i et referansebilde for senere sammenligning med detaljene i andre bilder, slik at deteksjon av lignende gjenstander og former kan oppnås.

De avsluttende testene på de ferdigstilte submodulene virket svært lovende, og resultatene så langt tyder på at det er fullt mulig å implementere en visjonsmodul for sanntid objektgjenkjenning på bildestrømmer med høy oppløsning, noe som erviktig for å muliggjøre deteksjon over lang rekkevidde.

Contents

Preface . . . i

Acknowledgment . . . ii

Summary and Conclusions . . . iii

Sammendrag og konklusjon . . . iv

1 Introduction **3**

1.1 Background . . . 4

1.2 Objectives . . . 6

1.3 Limitations . . . 6

1.4 Approach . . . 7

1.5 Structure of the Report . . . 7

2 Literature study **9**

2.1 Research methods . . . 9

2.1.1 Verifying sources and information . . . 10

2.2 Related work . . . 11

2.2.1 OpenCV . . . 11

2.2.2 SHIFT . . . 11

2.2.3 Object detection algorithms . . . 12

3 Background theory **13**

3.1 Threads and processes . . . 13

3.2 Concurrency and parallelism . . . 14

3.3 Inter process communication . . . 14

3.4 The Raspberry Pi 2 15
3.5 Hardware acceleration 17
3.5.1 Graphics Processing Units 17
3.5.2 OpenGL ES 19
3.5.3 The graphics pipeline 20
3.5.4 GLSL 24
3.5.5 Shader examples 25
3.6 Cameras 26
3.7 Computer vision 27
3.7.1 Image processing 27
3.8 SURF 28
3.8.1 SIFT descriptors 33
3.9 UAV laws and regulations 34

4 Module design **35**
4.1 System assumptions and requirements 35
4.2 High level design 37
4.2.1 The hardware abstraction layer 38
4.2.2 System submodule descriptions 38
4.3 Summary 42

5 Module implementation **43**
5.1 Image sampler 43
5.1.1 Calculating integral images 45
5.2 Window interface 47
5.3 Shader programs 48
5.4 High level computations 52
5.5 Control interface 54
5.6 Summary 54

6 Summary and Recommendations for Further Work **55**
6.1 Summary and Conclusions 55

6.1.1 Conclusion 56
6.2 Test results 57
6.3 Recommendations for Further Work 59
6.4 Short term 60
6.4.1 Finishing the remaining submodules 60
6.4.2 Scale invariance 60
6.4.3 Proper testing 61
6.5 Medium term 62
6.5.1 Concurrency on the CPU 62
6.6 Long term 62
6.6.1 OpenCL 63

A Acronyms **65**

B Code **67**

Bibliography **72**

Chapter 1

Introduction

UAV's are currently being used for many different tasks that require gathering information from the air. They are mobile, and can cover large areas in relatively small amounts of time. They are also cheap compared to manned aerial vehicles and can access places that are hard to reach by land, or are simply too dangerous for humans to traverse.

Search and rescue (SAR) operations is one category of problems that could benefit greatly from the use of UAV's. They could assist the search crew on the ground by efficiently scanning large areas for people that are lost or injured in order to reduce the time and resources required by a SAR operation[Gamnes (2014)].

The fact that the human population is increasing steadily and outdoor activities are quite popular, implies that more and more people will get lost or injure themselves in the wilderness in the future. Hence promoting the importance of efficient SAR crews equipped to handle the demanding tasks that are SAR operations.

In order to perform such a task the UAV will require a module for gathering information from the world around it, one way of doing this is to sample the surroundings using a camera and process these images, looking for patterns matching the target.

The goal of this project is to design a robust, portable and cheap vision module for detecting humans from the air suitable for use in real-time applications.

1.1 Background

This master thesis will continue work on the efforts to build an affordable UAV to be used in SAR operations. It is a continuation of the master thesis Autonomous Unmanned Aerial Vehicle In Search And Rescue conducted during the spring of 2013 [Hammerseth (2013)], the project thesis Autonomous Unmanned Aerial Vehicle In Search And Rescue - A Prestudy conducted during the fall 2013 [Gamnes (2013)], the master thesis Utilizing Unmanned Aerial Systems in Search and Rescue Operations conducted during the spring 2014 [Gamnes (2014)], and the project thesis Autonomous Unmanned Aerial Vehicle in search and rescue - Prototyping conducted during the fall 2015 [Pedersen (2015)] at the Norwegian University of Science and Technology.

This thesis will attempt to contribute to these efforts by focusing on the vision module required to detect humans on the ground. The master thesis written by Vegard B. Hammerseth [Hammerseth (2013)], proposed a solution for a vision module consisting of blob detection and eigenface detection on a pair of images, one thermal and and one color image. Using the thermal image to delimit the area on witch to perform more advanced processing makes the detection problem close to trivial. However, thermal cameras are extremely expensive, and even though cheaper alternatives are appearing on the market, the are still very expensive and suffer from very bad resolutions, making them close to useless at long distance object detection (unless multiple cameras are used, but this will increase cost dramatically).

The hardware platform for the control system of the UAS was chosen to be the Raspberry Pi 2 in the project thesis written during the autumn of 2015 [Pedersen (2015)]. The Raspberry Pi 2

will therefore be the hardware platform of choice for the vision module as well. An integrated module will lower the weight and cost of the final UAS, and is preferable as long as its processing power is sufficient.

Problem Formulation

A cheap, robust and portable vision module capable of detecting humans on the ground in real time from a UAV controlled by an embedded system does not currently exist. The purpose of this master thesis is to explore the requirements and limitations of such a module, as well as propose a working design and construct a prototype.

Object detection is an old problem that has been solved for specific situations numerous times [Szeliski (2010)], however, it is a computationally expensive task and performing object detection in real time on an embedded platforms is difficult, especially at long distances.

For a Unmanned Aerial system (UAS) to be usable in SAR operations, a module capable of detecting human targets on the ground is essential. The motivation behind using object detection on images is the fact that color cameras can be obtained easily and relatively cheap and compact alternatives are available. The increase in computational power on embedded platforms as well as the possibilities for hardware acceleration of certain algorithms also promote the use of higher level computer vision.

Although this module will be constructed with UAV's and detection of humans in mind, it could potentially be used in any application that requires detection of objects using color or grayscale cameras.

Literature Survey

The literature studied is treated in detail in chapter 2. Its main purpose is to get familiar with previous work done in the field of computer vision and to get familiar with existing embedded solutions. It was also necessary to acquire knowledge about the different fields required to design and construct the module.

What Remains to be Done?

The module will be implemented from scratch in order to support hardware acceleration using OpenGL ES 2.0. This is in order to achieve real time performance while running complex computer vision algorithms on large streams of input data.

The module must be designed, implemented and tested. Design choices such as the interface, internal algorithms to run as well as program structure and paradigm must be chosen.

1.2 Objectives

The main objectives of this Master thesis are

1. To design a module based on computer vision and image processing methods for detecting humans from long distances (witch implies large, high resolution image input).

2. Build a prototype of the module and test it.

1.3 Limitations

One of the major limitations of this project was time. Many of the fields that were required to complete it are rather complex, and a lot of time went into studying and gaining a deep enough understanding of them in order to produce meaningful results.

In order to make the system as affordable as possible, emphasis was put on using free, open source technology, or at least cheap and easy to acquire alternatives. The overall cost of the system should be as low as possible, and the components should not be too difficult to locate and acquire.

Also, the interface to the module was not implemented specifically for the final system. The internals were prioritized due to time constraints, but the interface was made clear so that it can be tailored towards, and integrated with, the final system in the future.

1.4 Approach

In order to get familiar with previous work done on the field, all background theory necessary for the design and construction of the module will be thoroughly studied by reading books, watching lectures, interacting with people familiar with the concepts and examining similar projects. The hardware platform chosen will be examined in detail in order to take full advantage of its resources. Furthermore, the module will be designed with performance and portability in mind, it will then be implemented on the target platform and tested as far as time allows.

1.5 Structure of the Report

This master thesis assumes that the reader is familiar with C programming, but an in-depth understanding of the language is not required. In addition, the reader should posses basic knowledge about embedded systems as well as terms and basic hardware used in general computing.

The rest of the report is organized as follows:

- Chapter 2, Literature study, examines the research methods used, as well as the most rel-

evant literature for the subjects.

- Chapter 3, Background theory, contains information about some of the most relevant background subjects of the project. It is included in order to make the rest of the report easier to understand, as well as to give some insight into the system we are aiming to develop in the long run.
- Chapter 4, Module design, describes the high level design of the vision module and discusses the different design choices made.
- Chapter 5, Module implementation, describes the low level implementation of the module, focusing on the algorithms, math and software interfaces.
- Chapter 6, Summary and recommendations for further work, summarizes the results of the thesis and gives some recommendations for improvements and future work.

Chapter 2

Literature study

This chapter examines in detail the methods and sources used for information gathering as well as why they where used. The most important findings are examined more closely at the end of the chapter.

2.1 Research methods

The internet was used as one of the primary means of information gathering in this thesis. Primarily through the search engine Scopus [*Scopus* (n.d.)] and the archives used for theses at NTNU, Digital Arkivering og Innlevering av Masteroppgaver (DAIM) [*DAIM* (n.d.)]. Google was also used, but since it tends to personalize search results, the information retrieved might suffer from being more subjective than the information retrieved through Scopus and DAIM.

The internet is a great source of information, it is easy and quick to access and therefore efficient at finding large amounts of information fast, those were the primary reasons for it being used as the main method for information gathering during this master project. Finding information fast was an important factor since time was very limited.

On the internet it can be daunting to find credible and up to date sources among the vast amount of information available. Using the aforementioned search engines makes this easier.

In addition to reading articles, online lectures were watched. The primary source of these were Youtube [*Youtube* (n.d.)] as well as the Massachusetts Institute of Technology (MIT) online courses [*MiT* (n.d.)] and the Khan Academy lectures [*Khan Academy* (n.d.)]. When watching lectures online, especially those on Youtube, it is important to verify the credibility of the lecturer and the information given.

Reading literature from the university libraries is also a great source of information. Academic books are often the most reliable sources of information and are generally quite objective and complete.

Talking to people familiar with the subjects at hand is also a great way to get familiar with the required fields. Information gathered through interactions with people and from lectures, can be less objective than good written sources like books and articles, but getting direct answers to important questions can be much quicker than reading through literature.

2.1.1 Verifying sources and information

It is always important to verify all sources and information. People are often wrong and articles can be outdated or, if the source is not reliable, wrong. Using the VIKO [*VIKO* (n.d.)] guidelines for finding relevant and credible literature can be very helpful, and all sources should be assessed based on credibility, objectivity, accuracy and suitability *VIKO* (n.d.)].

One way to verify information is to cross check it using multiple independent sources, or verifying them through experiments. The latter can take up a lot of time and resources however, and verification through multiple sources is therefore preferred in this case.

Using sources that have a standing reputation as credible is also a good idea, and university

libraries and written books can be good sources of trustworthy information.

2.2 Related work

In the field of computer vision, a tremendous amount of work has gone into developing algorithms and methods for object recognition. These methods have been used in many previous projects similar to the one at hand. The most important ones are listed here.

