Vision Utility Framework

A New Approach To Vision System Development

by

Amir Afrah

B.ASc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

February, 2009

Abstract

We are addressing two aspects of vision based system development that are not fully exploited in current frameworks: abstraction over low-level details and high-level module reusability. Through an evaluation of existing frameworks, we relate these shortcomings to the lack of systematic classification of sub-tasks in vision based system development. Our approach for addressing these two issues is to classify vision into decoupled sub-tasks, hence defining a clear scope for a vision based system development framework and its subcomponents. Firstly, we decompose the task of vision system development into data management and processing. We then proceed to further decompose data management into three components: data access, conversion and transportation.

To verify our approach for vision system development we present two frameworks: the Vision Utility (VU) framework for providing abstraction over the data management component; and the Hive framework for providing the data transportation and high-level code reuse. VU provides the data management functionality for developers while hiding the low-level system details through a simple yet flexible Application Programming Interface (API). VU mediates the communication between the developer's application, vision processing modules, and data sources by utilizing different frameworks for data access, conversion and transportation (Hive). We demonstrate VU's ability for providing abstraction over low-level system details through the examination of a vision system developed using the framework. Hive is a standalone event based framework for developing distributed vision based systems. Hive provides simple high-level methods for managing communication, control and configuration of reusable components. We verify the requirements of Hive (reusability and abstraction over inter-module data transportation) by presenting a number of different systems developed on the framework using a set of reusable modules.

Through this work we aim to demonstrate that this novel approach for vision system development could fundamentally change vision based system development by addressing the necessary abstraction, and promoting high-level code reuse.

Table of Contents

Appendices

List of Tables

List of Figures

List of Code Snippets

Acknowledgements

I have had the privilege of receiving guidance and support from some great colleagues and friends during the course of working on this thesis.

To my supervisor, Sid: thank you for your valuable guidance, motivation, and for providing me with great opportunities. I have learned a lot from you.

To Gregor: thank you for your help and friendship. This work would not be what it is without your input.

To Tony: thank you for your guidance, and for always managing to make me feel good with your positivity.

I thank all my friends in the HCT lab for making this journey fun and fulfilling.

Dedicated to my parents. Thank you for your love and support in every aspect of my life.

Chapter 1

Introduction

The field of computer vision has been going through an extraordinary metamorphosis within the last several years which has led to an increasing demand for the development of vision based systems. Vision processing, the automated analysis and manipulation of image and video data through computation, has been around since the 1960's. However, the activity in this field has accelerated greatly in the past 15 years. This increase is due to a number of factors, mainly the reduction in price and an increase in the accessibility of the tools (sensors and computation platforms), as well as our understanding of the field[15]. The tremendous boost in this field has sparked an increasing demand for end-to-end vision systems for deployment and prototyping of new algorithms from a heterogeneous set of sources and methods.

The task of developing end-to-end vision systems (or the *vision problem* as we refer to it in this work) has become increasingly sophisticated and complex as a by-product of recent growth in the field. In addition to general system issues, vision has specific requirements that introduce particular system development challenges. Foremost in the list of issues is the performance requirement of computer vision. Large volume of data and the computational complexity of current processing algorithms often exceeds the performance of readily available computation machines (PCs). Accessing and managing image data is another non-trivial issue that complicates the vision problem. There are numerous data format and protocols that are employed by different camera devices. Furthermore, new methods and algorithms in multiple camera research have emerged that employ inputs from several (possibly heterogeneous) cameras and other sensors to perform tasks such as calibration, tracking and 3D reconstruction[22, 25, 29, 40]. These multi-sensor systems amplify the issues associated with vision system development and require tremendous effort in order to access, manipulate, transport and process the data in real-time.

The need for addressing system-level development issues in the current state of computer vision follows the pattern described in [41]. Shaw and Garlan have identified the following pattern for general software engineering: as the complexity of a given problem increases, system level issues become more than just choosing an algorithm and data structures. Developers face issues such as composition of components, global control structures, communication protocols, synchronization and data access, physical distribution, etc. The vision problem has reached the point where system level issues are becoming quite significant. A framework that targets these issues in a systematic manner is necessary.