2.2.1 OpenCV

OpenCV is an open source library for computer vision and machine learning tasks. The library is heavily focused on performance and the algorithm implementations are heavily optimized for real time performance [*OpenCV homepage* (n.d.)]. OpenCV is also able to take advantage of hardware acceleration using graphic processing units (GPU's) through OpenCL [*Khronos group OpenCL* (n.d.)] or through Nvidia's CUDA compatible GPU's [*Nvidia CUDA* (n.d.)]. Unfortunately, many of the simple, integrated GPU's commonly used in embedded systems do not yet support the OpenCL API, and development boards with Nvidia GPU's that are suitable to also be used as flight controllers are uncommon and quite expensive [*Nvidia Jetson* (n.d.)].

Nevertheless, OpenCL is an extremely potent library, and although it does not have an interface to OpenGL ES 2.0, it might become feasible for beeing used in the future if OpenCL compliant GPU's become more widespread..

2.2.2 SHIFT

SHIFT is an example of the current work being done in order to enable drones to be used in computer vision applications. It is a hardware add on that can be attached to drones in order to give them computer vision capabilities, and despite not being finished yet is a good example of the potential solutions the future might hold. It is not expected to be cheap however (currently,

preorder are at 599 USD) , and as such, a specialized solution using open source software and hardware might still be cheaper and more practical [*3DR SHIFT* (n.d.)].

2.2.3 Object detection algorithms

There are a myriad of algorithms designed for performing object detection, recognition and tracking. Many of them are listed in the book [(Szeliski 2010)], and related papers. In this thesis we will mainly consider the SURF algorithm discussed in detail in chapter 3.

In addition, the thesis [Timothy B. Terriberry (n.d.)] to a large extent implements the core system of the vision module designed in this thesis, however, it is implemented on OpenGL 3.0 compliant hardware, and as such is not suitable for embedded systems.

Chapter 3

Background theory

This chapter will discuss the background theory used throughout the thesis. Its main purpose is to make the following chapters easier to understand and to help justify certain design choices.

3.1 Threads and processes

In computing, a process is a single program being executed that has its own memory context, this memory is not directly accessible by other processes. A thread is a single thread of execution, a process may contain multiple threads and they all share the same memory address space [Stallings (2014)].

These terms are important as they allow software to take advantage of parallelism and concurrency, which allows for major optimization in terms of execution time on certain hardware platforms discussed later in the chapter.

In this project, POSIX [*POSIX base definitions* (n.d.)] will be used in order to create and manage threads. POSIX is a set of standards managed by the IEEE [*IEEE homepage* (n.d.)] organization that defines a standardized implementation of certain API's, among them a thread library.

3.2 Concurrency and parallelism

Concurrency refers to the potential of a task to be split up into multiple independent threads of execution and then execute these threads independent of one another. On systems possessing multiple processing cores this means two or more tasks (processes or threads) may execute at the same time on two or more different cores, thus speeding up execution time. When parts of the task executes on different cores at the same time, they execute in parallel.

If the system does not posses multiple cores or the software in question is not written as to take advantage of multiple processing cores, the system may still give the illusion of parallelism by multiplexing (rapidly swapping between) multiple tasks. This will not improve performance however, and is of less interest in this thesis, but is of great interest in applications where the illusion of parallel execution makes the system more intuitive or in the case where multiple processes must perform actions at regular intervals.

3.3 Inter process communication

Inter process communication (IPC) refers to the techniques used to allow different processes who do not share the same address space to communicate with each other. There are many ways to do this, and the most popular are shared memory and message passing [Stallings (2014)]. Neither will be examined in detail in this thesis, but IPC is very relevant to potential future integration of the vision module into a complete system, since a means of communicating with the rest of the system (like the flight controller) will be necessary.

3.4 The Raspberry Pi 2

The Raspberry Pi 2 is a small, affordable computer consisting of a system on a chip (SoC), a graphics processing unit (GPU) and several other peripherals such as buses, an SD card slot and interfaces for power and pin connectors.

The Raspberry Pi 2 was chosen as the base hardware platform for the system in the project thesis written during the fall of 2015 [Pedersen (2015)], and will be used as the base platform in this thesis as well based on the findings of the aforementioned project report.

Hardware shields for the Raspberry Pi 2 capable of turning it into a autopilot already exists [*Navio online shop* (n.d.)], and integrating the vision module with the pi would thus allow most of the system to run on a single hardware platform, reducing the complexity, cost and weight of the final system.

The full specs for the Raspberry Pi 2 model B is listed below, taken from its datasheet [*Raspberry Pi model 2 datasheet* (n.d.)]:

Raspberry Pi2	
System on a Chip (SoC):	Broadcom BCM2836 SoC
CPU:	900MHz quad-core ARM Cortex-A7
GPU:	Dual Core VideoCore IV, Opengl ES 2.0 compliant.
RAM:	1GB LPDDR2
Operating system:	Runs Linux, boots from SD card.
Flash:	SD-Card, 16GB is standard.
Buses supported (hardware supported):	i2c, UART, SPI
Pins:	40-pins, 27 GPIO pins and +3.3 V, +5 V and GND supply lines.
USB:	4 x USB 2.0 Connector
Camera connector:	15-pin MIPI Camera Serial Interface (CSI-2)
Weight:	Approximately 60g
Dimensions:	85 x 56 x 17mm
Power:	Micro USB socket 5V, 2A
Cost:	30 USD

The most important things to take notice of is the quad core 900 mHz processor as well as the presence of a Opengl ES 2.0 compliant GPU. This opens up the possibilities of hardware acceleration of certain parts of the computer vision algorithms used for object recognition, without heavily sacrificing portability.

3.5 Hardware acceleration

Hardware acceleration refers to using specialized hardware in order to perform certain operations faster than is possible with software running on a general CPU [Sunil P Khatri (2010)]. This section will describe the basics of the hardware acceleration options available on the Raspberry Pi 2 that could potentially speed up the algorithms required for detecting humans.

3.5.1 Graphics Processing Units

A Graphics Processing Unit (GPU) is a specialized electronic circuit optimized for rapidly manipulating memory using many processing cores and parallel threads. They are primarily used in order to create images from symbolic data [John Hennessy (2011)], but are powerful, highly parallel processing units on their own.

The internals of most GPU's are proprietary and thus are not visible to the users. As such, mostly all GPU's are shipped with drivers that implement the OpenGL [*OpenGL homepage* (n.d.)] or Direct3D [*DirectX* (n.d.)] application programming interfaces (API's). In many cases both are supported. Additionally, some GPU's support API's tailored towards GPU computing applications [*Nvidia CUDA* (n.d.)]; [*Khronos group OpenCL* (n.d.)]. In these cases the GPU resources are used to perform general computational tasks often performed on the CPU. GPU's with computing capabilities are not that common in simpler embedded systems however (Although Mali GPU's used in many smartphones, tablets and some modern development boards do support

it [*MALI GPU OpenCL* (n.d.)]), and if the user wants to take advantage of the processing power of such GPU's, one must use traditional API's such as OpenGL and tailor the GPU programs towards more general computational tasks not directly related to graphics. This is referred to as general purpose GPU (GPGPU) programming [John Nickolls (2011)].

Not all tasks will benefit from using a GPU. The main strengths of GPU's are their high number of floating point operations per second (FLOP's) and their memory bandwidth. However, they require the problem at hand to be highly, or at least mostly, concurrent in order to outperform classic general purpose CPU's.

Integrated GPU's

An integrated GPU is defined by the fact that it shares main memory with the rest of the system [John Nickolls (2011)]. The operating system will usually partition the main memory and map a certain portion of it to be used by the GPU. This architecture is also known as a Unified Memory Architecture (UMA) [John Nickolls (2011)]. This solution is the most common one as it is easier to implement, in embedded system such as smartphones and development boards integrated GPU's are almost exclusively used. Even laptops will have integrated GPU's in most cases.

Dedicated GPU's

A dedicated GPU has hardware memory made especially for, and dedicated to, the GPU. Usually only the most powerful GPU's will have dedicated memory and they are not commonly used in embedded systems due to their complexity and high power usage [John Nickolls (2011)].

The main significance of an integrated versus a dedicated GPU to a software developer, is the fact that the developer can usually choose how much memory is to be used by the GPU in the integrated case. This can heavily impact performance if the amount of memory mapped is

not sufficient or too much (leaving less for the CPU).

3.5.2 OpenGL ES

As mentioned in the sections above, developers can make their software take advantage of a GPU by using the drivers supplied by the manufacturer in the form of OpenGL or Direct3D implementations. There are many versions of the OpenGL and Direct3D API's and what versions a GPU can support is dependent on its hardware design. In this thesis we will only consider OpenGL ES 2.0, since it is the API supported by the Raspberry Pi 2 GPU.

OpenGL ES (OGLES) is a cross-platform API for 2D and 3D graphics on embedded systems. It is a subset of the OpenGL (OGL) API which is the most widely used 2D and 3D graphics API [*Khronos group OpenglES* (n.d.)]. By itself it is a specification, defining the interface between software and graphics acceleration hardware. As previously mentioned, companies that make GPU's or video cards may choose to make their hardware according to these specifications and ship them with drivers that are software implementations of the OGLES API. This gives software developers a standardized interface to the GPU, hence making software that uses OGLES compliant drivers portable to any platform with a GPU and driver implementation that complies with the API.

In order for the API to be fully independent of the platform, it has to be operating-system independent and hence, independent of the windowing system used to display the graphics. OGLES therefore includes the EGL specification, witch works as a abstraction layer between the windowing system and the API [*Khronos group EGL* (n.d.)].

Despite this, a small part of the window system will still be dependent on the underlying hardware, and so the module will be designed with this in mind so that portability is not severely

compromised.

Modern OGL (OGL versions 2.X and above) and OGLES 2.X specifications are based on a programmable graphics pipeline, which means software developers can program the hardware pipeline by uploading raw byte strings representing the program they want the pipeline to execute to the GPU. Older versions are based on a fixed pipeline, they cannot be programmed and perform fixed actions on stored data.

This thesis will only consider programmable pipelines as they are flexible enough to be used for tasks not directly related to rendering graphics, like computer vision.

Both modern OGL and OGLES 2.X are perform programmable operations on, and store data in, buffers representing storage in memory. These buffers are referred to as framebuffers (if they are the target buffer being rendered to) and textures/buffer objects (if they serve as input). The data they contain represent fragments or vertices/vertex attributes, fragments are sets of data that are required to compute the final value of a pixel, potentially becoming an image. While vertices and their attributes represent points in 3D or 2D space.

3.5.3 The graphics pipeline

The graphics pipeline in modern OpenGL and OpenGL ES 2.0 refers to the stages of data processing done on the GPU when a draw call is issued. Most of the following information was found on the OpenGL ES 2.0 specifications listed on the Khronos group web page [*Khronos group OpenglES* (n.d.)].

A draw call is a simple OpenGL ES function that is called by the user program and triggers the graphics pipeline calculations. When a draw call is issued from the CPU, the GPU will take

as input the currently bound buffer object (witch are uploaded and bound form the program interfacing the GPU through its drivers) containing vertex data and send it through the pipeline, processing it into data (potentially representing an image) and storing it in the currently bound framebuffer.