Currently system-level issues are addressed in the vision community using two approaches: developing in-house solutions and using available frameworks and packages. Development of in-house solutions results in a large amount of effort and resources as it requires users to implement the needed functionality. Furthermore, due to the high cost of development, these solutions tend to deal with system-level issues minimally which introduces a number of repercussions. The major two drawbacks being lack of robustness and generality as custom solutions are very application specific with no easy means for reuse. Due to these issues, the negative trade-offs that exist between flexibility and development effort is significant enough to deter many developers from this approach. The use of standardized frameworks and packages for addressing the system-level issues is generally a superior route. Standardized frameworks decrease development efforts by exploiting the redundancy in commonly required functionality thus providing module reuse.

There exists a number of frameworks and packages that target the vision problem such as OpenCV, Gandalf and VXL. Although these frameworks address critical computer vision tasks, in their current rendition they have a number of major shortcomings (as discussed in Chapter 2). The approach taken by existing frameworks prevents them from maximizing the potential *abstraction over low-level details* and *reusability* that could be extracted from computer vision.

In this thesis we are targeting the shortcomings of existing frameworks for addressing the vision problem. The approach we present is to provide a conceptual classification of the vision problem into a number of decoupled sub-tasks. We demonstrate that using this approach we can provide a framework with a well defined scope that promotes module reusability and abstraction over low-level infrastructure-level details. In the remainder of this chapter we present: the specific issues that exist with respect to current vision based system development approaches, our approach for addressing these issues and an overview of the contributions we have made through this work.

1.1 Problem

The main problem that we are addressing is that current approaches for vision system development do not fully address the need for *abstraction* over the low-level details and *high-level module reuse*.

Computer vision is a rich field that consists internally of a number of secondary components in addition to its primary task of processing image data via computation. These secondary issues mainly address the tasks of retrieval, pre-processing and delivery of image data from sources (cameras or image/video files) to the module responsible for performing the processing. Each one of these tasks is non-trivial and introduce a number of issues for developers. Current approaches require vision developers to explicitly deal with these issues (such as data types, communication and access to sources) which leads to a high awareness of the low-level details of these tasks (as discussed in Chapter 2). This awareness complicates the development task thereby increasing the load on developers.

Existing frameworks provide a function based approach for addressing (what they consider to be) an appropriate sub-set of the vision problem (as discussed in Section 2.3). Using this approach reusability is achieved within each framework through function reuse. Since no standardized classification of the vision problem currently exists, the functions provided by these frameworks often overlap in scope. As demonstrated by Makarenko et al. in [26] this leads to very poor reusability in the global sense.

1.2 Approach

The approach we have taken in this work is to provide a new methodology for vision based system development in order to address the shortcomings of current frameworks.

Firstly, we present a novel classification of the computer vision sub-tasks that is based on separating data processing from data management. We further classify the sub-components of data management into three strongly decoupled sub-tasks: data access, data conversion, and data transportation.

Secondly, we present a framework that is based directly on the classification of sub-tasks and demonstrate that there are significant advantages to vision system development using this approach.

Thirdly, we present a framework that implements the data transport sub-component of data management and demonstrate that there are significant advantages to having a transport mechanism for vision based development.

Much of the inspiration for this approach has been derived from the success of a similar approach in the field of Computer Graphics. The introduction of separation between graphics programming and display/interaction in Computer Graphics was done with Open Graphics Library and utility frameworks such as Graphics Library Utility Toolkit (GLUT), which revolutionized the Computer Graphics industry and made it accessible to a wider set of developers.

1.3 Contributions

The work presented in this thesis provides three main contributions with respect to the problems facing computer vision based application development (as described in Section 1.1). The following is an overview of these contributions:

1. We present a novel systematic decomposition of the vision problem into a number of decoupled sub-components.
2. We present the Vision Utility (VU) framework which is based on this decomposition and provides abstraction over the infrastructure-level tasks of computer vision.
3. We present the Hive framework that provides data transportation between vision modules, hence promoting module reusability.