In addition to the input buffers, data can be passed to the programmable stages (colored green in the following image) as uniforms, representing variables to be used in fragment and vertex calculations. This allows the user to send additional data such as sample images and numerical variables to the shader programs, much like the input to functions in programming languages like C.

Below is an image representing each stage of the graphics pipeline, followed by short descriptions of each stage.

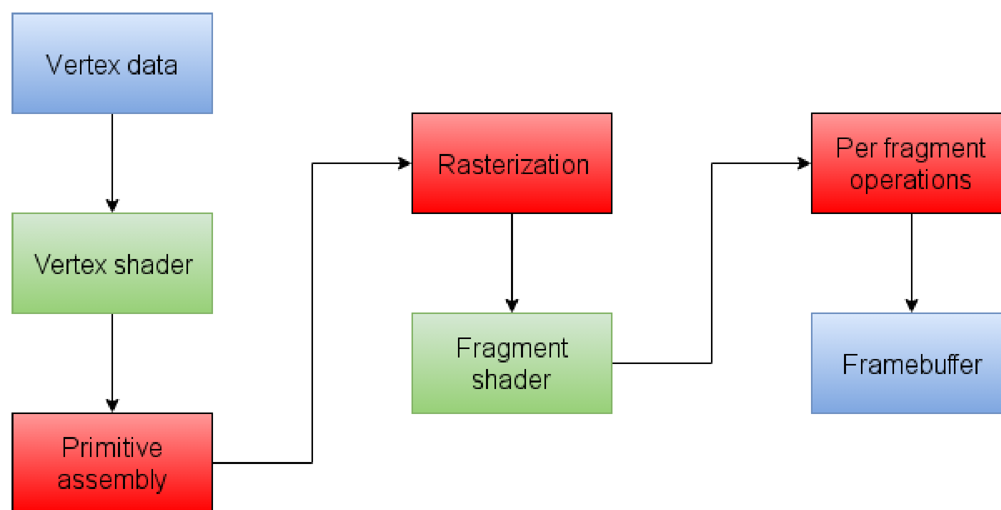

Figure 3.1: The graphics pipeline in OpenGL ES 2.0 compliant hardware. The blue squares are input and output buffers, the green stages are programmable by the user and the red stages are fixed (although the fragment operations have configurable parameters).

Vertex data

The vertex data consists of data representing attributes of points in 3D and 2D space. This data is sent from the CPU to the GPU and bound to an integer handle on the CPU side. The CPU can then issue API calls that bind the buffer to the OpenGL ES context, so that it becomes the input for the graphics pipeline once a draw call is issued.

Vertex shader

The vertex shader is the first stage of the pipeline that can be programmed. It performs operations on vertices stored in the input buffer object such as changing their position, color or orientation.

In this thesis we will only use a single, specialized vertex shader in order to render a 2D quad that covers the whole window (the size of the output buffer witch will match the camera input resolution) and color this with the texture sample taken from the camera before processing the result. We are not interested in 3D graphics in this case and as such, the thesis will not go in depth about the functionality of vertex shaders.

Primitive assembly

The primitive assembly stage takes as input the vertices from the vertex shader and assembles them into primitive geometry such as lines and triangles. A deep understanding of this stage is not necessary as we are not interested in 3D graphics in this thesis and its main job will simply be to assemble two triangles from four vertices representing a quad covering the window.

Rasterization

The rasterizer takes as input the primitives from the primitive assembly stage and rasterizes them, witch refers to filling the primitive with small squares representing fragments and calculating the position (through interpolation) of all the fragments that are contained by each primitive. The number of fragments will be equal to the number of buffered pixels in the output framebuffer witch in this case will be equal to the resolution of the input sample image taken from the camera.

Fragment shader

The fragment shader is the most important stage for the application we are interested in. It is the second and final programmable pipeline stage in OpenGL ES 2.0. It takes as input the rasterized fragments from the previous stage and performs per-fragment calculations resulting in data that can be output to the next stage. The fragment shader is where most of the programmable GPU calculations take place in this application, and as such, all GPU side programs used for image processing and computer vision algorithms in the vision module will be implemented as fragment shaders.

Per fragment operations

Per fragment operations is the stage that perform advanced fragment tests such as depth testing and stencil testing. The tests and operations at this stage will not be used as they are primarily used for 3D graphics and special effects (such as determining occlusion based on fragment depth, allowing the pipeline to discard fragments that are behind other fragments).

Framebuffer

The framebuffer is the output storage located in video memory that stores the output from the graphics pipeline. It can contain multiple buffers that are rendered to at the same time, but in this application we will only use a single color buffer for the framebuffer targets. These can be bound to texture uniforms at later times and used as inputs to other shaders.

3.5.4 GLSL

The OpenGL Shading Language is the language used to program the shader pipeline of OGLES 2.X compliant hardware [*GLSL documentation* (n.d.)]. It is derived from C, and a programmer fluent in C will most likely find it easy to understand code written in GLSL.

In addition to the standard C data types, GLSL also specifies additional types for storing vectors and matrices. It also describes special qualifiers used to specify things such as inputs and outputs between different graphics pipeline stages as well as he precision of the data stored.

It is important to note that even though they have the same name, the GLSL specification for OpenGL ES is slightly different from the specification in OpenGL.

The following section shows an example of a vertex shader that runs during the vertex processing stage and a fragment shader that runs during the fragment processing stage of the graphics pipeline. Both are minimalistic and are tailored towards rendering a screen aligned quad covered with an image stored in video memory, thus they can be used to display images stored in video memory to the screen.

3.5.5 Shader examples

Vertex shader for quad rendering

```
/* The attribute qualifier specifies that this is an attribute to be read as input
   from the currently bound buffer object. In this case, it represents the
   position of a vertex, simply a point in 3D space. */

attribute vec4 position;

/* The varying qualifier, when used in the vertex shader, specifies that this is
   an output variable to be used as input to the next shader stage. */

varying vec2 frag_tex_coord;

void main(){
    /* This variable will be linearly interpolated in order to produce coordinates
       for sampling textures at positions matching the coordinates of the
       corresponding fragment in the fragment shader. */

   frag_tex_coord = position.xy * 0.5 + 0.5;

    /* gl_Position is an inbuilt vertex shader variable that is used to
       interpolate the position of fragments in the fragment shader. This variable
       must be written to or undefined behavior will occur. */

   gl_Position = position;
}
```

Fragment shader for coloring a quad with a texture (image stored in video memory)

```
/* The varying qualifier, when used in the fragment shader, specifies that this is
   an input variable passed from the previous shader stages. */

varying highp vec2 frag_tex_coord;

/* A sampler is a handle that allows the fragment shader to read in and sample
   data from texture images stored in video memory. Here it represents a camera
   sample stored in vram. The highp qualifier specifies hight precision data, and
   the uniform qualifier means that the value of this handle will be specified by
   the program maintaining the openGL ES context. */

uniform highp sampler2D sample_texture;

void main(){
       /* gl_FragColor is an inbuilt variable at the fragment shader stage that
          represents the final color of a pixel in the output buffer. */
     gl_FragColor = texture2D(sample_texture, frag_tex_coord);
}
```

3.6 Cameras

A camera is a device that samples light in order to produce a array of data representing an image [Milan Sonka (2007)]. There are many types of cameras and witch one to use will greatly affect the final performance and properties of the system.

Since the final choice of a camera is very dependent of the rest of the system the vision module will be a part of (primarily due to weight limitations) [Gamnes (2014)];[Hammerseth (2013)], a camera module will not be chosen in this thesis, and focus will be put on making the camera interface modular and easy to configure for new camera sensors.

3.7 Computer vision

Computer vision is a broad field that aims to provide computers with the means to analyze and understand the contents of input given by visual systems, usually cameras, by processing visual data into higher order symbolic representations [Szeliski (2010)]. In many ways it can be seen as the opposite of the process carried out by the modern graphics pipeline in most GPU's which aims to produce images from symbolic data.

3.7.1 Image processing

Image processing is central to all computer vision algorithms, and refers to using mathematical operations on input images in order to produce new output images or a set of data parameters [Milan Sonka (2007)]. Often these operations are required in order for higher level computer vision algorithms to produce meaningful results, examples of this are noise reduction filters and blurs [Milan Sonka (2007)]. In other cases they are used to reduce the complexity of images in order to improve the performance of higher level algorithms, often by getting rid of unnecessary data. An example of this is gray scale conversion witch can greatly reduce the amount of data in a color image in cases where color is not important [Szeliski (2010)].

Image processing tasks often lend themselves well to concurrency, and since they can be quite costly performance wise they are prime targets for being implemented on the GPU.

One important set of image processing operations are filtering operations. They can be performed by convolving a kernel with an input image in order to produce a new output image. Convolution is defined mathematically as:

$$f*g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

Witch refers to convolving the function f with g. In image processing this is done discretely by taking an X by X kernel of values representing the discretely sampled values of the function g. And, at every pixel in the image represented by f = I(x, y), summing the contributions of neighboring pixels weighted by the corresponding values in the kernel.

Convolution will be used extensively during this thesis, and in many cases they can be efficiently implemented on the GPU. The amount of neighboring an OpenGL ES compliant GPU can sample in a single shader execution is limited however, and large kernels(greater than 5x5, found by experimenting on the Raspberry Pi 2, but varies between GPU's) can therefore not be used.

In some cases however, kernels are separable, this occurs if the kernel has a matrix rank of one [Eriksen (n.d.)]. In these cases one can convolve the target image with the middle row and column of the kernel in the x and y directions respectively during separate passes. This reduces the number of samples per pixel needed from X * X to X + X, thereby saving a large amount of pixel accesses.

3.8 SURF

The Sped Up Robust Features (SURF) algorithm [Smith (n.d.)] is a multi stage algorithm used for scale and rotation invariant feature detection and description. It is a predecessor to the Scale Invariant Feature Transform algorithm [Lowe (n.d.)], and was implemented as a faster alternative to SIFT [Kristen Grauman (n.d.)].

The computer vision object detection algorithm used by the vision module will be a close approximation to the SURF algorithm , as such, it will be described in detail in this section.

SURF works by calculating points of interest in the input image, before generating a set of descriptors that can be stored and compared to descriptors calculated from other images in order to match objects represented by the image data. These descriptors are able to match objects that are partially occluded in one of the images, or has been scaled or rotated. This, combined with its speed, makes it an ideal choice for the task of object recognition and detection [Herbert Bay (n.d.)].

With and implementation of an object recognition algorithm, we can process and store the features of images of humans in different poses or images of the parts of a human body, and look for features matching those features in real time. Reporting a detected human if there is a match.

The detailed stages of the SURF algorithm are as follows, taken from [Kristen Grauman (n.d.)], [Herbert Bay (n.d.)], [Szeliski (2010)] and [Smith (n.d.)]. First an integral image is calculated from the input grayscale image. In an integral image, the value of each pixel is equal to the sum of all the pixel intensities of every pixel that has coordinates less than, or equal to, the current pixels x or y value.

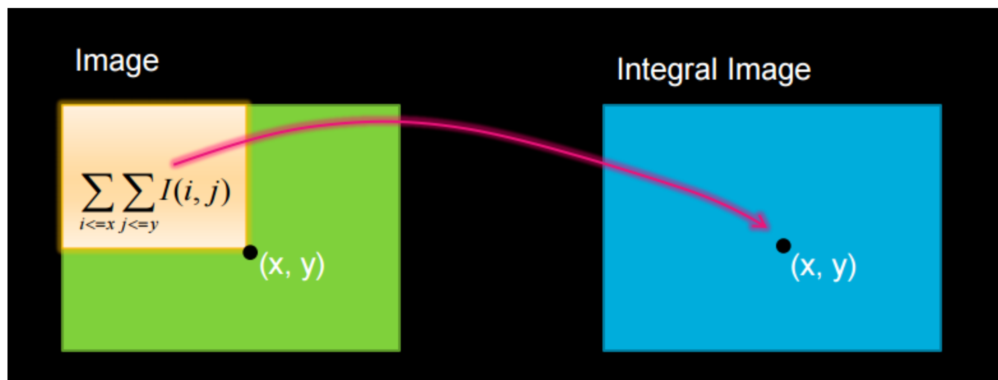

Figure 3.2: Image credit: *Scott Smith.* Graphical representation of an integral image.

An integral image can be easily computed on the CPU using an implementation of the following pseudo code:

```
for(int i = 0; i < image_width; ++i){
   for(int j = 0; j < image_height; ++j){
      II(i, j) = I(i, j);
      if (i != 0){
         II(i, j) += I(x - 1, j);
        }
        if(j != 0){
         II(i, j) += I(x, j - 1);
        }
    }
}
```

This procedure is can be calculated on the GPU using OpenGL ES but it is a rather complex task due to its heavy data dependency on previous calculations,. Using OpenCL or CUDA would possibly alleviate this, but for OpenGL this task is difficult.

The reason for using integral images is the fact that computation of the sum of intensities within a square block of pixels in the original image can be computed from its integral image in constant time regardless of the area over witch pixels are summed. This is expressed through the formula:

$$\sum_{a=u}^{x} \sum_{b=v}^{y} \mathrm{I(a, b)} = \mathrm{II(x, y)} - \mathrm{II(x, v)} - \mathrm{II(u, y)} + \mathrm{II(u, v)}$$

where II(x, y) is a sample taken from the integral image, and I(x, y) is a sample from the original image taken at at the coordinates given by x and y.

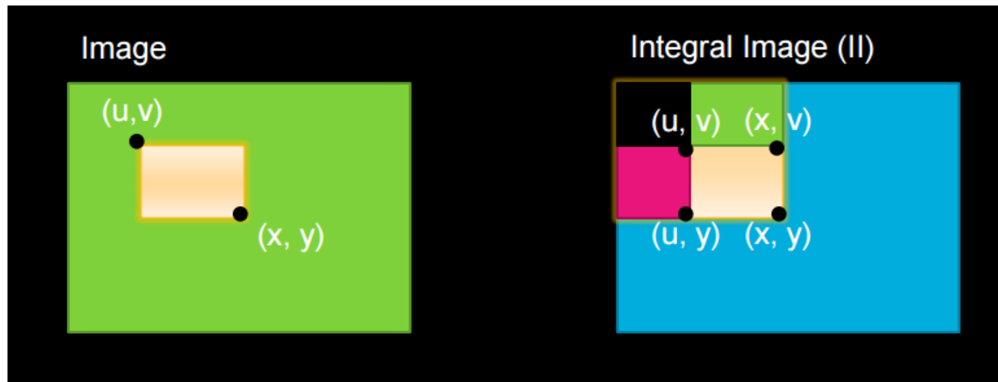

Figure 3.3: Image credit: *Scott Smith.* Graphical representation of the intensity sum over a block in an integer image.

This allows convolutions that have kernels with uniform values (such as box filters) to be done in constant time regardless of the kernel size.

After an integer image has been calculated, the algorithm starts searching for keypoints. Keypoints are points of interest in an image that can be easily localized, that is, separated from their surroundings. Such points have the property that their gradients, witch are discrete representations of the derivative (rate of change in intensity) at the points in question, are rapidly changing in all directions around them.

Calculating the gradient (derivatives) of an image can be done by convolving the image with certain filters. There are a myriad of such filters, and here we will discuss the clever solution used by SURF.

The SURF algorithm takes advantage of the integral image previously calculated in order to approximate a hessian matrix representing the second partial derivatives at a point in the image. The hessian matrix can be written as:

$$H(\bar{p},\sigma) = \begin{bmatrix} I_{xx}(,\sigma) & I_{xy}(\bar{p},\sigma) \\ I_{yx}(\bar{p},\sigma) & I_{yy}(\bar{p},\sigma) \end{bmatrix}$$

Where σ can be approximated as:

$$\sigma \text{ (filtersize)} = \text{(filtersize / 9)} * 1.2$$

$\bar{p}$ represents the x and y coordinate vector for the current image position and σ, represents the standard deviation (the size) of the underlying kernel used to calculate the partial derivatives [Smith (n.d.)].

The SURF algorithm directly calculates the second order partial derivatives in the hessian matrix by using the filters shown in the following figure:

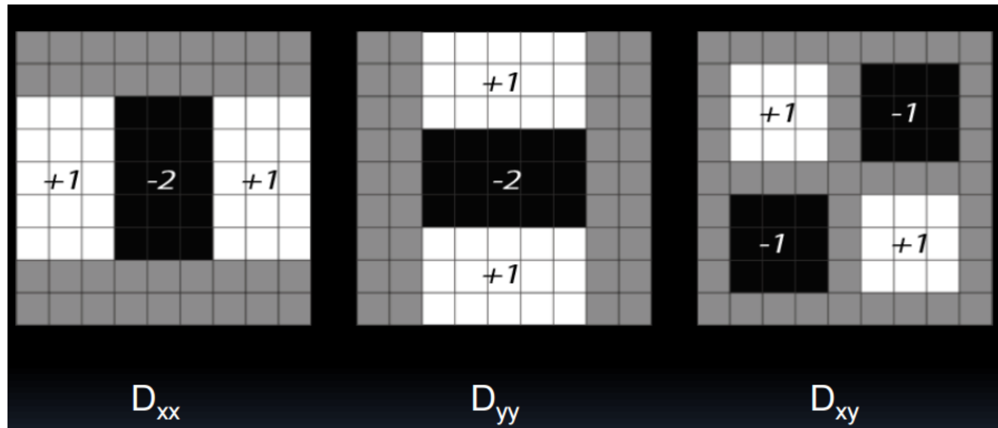

Figure 3.4: Image credit: *Scott Smith.* The kernels used to approximate the derivatives of the Laplacian of Gaussian function used in the SIFT algorithm. Note the large, uniform areas that lend themselves to integral image computations. The gray areas have a weight of 0.

These kernels approximate the Laplacian of Gussian (LoG) function used to compute the second order partial derivatives in the SIFT algorithms [Kristen Grauman (n.d.)]; [Szeliski (2010)]. Using these, the hessian can be quickly computed using the integral image to calculate the intensity sum of the block areas.

Features are then computed by using the filters shown above to calculate the hessian at pixels at different scales (different size kernels) and then searching for points that are maximum

values in the scale space defined by $\sigma and \bar{p}$, such maxima are located by looking at the surrounding pixels in 3x3x3 cubes where the coordinates are x, y and σ. The points that produces the maximum value for the eigenvalues of the hessian matrix at the given point [Kristen Grauman (n.d.)]; [Szeliski (2010)] are considered as interest points. If the maximum value of these hessian eigenvalues are larger than a certain threshold, the point is marked as a keypoint.

The reason for the multiple scale sampling is the desire to obtain scale invariant feature descriptors. These descriptors will match in two different images even if one of the images is drastically scaled down or the target in the image is much farther away.

After the strong keypoints have been localized, the algorithm computes the rotation (and already scale) invariant descriptors of each keypoint by first multiplying the feature region with a kernel representing two haar wavelets [Kristen Grauman (n.d.)] in order to determine the dominant gradient direction in the region. A descriptor window is then generated based on the scale at the level of the current keypoint maxima, the window is rotated to be aligned with the dominant direction and divided into 4 x 4 subregions. Haar wavelets are then multiplied with the points in each region in order to give approximations to the gradient directions in each cell, and four descriptive values are calculated for each point, dx, |dx|, dy and |dy|. This results in 4 x 4 x 16 = 64 values that are stored as the descriptor vector for the current feature.

3.8.1 SIFT descriptors

The SURF algorithm is the target algorithm of choice in this thesis as it is fast, robust and invariant to scale and rotation. However, due to some technical issues encountered during the implementation process, the SURF descriptors were swapped out with SIFT descriptors. These are calculated by putting the gradients into bins representing direction, and using the most populated bins to determine the sample window direction. A 16 x 16 sample window is then rotated to this directional vector and separated into 4 x 4 regions. In each region a new histogram repre-

senting the 16 gradients (each histogram has 8 bins) in each region is calculated and the resulting 8 bins are used as elements in the feature descriptor, resulting in a 128 element vector.

It is worthy to note that the SIFT descriptors are twice as large as SURF descriptors, thus comparing them to other keypoints become more expensive and matching thus takes more time.

3.9 UAV laws and regulations

The flight of drones and UAV's in Norway is subject to national laws and regulations. They will not be discussed in detail in this thesis but some are important as they give a reference framework for the maximum limits on performance. It is illegal to fly UAV's in Norway at heights greater than 120 meters [*Luftfartstilsynet homepage* (n.d.)] based on this the assumption will be made that the vision module does not need to be able to detect its target at distances exceeding 120 meters.

This will be kept in mind as a maximum limit although it is very unlikely that the module will be able to detect humans from this distance unless a camera with a very narrow focus and high resolution is used. The better the camera and processing hardware, the higher the drone can fly and the more area it can cover in shorter amounts of time. However, it only need to perform reasonably better than a human in order to be useful in a SAR operation, and maximizing the height at witch it is able to detect a lost person can be seen as a long term goal.

Chapter 4

Module design

This chapter describes the design of the module, and will examine the different choices made during the design process.

4.1 System assumptions and requirements

The Raspberry Pi 2 specs will serve as the base hardware requirements for the whole system, and as such, the module assumes the underling platform can at least match the performance and specs of the Raspberry Pi 2. It should be noted however, that more powerful and cheaper platforms are already becoming available on the market [*Pine64 homepage* (n.d.)]; [*Udoo homepage* (n.d.)], and although there are currently no major communities developing solutions to use them as flight controllers, they are more than capable of running heavy computer vision applications, and their GPU's make them capable of doing so in real time. For this reason, portability was heavily in focus when designing the module as the possibilities of better and more appropriate system platforms to appear in the future is quite high.

C was chosen as the programming language to be used, as its functionality is sufficient for the task at hand. C compilers are available on most platforms, thus adding to the portability of the module. In addition, the low level API's used all have driver implementations written in C

making them easier to use in the application.

The module will assume that the target platform is capable of running an operating system based on the Linux kernel [*Linux homepage* (n.d.)] and that it has a GPU compliant with the OpenGL ES 2.0 API. In addition the module prototype will be implemented using the Video 4 Linux 2 (V4L2) drivers, and assumes a video sensor compatible with this driver will be used. However, the module will be implemented in such a way that the code interfacing the underlying camera hardware can be easily replaced if needed, as the camera sensor is likely to be swapped out in the final system.