Novel contributions of the Hive framework have been published in the International Conference on Distributed Smart Cameras (ICDSC'08) and the International Conference on Computer Vision Theory and Applications (VISAPP'09)[1, 30].

1.4 Overview

The remaining chapters of this thesis are laid out as follows: Chapter 2 will evaluate the previous research and existing frameworks that target the vision problem and other fields as they relate to this research. Chapter 3 presents our decomposition of the vision problem which provides the foundation for our approach to vision based system development. Chapter 4 presents the API and architecture of the VU framework and the proof of concept system developed using this framework. Chapter 5 presents the transport framework and a detailed description of its architecture and implementation as well as presenting a set of modules and applications based on this framework. Chapter 6 is the conclusion that summarizes the problem, the approach and the contributions of this work as well as presenting a future direction for the continuation of this research.

Chapter 2

Related Work

The vision problem contains many sub-components that extend into a number of different fields. This chapter provides a discussion of the current state of research and existing frameworks that aim to address the variety of limitations faced in vision system development.

This chapter is organized as follows: firstly, we review existing frameworks that focus on specific infrastructure-level issues of the vision problem. More specifically, we evaluate frameworks that target accessing image data from devices, converting the format of image data and data transport. Secondly, we present an overview of frameworks for multimedia application development and evaluate them with respect to requirements of vision based development. Finally, we evaluate existing frameworks that specifically attempt to target the vision problem in a comprehensive way.

2.1 Frameworks for Vision Data Management

In this section we evaluate existing frameworks that target infrastructure-level requirements of the vision problem. We focus on existing solutions for addressing the following tasks: accessing image data, conversion of image data and inter-module transportation of data in systems.

2.1.1 Access to Image Data

The following is an evaluation of several significant frameworks for standardization of image acquisition from various different sources:

Device Standardization There have been a number of efforts to create standardized formats for device manufacturers such as IIDC 1394-based Digital Camera Specification[3] and VAPIX network camera communication specification[11]. IIDC standardizes access to camera devices that use FireWire as the camera-to-PC interconnect. The IIDC standard specifies the camera registers, fields within those registers, video formats, modes of operation, and controls. IIDC ensures uniformity of access to devices that adhere to its standards. VAPIX is a HTTP protocol developed by Axis Corporation for communication with network cameras via TCP/IP and the server client model. Using VAPIX the image data and camera configuration data in VAPIX are sent as HTTP commands to and from the camera device allowing uniform communication to any network device that implements VAPIX. Although these standard formats present a theoretically valid approach, a convergence of such standards by manufacturers is not likely within the short term if ever.

Video4Linux Video4Linux (V4L) is an example of a class of solutions that attempt to provide seamless access to source via a uniform interface[38]. V4L provides standardized access to video devices by including a kernel interface for video capture. This approach utilizes Linux's paradigm of treating all input and output communication as reads and writes to a file and presents imaging devices as file handlers to users. V4L defines standard types for devices and video properties. It functions for opening and closing devices, changing device properties, data formats, and input and output methods that are implemented via system calls. Using these defined types and methods, programmers have access to the sources that are installed on a particular machine. Although V4L provides abstraction over specific camera protocols (e.g. IIDC) to the user quite effectively, it has two drawbacks. It is highly platform dependent and there is a high barrier to adding support for new devices. In order to add support for a new device (or class of devices) a developer needs to write kernel drivers which is a cumbersome task and eliminates any hope of an opportunity for platform independency.

2.1.2 Image Conversion Frameworks

The conversion of image data involves tasks such as image format conversion, resizing and affine transformation, etc. Performing these functions is not trivial and there have been a number of efforts to provide frameworks that provide support for these tasks. We discuss some of these frameworks in this section.