In addition, POSIX threads will be used in order to create threads for parallel execution of certain submodules. As such, the OS must have implementations of the POSIX compliant pthread library available.

Long range detection should also be considered as it is central to the environment the module will be operating in. Detection at longer ranges requires the same minimum density of pixels over an object as the minimum scale at witch the algorithm can detect said object. Thus, longer range detection requires higher resolution (and thus more processing power) or a narrower field of vision (can be adjusted with the camera lens). Because of this, performance is very important in this project, and is one of the main reasons hardware acceleration on a GPU is put in such central focus.

These assumptions are not as limiting as they might sound as most of the modern development boards currently on the market meet these criteria. Also, some of them are becoming compliant with the OpenCL API [*Khronos group OpenCL* (n.d.)] witch is more suited towards general computational tasks (such as computer vision) than OpenGL ES compliant hardware (witch is aimed at performing 3D and 2D graphics computations). Thus, it might be desirable to replace the OpenglES code with OpenCL compliant code in the future.

4.2 High level design

This section will propose a high level design of the module and will attempt to separate its parts into clearly defined submodules with clear interfaces. This design closely resembles an object oriented paradigm with variable structures encapsulating variables and dedicated methods performing operations on these structured blocks of data. This will make integration with the full system easier and will hopefully make it easier to replace parts of the module if needed.

The following figure shows the full high level organization of the submodules in the system.

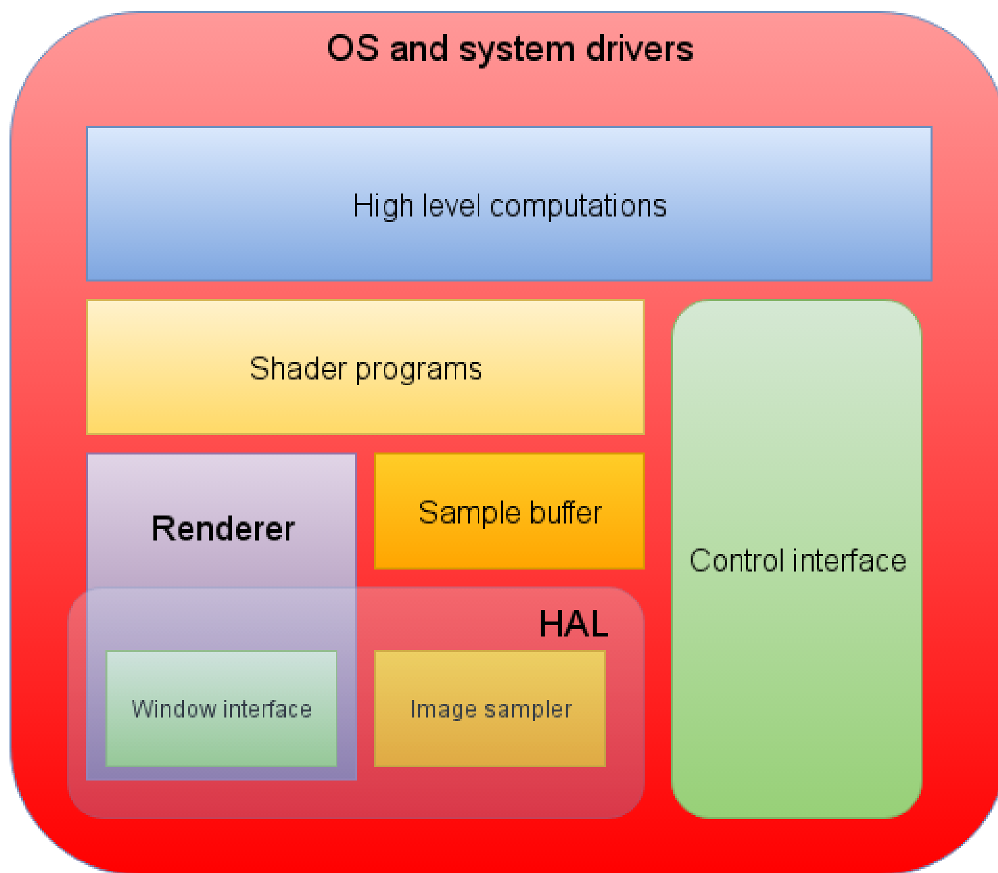

Figure 4.1: The full software stack of the vision module. All submodules are entailed by the operating system and its drivers. The system drivers include the OpenGL ES 2.0 Broadcom drivers and the V4L2 drivers interfacing a camera module.

4.2.1 The hardware abstraction layer

The hardware abstraction layer (HAL) contains the submodules that are specific to the underlying system hardware used. It includes the windowing system and the image sampler, witch are discussed further in the following section.

The main purpose of the HAL is to clearly separate from the rest of the module the parts of the vision module that needs to be reimplemented if the underlying hardware configuration changes. Thus making the system easier to port without notably affecting performance.

4.2.2 System submodule descriptions

This section will describe the different submodules in detail, focusing on the high level tasks handled by each module as well as their interface to the other modules.

OS and system drivers

As mentioned in the previous sections, the module will be designed to run on within a Linux based operating system. The main reason for this is the popularity of the Linux OS in modern embedded systems, most of witch support some form of Linux distribution.

Linux is also an open source, free to use OS, witch helps reduce the cost of the overall system and also allows users to modify the kernel if needed. In addition Linux can be modified in order to meet real-time constraints [*Xenomai homepage* (n.d.)], something that is of utmost importance in a real-time system such as a UAV.

The operating system manages the vision module as a process with its corresponding threads. It provides the module with access to the underlying hardware through system drivers. These include the OpenGL ES 2.0 driver implementations supplied by Broadcom that allows user level processes to access the GPU, as well as the V4L2 drivers that allow access to camera modules.

Image sampler

The image sampler is responsible for interfacing the camera used to collect information about the surroundings. It contains methods for sampling a single image and storing the raw pixel data in system memory so that it can be accessed and processed by the rest of the submodules. It is a part of the HAL as it requires access to drivers specific for the camera sensor used, and in the case that the full UAV system wishes to use a different camera access driver, this submodule must be reimplemented by changing the sampling methods.

Window interface

The window interface is used for displaying the contents of the buffers being rendered into by the GPU. Its main use is for debugging and visualization of data. Although it is not strictly necessary in order for the module to perform its main task of human detection, OpengGL ES requires a window to be initialized in order to function. This is one of the main reasons why it might be desirable to replace OpenGL ES with OpenCL in the future (if it becomes widely supported) as it does not have this limitation and would thus increase portability in addition to its other, performance boosting, advantages [*OpenCV homepage* (n.d.)].

The window interface is a separable part of the renderer submodule discussed in the next section. The reason for this separation is the fact that the windowing system is very platform dependent, and separating it from the main renderer (witch is quite complex on its own) reduces the amount of code that needs to be replaced if the underlying hardware is changed.

Renderer

The renderer is responsible for handling all the framebuffers used as rendering targets (outputs) and inputs to the shader programs, as well as the uniform variables passed to the shaders. It also contains methods for rendering to the display through the window interface it contains. In addition, the renderer takes care of the OpenGL ES rendering context, and keeps track of the global state of the GPU pipeline.

Keeping the memory buffers and uniform variables batched together in a single submodule makes the module variables easier to manage and makes the system more maintainable by avoiding spreading out the variables. It helps keep the interfaces clean and avoids spaghetti code.

Sample buffer

The sample buffer is the submodule responsible for handling the copy of the image sample located in video memory. It takes input from the image sampler and copies this into a texture buffer located in the memory dedicated to the GPU. This is necessary as it is currently not possible to copy the image sample directly into video memory from the camera, as the sample needs to be uploaded through the OpenGL ES API call for creating textures, witch takes a pointer to system memory as input.

It should be noted that this is a major bottleneck in the vision module, as the full image sample needs to be loaded into memory twice, once from the camera IO buffer into system memory, and then from system memory into video memory. Finding a way to upload the sample directly into video memory would make the sample buffer submodule obsolete and greatly increase the module performance.

Shader programs

This submodule consists of all the programs performing mathematical and algorithmic operations on the GPU. This submodule takes as input the textures and buffer objects managed by the renderer and the sample buffer, before processing them in order to generate new image buffers representing data useful for object recognition. This stage will contain all image preprocessing computations as well as some GLSL implementations of computer vision algorithms.

For performance reasons as much functionality as possible should be implemented and executed at this stage as long as the tasks in question are suited for the parallel processing environment on the GPU.

High level computations

The high level computations refer to the final stages of the computer vision algorithms running on the CPU. These algorithms take as input the output images from the shader programs and perform the final object recognition tasks. Particularly stages of the computer vision algorithms that are sequential in nature or does not perform significantly better on a parallel processing unit like the GPU will be handled by this submodule.

Control interface

The control interface is responsible for handling IPC and thereby enabling communication with other processes. It receives input from the high level computations, and makes the relevant information accessible by the rest of the system the vision module is designed to be a part of. In addition, it can read information passed to the vision module from other processes, thus allowing the user of the whole system to control the vision module.

4.3 Summary

This chapter has proposed a high level module design focusing on portability, modularity, performance and maintainability. Attempting to clearly divide the module into several submodules with defined high level tasks in order to make it easier to integrate the module with the final system, as well as replace parts of it if needed. It is likely that new and better technology will become available in the future, and the module might need to be adapted to new hardware platforms or new external cameras and/or Linux distributions with different low-level windowing systems.

Chapter 5

Module implementation

This chapter will go through the low level implementation of each of the submodules in the vision module. The theory and underlying structure of each will be examined in detail and the algorithms implemented on the GPU and CPU will be described.

The renderer and the sample buffer will not be discussed further in this chapter, as their functionality was described in chapter 4, and their implementations are rather straight forward and trivial. They function merely as containers for buffers and variables and their methods contain mostly OpenGL ES standard API calls.

5.1 Image sampler

The image sampler is responsible of taking image samples from the camera sensor and converting them to raw byte arrays stored in system memory. The following code snippet shows the C interface to the module:

```
typedef struct HAL_image_sampler{
    GLint driver_file_descriptor;
    uint8_t* buffer;

#if USE_INTEGRAL_IMAGE
    uint32_t* ibuffer;
#endif
    GLint buffer_width;
    GLint buffer_height;

    GLuint current_sample;
}HAL_image_sampler;

GLint init_image_sampler(HAL_image_sampler* sampler);
GLint destroy_image_sampler(HAL_image_sampler* sampler);
GLint init_mmap(HAL_image_sampler* sampler);
GLint sample_image(HAL_image_sampler* sampler);
GLint query_capabilities(HAL_image_sampler* sampler);
GLint configure_settings(HAL_image_sampler* sampler);

#if USE_INTEGRAL_IMAGE
GLint sample_grayscale_image(HAL_image_sampler* sampler);
GLint sample_integral_image(HAL_image_sampler* sampler);
GLint convert_sample_to_grayscale(HAL_image_sampler* sampler);
GLint convert_sample_to_integral_image(HAL_image_sampler* sampler);
#endif
```

The image sampler currently contains a handler to a driver file descriptor that is used to interface the underlying camera drivers. In addition, it contains a pointer to a buffer that will hold the sampled image data. This pointer is memory mapped to an address returned by the underlying camera driver when the sampler is initialized.

In addition, the sampler contains some conditional code wrapped in the USE_INTEGRAL_IMAGE macro. This code was used to test the performance of integer image calculations on the CPU, more specifically, the image sampler was responsible for calculating an integer intensity image from the sample returned by the camera and storing it in the uint32_t buffer. This buffer has a larger word size due to the potentially large values of the highest coordinate pixel in the resulting integral image. This potential for overflow in lower word size buffers is one of the main reasons the integral image was made more difficult to implement on the GPU due to the potentially large rounding errors that would occur during range mapping. This was again one of the main reasons for the simplified object detection algorithm used in the final module as SURF looses its appeal without the possibility of rapidly calculating image convolutions.

5.1.1 Calculating integral images

The SIFT algorithm is heavily dependent on the computation of integral images in order to calculate the hessian matrices used to localize feature keypoints. Without integral images, calculating the hessians at different scale levels in order to make the SIFT algorithm scale invariant becomes infeasible, as the cost of convolving large kernels with the sample image is not possible in most OpenGL ES 2.0 compliant GPU's (or modern, powerful CPU's for that matter).

However, the calculation of the integral images, as mentioned in the section describing the SURF algorithm, is possible, but very difficult to do on the GPU due to heavy data dependence of the calculations and the limitation in word size of the GPU texture buffers, leading to rounding errors of up to 2%, even in OpenGL 3.0 buffers [Timothy B. Terriberry (n.d.)], as well as the need to transform data representations and store them in intermediate buffers, increasing memory usage and code complexity. This would potentially be heavily improved by using a API aimed at computational GPU programming (such as OpenCL or CUDA). This has already been done in the publication [Bilgic B. (n.d.)] and improves the calculation time of the integral image by a factor of four as compared to the CPU, witch is good, but nowhere near the improvement of

less data dependent image processing tasks. While using a computational API makes the implementation of the integral image calculation easier, it is not a requirement as it is still possible to do the computations on a OpenGL compatible GPU, as proven in the paper cited here [Timothy B. Terriberry (n.d.)]. However, this project uses the OpenGL 3.0 API witch has access to higher resolution buffers, reducing the problem of rounding errors and making data management easier. And the implementation on an OpenGL ES 2.0 compatible card might not be feasible.

Due to the complexity and uncertainty of this implementation, it was decided to forgo scale invariance for now and use sobel filters [Szeliski (2010)] to calculate the hessians at one scale level. In the next chapter of this thesis, solutions to deal with this limitation will be proposed.

It should also be mentioned that a sample implementation of the integral image calculation on the CPU side was tested in this project, but it increased rendering time by around 200ms for a small 700x700 resolution image, thus making real time computation infeasible for larger images witch are required to do long range detection of objects.

A large amount of work was done during this project in order to attempt to make the integral image solution work on the GPU, or to find some other way in witch to calculate it quickly without much success. One potential solution though, could be to allocate two buffers and dedicate a CPU core to continuously convert images into the buffer that was read last using the pthread library. Even if this results in a speedup, one will still have to deal with the rounding errors and potentially reduced robustness of the SURF algorithm under these circumstances. And finding solutions for this is not trivial and worthy of a publication in and of itself.

5.2 Window interface

The window interface implementation in C is shown in the code snippet below. It is rather simple and contains a single method for initializing a Hal_window instance. This instance contains a native_window variable that functions as a handle to the standard interface to the window used by the OpenGL ES API. The window contains the standard framebuffer to witch all information to be displayed on the screen will be rendered. This is useful for debugging and visualization of data, but not necessary in the final system implementation.

The variables encapsulated by the WINDOW_SYSTEM macro are the platform dependent variables specific to the Raspberry PI 2. Dispmanx is the low level windowing system used, it is fast, but is getting outdated and severely lacks in documentation [*Videocore Dispmanx* (n.d.)].

Behind the scenes, the window submodule uses the EGL API in order to bind a platform specific window to the platform independent EGLNativeWindowType handle. And replacing the Dispmanx windowing system requires that the user reimplements the init_window method so that it binds the new window to the EGL handle.

```
typedef struct HAL_Window{
   uint32_t window_width;
   uint32_t window_height;
   uint32_t screen_width;
   uint32_t screen_height;
   EGLNativeWindowType native_window;

#if WINDOW_SYSTEM == DISPMANX
   EGL_DISPMANX_WINDOW_T dispmanx_window;
   DISPMANX_DISPLAY_HANDLE_T dispman_display;
      DISPMANX_ELEMENT_HANDLE_T dispman_element;
```

```
   DISPMANX_UPDATE_HANDLE_T dispman_update;
   VC_RECT_T dst_rect;
   VC_RECT_T src_rect;
#endif
}HAL_Window;

EGLint init_window(HAL_Window* window);
```

The window submodule's primary function is to make the underlying windowing system easy to replace and add to the portability of the vision module.

5.3 Shader programs

The shader programs are the GPU implementations of the tasks in the vision module that benefit from the parallel execution offered by graphics pipeline.

The following code snippet shows the internals of a single shader instance:

```
typedef struct Shader{
    GLuint program;

    GLint uniforms[MAX_UNIFORMS];
    char* uniform_names[MAX_UNIFORMS];
}Shader;

GLint init_shader(Shader* shader, char* vertex_shader_name, char*
    fragment_shader_name);
GLint init_uniforms(Shader* shader);
GLuint create_shader(const char* filename, GLenum type);
GLint use_shader(Shader* shader);
GLint load_uniform_location(Shader* shader, GLuint uniform_index);
GLint print_log(GLuint object);
```

```
GLint load_shader_from_file(Shader* shader, char* vertex_shader, char*
    fragment_shader);
```

A shader instance stores a single integer handle that represents the program loaded into GPU memory using the load_shader_from_file method. The uniforms and uniform names keep track of the handles of the variables that can be uploaded to the shader represented by the program handle. The renderer can query this handle and use it to upload data to the GPU variables they represent.

The shaders are initialized by passing a string to the load_shader_from_file method representing a text file with the corresponding name stored in the src/shaders/fragment/ folder. The method then handles uploading, compilation and error checking of the shader program.

Currently, the shader programs submodule consists of a pipeline of shaders called in succession while rendering into, and taking input from, the framebuffers managed by the renderer and the sample stored in the sample buffer. The current computational sequence is shown in the following diagram:

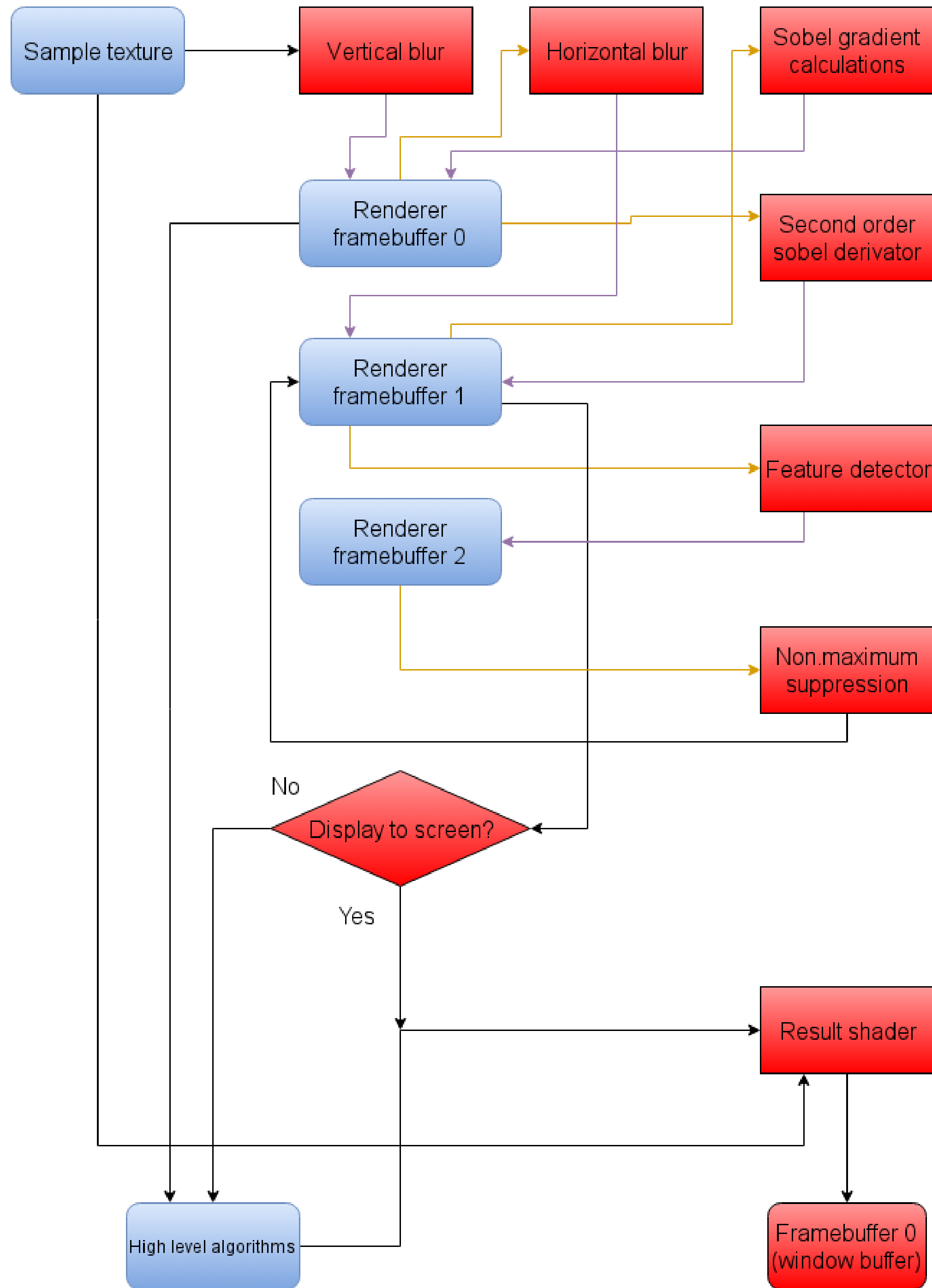

Figure 5.1: The sequence of shader programs called and the respective framebuffers stored in the renderer they use as inputs and outputs.

The first two stages represent a separable, Gaussian blur filter convolution used to remove noise. Since the filter is separable, separating the process into two stages allows us to use a big-

ger kernel while at the same time drastically reducing the number of texture samples required at each pixel. Greatly boosting performance.

The next stage calculates the first order gradients in the x and y direction by convolving the image with two 3x3 kernels known as sobel filters, they are weighted as follows:

$$dy(x,y) \approx \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad dx(x,y) \approx \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The Sobel operators are also separable, and this stage can also be split into two. An increase in performance is unlikely however, as the cost to send a new draw call to the shader pipeline is probably more than the few texture look ups saved as the kernels are already quite small.

It is also important to note that Sobel filters are very sensitive to noise, this gives them a "hidden" performance cost due to the fact that the Gaussian blur applied earlier is required for the Sobel operators to produce good results. Other, less noise sensitive filters might produce better responses without requiring a smoothing blur, and as such, might be cheaper in the whole scheme of things. Still, Sobel filters where used here as a noise reduction Gaussian filter gives clearer responses and thus makes debugging during development easier. In addition, the Sobel operators are currently placeholders and will hopefully be replaced by better, scale invariant, gradient calculators in the future.

After the first partial derivatives are calculated, the next stage runs the Sobel operators on the output from the previous stage. This produces a image representing the second order partial derivatives of the original image. These values can then be used to calculate the hessian matrix.

After the second order derivatives are calculated, the feature detection stage runs. Since we are not using scale space maxima in this case, this stage simply computes the hessian matrix according to the formula given in chapter 3, and discards all points that have small hessian eigenvalues. This is equivalent to calculating the following formula:

$$det(H(x,y)) - \alpha * trace(H(x,y)) > t$$

Where α is between $0.06 - 0.06$ [Kristen Grauman (n.d.)]. This formula is slightly modified as we do not take into account the scale space, and the Gaussian weighting is dropped (since we do not care about scale space invariance at this moment).

Once the thresholded points are calculated the next shader stage runs non maximum suppression in a 3x3 neighborhood around each potential feature, discarding the feature if any of its neighbors are larger in value than it is (that is, has larger eigenvalues).

Additionally, a final stage that displays buffer results on the screen can be enabled to visualize the data. This stage should be turned off when the module is used for detection as it slows down the iteration time.

5.4 High level computations

The high level computations consist of the final computations of the feature descriptors that run on the CPU. This stage reads back the values stored in the image representing detected keypoints and the image containing the gradients of the image.

The following diagram shows the execution flow of the high level computations stage:

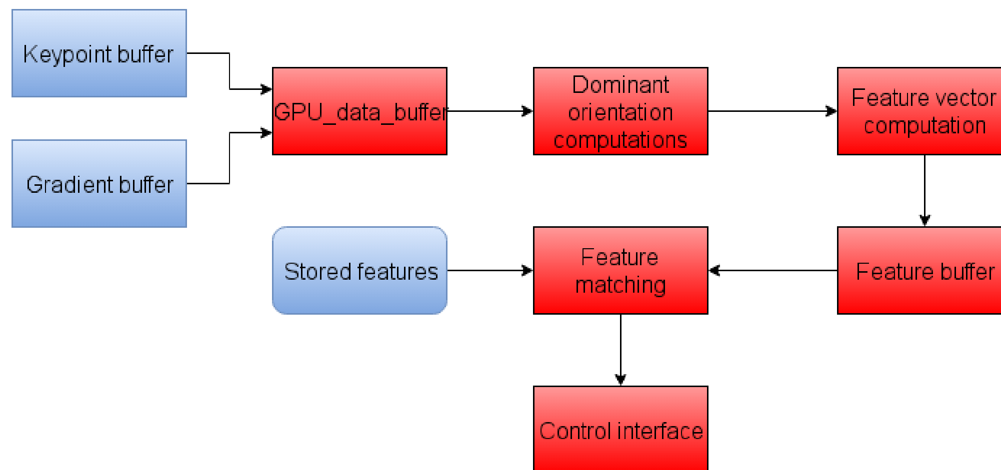

Figure 5.2: The sequence of high level computations that calculate the keypoint descriptors and perform feature matching.

During the first stage, data is read back from the final rendering target framebuffers in the renderer. This is a rather slow stage, and takes on average 80 ms on a 700x700 image, as such, this stage (and the entire high level computation stage for that matter) should be ran in its own thread such that the rest of the system can continue the next iteration of computations. For the sake of keeping the module clean and easier to debug during development, this has not been done yet, but should be implemented as soon as possible as it is a rather simple extension and can greatly speed up the module iteration time by effectively pipelining the submodules.

At the second stage, the orientation of the descriptor window is calculated by using the SIFT method described in chapter 3. The stage immediately following computes the SIFT futures and stores them in a feature buffer that can be saved for later comparisons, or they can be compared to a stored set of features in order to determine if a match has been found.

Using SIFT instead of SURF descriptors, doubles the time it will take to match a feature to features in other images due to the large feature vectors. SIFT features are easier to compute however, as they do not require lots of convolutions with oriented haar wavelets. In the case that the full SURF algorithm is implemented and integer images are successfully implemented,

using SURF features is preferable due to their smaller feature vectors and robust matching rate [Szeliski (2010)].

5.5 Control interface

The control interface is responsible for sending messages between the other processes running on the system as well as controlling the state of the vision module, allowing external users to control the module by sending commands to it through shared memory. The control interface currently runs in parallel with the rest of the system in its own thread, this is mainly for convenience, allowing the control interface to continuously monitor user inputs as it is not heavy on performance.

5.6 Summary

In this chapter, the details of the underlying system implementations have been discussed. It should be noted, that not all of the submodules have been fully implemented, and some of them (notably the last stages in the high level computations, and the control interface) however, the framework for the high level processes has been laid out, and their implementation is mostly based on well documented findings and tasks that have been implemented in numerous other systems.

The reasons for the incomplete modules are summarized in the last chapter, but were mainly due to priorities of witch parts of the module to spend time on.

Chapter 6

Summary and Recommendations for Further Work

This chapter will summarize what has been done and the experiences gained throughout the thesis work. The results will be presented and discussed, and recommendations for further work will be given based on the problem formulation, objectives and the remaining tasks that need to be addressed in order to complete, and in the future, improve the vision module.

6.1 Summary and Conclusions

In this thesis, a vision module to be used for detecting humans from UAV's used by SAR crews on the ground has been designed, and the submodules responsible for feature detection and extraction, as well as rendering and managing buffers and memory transfers, has been implemented and tested.

Algorithms for object detection were examined, and the SURF algorithm was chosen as the core detection method for locating humans on the ground. A large portion of the algorithms and calculations needed where implemented on the Raspberry Pi 2 GPU using the OpenGL ES 2.0 API in order to improve performance and enable real time performance of the module.

Although the design of the module was completed, the full implementation of all the modules was not completed in time. The main reason for this was the fact that the integral image calculation required by the SURF algorithm proved to be very difficult to implement on a Open-

glES 2.0 compatible GPU. In order to potentially solve this, the high level processing modules aimed at handling the future descriptors in an image where down prioritized, as they are not very difficult to implement and the task has been completed and written about in numerous papers and projects in the past. A lot of focus and effort went into attempting to implement an efficient integral image computation, as success here would be huge for the calculation of feature points as well as enabling scale invariant features.

6.1.1 Conclusion

Althought little success was found attempting to do integral image calculations on the GPU, most of the methods proposed in the paper [Timothy B. Terriberry (n.d.)] was tried out, and the knowledge that it is possible on certain GPU's is good to know for future improvements, as the module can possibly be ported to more suited platforms. So far, the conclusion is that the integral image calculations should be done on a multicore CPU, attempting to take advantage of multiple CPU cores is probably the way to go.

The performance gains of the GPU implementations of the other image processing computations were substantial, and the results look promising with respect to enabling long range human detection in real time in the future.

The following section shows some sample images and results from the GPU computations, and list some timer benchmarks found during the testing of the submodules.

6.2 Test results

This section shows a couple of sample images taken at different stages of the shader program sequence. Their original sample size and rendering time is listed in the image descriptions. It should be noted that the rendering time does not include the time it takes to sample an image into main memory, but it does include the time used to move it into video memory. The average time to sample a 700x700 image into main system ram as approximately 20 ms. The feature point stage includes all preceding shader stages listed in chapter 5. The same is valid for the Sobel result.

Figure 6.1: The gradients calculated by the Sobel operators. This image was rendered in 700 x 700 resolution in 10.46 ms and has been cropped for display here. The colors are due to value mapping between 0.0 and 1.0 in order to fit into OpenGL ES texture ranges.

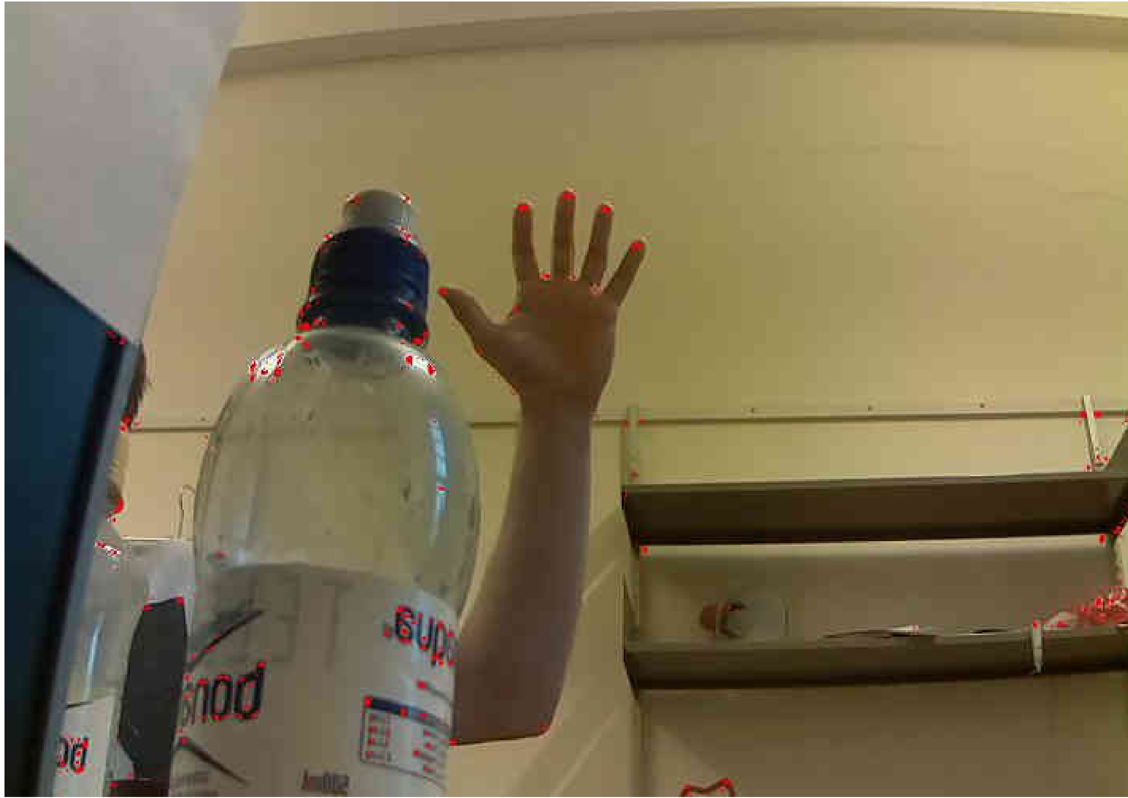

Figure 6.2: Feature points detected using the GPU shaders, points marked as red dots. This image was rendered in 700 x 700 resolution in 39.14 ms and has been cropped for display here. Notice the tendency of points to cluster at corners. Note that some overhead is present due to the final double buffer rendering to display the image, and in a real situation without a window it would be even faster.

Figure 6.3: Another image with feature points marked as red dots. This one is uncropped and was rendered at 1940 x 1940 pixels in 148.14ms. These calculations could be further improved by taking advantage of the multicore CPU's as well as the GPU.

6.3 Recommendations for Further Work

This section examines what can be done in the future to complete the work on the vision module an improve upon, or replace , part of the current module.

6.4 Short term

This section describes some short term goals that should be prioritized in the future. They are of great importance to the robustness and ease of use of the module and are probably required in order for it to be used in a real SAR operation.

6.4.1 Finishing the remaining submodules

The last submodules where not prioritized for the reasons mentioned at the start of this chapter. However, they should be implemented and the responses of the system tested once the whole module is complete. Since the remaining part of the submodules simply perform calculations of descriptors and processing of the results. This stage possibly be implemented using OpenCV, witch would be ideal as the library is heavily optimized for multicore processing and has a large, active community surrounding it.

6.4.2 Scale invariance

The algorithm used for object recognition in this thesis is not scale invariant. This makes it a lot more difficult to detect humans at the variable heights a UAV will find itself in due to landscape variations, weather conditions and terrain. This next section proposes some potential addons or modifications to the module that can be developed in the future.

Object tracking

Object tracking is a field closely related to, and sometimes coupled with, object detection. In this case, object tracking could be used as a means of making the UAV adapt to its targets during flight. It could be initialized by one of the SAR users on the ground by taking a closup picture of one of the crew members, marking this region, calculating its feature descriptors and the track

this person as it flies up into the air. Since object tracking is based on looking for features in local areas around the previous features and updating the old ones when close matches (presumed to be the old features that have since been changed by image rotations, translations and scale). Thus, the UAV would gradually update its feature set describing its target on the way up, and during flight it can look for feature set matching one, or many, of the gradually varying sets computed on its way up.

A good example of a object tracking algorithm is the Matrioska algorithm [(Ed.)] developed for high speed, real time tracking of objects with partial occlusion. This algorithm could be investigated and potentially integrated with the system.

One weakness of this approach is that it will not necessarily be able to detect humans in different poses than the one initially tracked. If a human is lying on the ground, and the member of the SAR crew was standing upright through the whole process, the UAV might not be able to detect the person due to the large change in shape.

In addition, this would require more initialization to be done by the SAR crew, and as we are trying to assist them by making the search easier and more efficient, letting them do more work might not be ideal.

6.4.3 Proper testing

After making the module scale invariant, it should be tested at long ranges, and the resolution and camera as well as camera lens requirements should be thoroughly examined. Testing the module at multple images of humans at long ranges should be performed and sets of features representing humans should be stored and prepared for use in a real SAR operation.

6.5 Medium term

This section describe some medium term goals that could make the module more potent. They are points where optimization of the current module is possible, but not necessarily of paramount importance.

6.5.1 Concurrency on the CPU

The current implementation of the vision module does not take full advantage of the multicore CPU of the Raspberry PI 2. Although some of the submodules run in their own threads of execution, this is mostly for the sake of convenience and not performance. Cleverly dividing the module into several threads where possible and properly synchronizing them could allow for more parallelism in the module. FOr example, the image sampler could read in a new image while the previous image sample is being processed in the graphics pipeline, and the high level algorithms could process the previous images output by the shader programs at the same time, this has the potential of drastically reducing the iteration time of the module with little work required.

6.6 Long term

This section describe the long term goals that could be implemented in the future. They provide great potential for improving the module and allowing for more flexible implementations of certain algorithms. It could also potentially help solve the issue with calculating integral images on a GPU.

6.6.1 OpenCL

Mali GPU's are becoming very popular on the embedded market [*MALI GPU OpenCL* (n.d.)], more and more embedded development boards currently ship with these GPU, and more can be expected to appear in the future [*Pine64 homepage* (n.d.)]; [*Mali ARM homepage* (n.d.)]. These GPU's support the OpenCL computing API witch allows programs to take advantage of multiple processing units on the same system in order to divide work among the available CPU cores and GPU's without having to deal with the restrictions of OpenGL ES 2.0 (mainly texture buffer word sizes and redundant data structures like the windowing system and the vertex shader). This will not only improve performance but also allow the user more freedom to finely split their programs into concurrent and sequential parts to be executed on GPU's and CPU's respectively. Converting the vision module to use the OpenCL API will require rewriting a lot of code, but reusing the design might still be feasible.

Appendix A

Acronyms

1D One dimension

2D Two dimension

3D Three dimension

API Application Programming Interface

CPU Central processing unit

CSI Camera serial interface

DAIM Digital Arkivering og Innlevering av Masteroppgaver

FLOP Floating point operation

GLSL OpenGL Shading Language

GPU Graphics processing unit

IPC Inter process communication

LoG Laplacian of Gaussian

MCU Micro controller unit

MIT Massachusetts Institute of Technology

NTNU Norwegian University of Science and Technology

OGL OpenGL (Open Graphics Library)

OGLES OpenGL ES (Open Graphics Library Embedded Systems)

OS Operating system

SoC System on a chip

SAR Search And Rescue

UAV Unmanned Aerial Vehicle

V4L2 Video 4 Linux 2

Appendix B

Code

All the code is available on github:

- https://github.com/tommp/Ojo

In order to run the software, it must be compiled with the supplied makefile. V4L2 drivers must be in stalled and the paths in the CONFIG.h file must be properly set. The V4L2 bcm drivers must then be loaded into the kernel by issuing: sudo modprobe bcm2835-v4l2

Bibliography

3DR SHIFT (n.d.), https://3dr.com/shift-computer-vision-drones/l. Accessed: 2016-04-16.

Bilgic B., M. I. (n.d.), 'Efficient integral image computation on the gpu', dspace.mit.edu/openaccess-disseminate/1721.1/71883. Accessed: 2016-03-24.

DAIM (n.d.), https://daim.idi.ntnu.no/. Accessed: 2016-03-09.

DirectX (n.d.), https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx. Accessed: 2016-04-13.

(Ed.), A. P. (2013), *Image Analysis and Processing – ICIAP 2013*, Springer.

Eriksen, E. (n.d.), 'Lecture 2 the rank of a matrix', http://www.dr-eriksen.no/teaching/GRA6035/2010/lecture2-hand.pdf. Accessed: 2016-04-14.

Gamnes, G. (2013), *Autonomous Unmanned Aerial Vehicle in Search and Rescue - A Prestudy*, NTNU.

Gamnes, G. (2014), *Utilizing Unmanned Aerial Systems in Search and Rescue Operations*, NTNU.

GLSL documentation (n.d.), https://www.opengl.org/documentation/glsl/. Accessed: 2016-02-12.

Hammerseth, V. (2013), *Autonomous Unmanned Aerial Vehicle in Search and Rescue*, NTNU.

Herbert Bay, Tinne Tuytelaars, L. V. G. (n.d.), 'Surf: Speeded up robust features', http://www.vision.ee.ethz.ch/~surf/eccv06.pdf. Accessed: 2016-04-16.

IEEE homepage (n.d.), https://www.ieee.org/index.html. Accessed: 2016-04-25.

John Hennessy, D. P. (2011), *Computer Architecture - A Quantitative Approach, 5th Edition*, Morgan Kaufmann Publishers.

John Nickolls, D. K. (2011), *Computer Organization and Design - Appendix A: Graphics and computing GPU's, 4th Edition*, Patterson and Hennessy.

Khan Academy (n.d.), https://www.khanacademy.org/. Accessed: 2016-03-26.

Khronos group EGL (n.d.), https://www.khronos.org/egl. Accessed: 2016-02-15.

Khronos group OpenCL (n.d.), https://www.khronos.org/opencl/. Accessed: 2016-04-13.

Khronos group OpenglES (n.d.), https://www.khronos.org/opengles/. Accessed: 2016-02-11.

Kristen Grauman, B. L. (n.d.), 'Exerpt chapter from synthesis lecture draft: Visual recognition', http://www.cs.utexas.edu/~grauman/courses/fall2009/papers/local_features_synthesis_draft.pdf. Accessed: 2016-04-16.

Linux homepage (n.d.), https://www.linux.com/. Accessed: 2016-04-24.

Lowe, D. G. (n.d.), 'Distinctive image features from scale-invariant keypoints', https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf. Accessed: 2016-04-16.

Luftfartstilsynet homepage (n.d.), http://www.luftfartstilsynet.no/selvbetjening/allmennfly/Droner/. Accessed: 2016-04-24.

Mali ARM homepage (n.d.), https://www.arm.com/products/multimedia/mali-gpu/index.php. Accessed: 2016-05-24.

MALI GPU OpenCL (n.d.), http://malideveloper.arm.com/resources/sdks/mali-opencl-sdk/. Accessed: 2016-04-25.

Milan Sonka, Vaclav Hlavec, R. B. (2007), *Image Processing, Analysis and Machine Vision, third edition*, Cengage Learning.

MiT (n.d.), http://ocw.mit.edu/courses/. Accessed: 2016-03-26.

Navio online shop (n.d.), http://www.emlid.com/shop/navio2/. Accessed: 2016-03-13.

Nvidia CUDA (n.d.), http://docs.nvidia.com/cuda/#axzz49sGbE6Am. Accessed: 2016-04-13.

Nvidia Jetson (n.d.), http://www.nvidia.com/object/jetson-tx1-module.html. Accessed: 2016-04-15.

OpenCV homepage (n.d.), http://opencv.org/. Accessed: 2016-04-13.

OpenGL homepage (n.d.), https://www.opengl.org/. Accessed: 2016-02-05.

Pedersen, T. M. (2015), *Autonomous Unmanned Aerial Vehicle in Search and Rescue - Prototyping*, NTNU.

Pine64 homepage (n.d.), https://www.pine64.com/. Accessed: 2016-04-24.

POSIX base definitions (n.d.), http://pubs.opengroup.org/onlinepubs/9699919799/. Accessed: 2016-04-25.

Raspberry Pi model 2 datasheet (n.d.), https://www.adafruit.com/pdfs/raspberrypi2modelb.pdf. Accessed: 2016-05-21.

Scopus (n.d.), http://www.scopus.com/. Accessed: 2016-05-29.

Smith, S. (n.d.), 'Advanced image processing', http://www.sci.utah.edu/~fletcher/CS7960/slides/Scott.pdf. Accessed: 2016-04-16.

Stallings, W. (2014), *Operating Systems - Internals and Design principles, 8th Edition*, Pearson.

Sunil P Khatri, K. G. (2010), *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*, Springer Science and Business Media.

Szeliski, R. (2010), *Computer Vision: Algorithms and Applications*, Springer.

Timothy B. Terriberry, Lindley M. French, J. H. (n.d.), 'Gpu accelerating speeded-up robust features', http://www.cc.gatech.edu/conferences/3DPVT08/Program/Papers/paper154.pdf. Accessed: 2016-03-22.

Udoo homepage (n.d.), http://www.udoo.org/homepagex86/?utm_expid=71226024-0.d_yqlLQLRE2ZCprdMdfRZg.1. Accessed: 2016-04-24.

Videocore Dispmanx (n.d.), http://elinux.org/Raspberry_Pi_VideoCore_APIs#vc_dispmanx_.2A. Accessed: 2016-04-15.

VIKO (n.d.), http://www.ntnu.no/viko/english. Accessed: 2016-03-12.

Xenomai homepage (n.d.), https://xenomai.org/. Accessed: 2016-04-24.

Youtube (n.d.), https://www.youtube.com. Accessed: 2016-05-19.