Open Source Image Format Packages Currently there exist a variety of open source cross-platform software packages for image conversion and manipulation such as ImageMagick[19], DevIL[13], and Netpbm[32]. These frameworks provide standard representations of images and a large set of manipulation routines that extend well beyond image format conversions and affine transformations. The main issue with the approach taken by these frameworks is the over exposure of low-level details of routines.

As we demonstrate in Chapters 3 and 4, the functionality of these frameworks could be separated into data management and data processing. This separation would allow the framework to provide high-level abstractions over the conversion routines and remove users' awareness of low-level details via a declarative model.

CoreImage$^{\text{TM}}$ and CoreVideo$^{\text{TM}}$ CoreImage and CoreVideo provide a plug-in based architecture for image manipulation and processing that utilizes graphics cards for hardware acceleration[20]. Although these frameworks provide a limited set of image manipulation and hardware accelerated processing, the model being used is quite successful at providing abstraction over the low-level details. The API for these frameworks presents functionality as processing blocks that can be aggregated to form a pipeline that performs one, or a number of, tasks. Image formats and properties are completely abstracted away from users in the intermediate sections of the pipeline. Users simply connect processing blocks together without externally managing images or pixels. This level of abstraction removes the necessity for users to deal with a large overhead of image manipulation.

2.1.3 Data Transport Middlewares

Modularity, code reuse and standard accessibility to system components are issues that have been quite apparent and pronounced in various disciplines; particularly in Robotics and Haptics research and development. Since communication between different sensors, actuators and control algorithms are central to these two fields there have been several projects that have attempted to provide middle-wares to support abstraction and module based development. This section provides an evaluation and discussion of the most widely used frameworks which is helpful in understanding the various decisions that we have made with respect to the development of our communication framework. We discuss the advantages and disadvantages of each framework and their relevance to vision.

Robotics

The most widely used frameworks for robotics developemnt are YARP[28], Player[10], Orca[6], Orocos[7] and CARMEN[31]. We discuss the first three as they provide the most insight into the different communication paradigms:

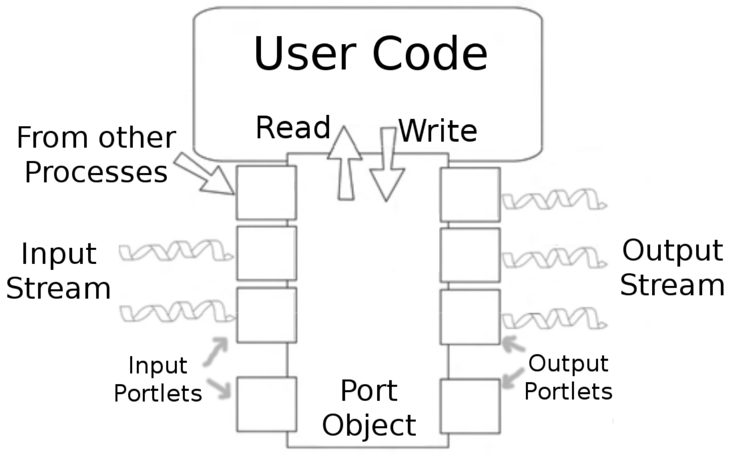


Figure 2.1: YARP's communication model with modules. Figure adapted from[28].

Yet Another Robotics Platform YARP provides a flexible communication medium based on the observer pattern between different running processes. Figure 2.1 shows the communication of the YARP module with users' code and other modules. The process that produces data opens an "out port" on a specific data type and the receiving process opens an "in port" to receive a data type. Data transfer will take place between the two ports upon connection. YARP provides an abstraction over the data transport medium and utilizes the network (TCP/UDP) in its current implementation. Although the communication mechanism of YARP is very similar to communication required by vision systems, some of its features are very specific to robotics. Much of this difference is due to the fundamental difference in use of a large number of sensors and actuators in robotics that can often lead to physical exceptions causing instability and unexpected crashes. There is a large emphasis on creating mutually exclusive blocks of processing that minimally interfere with each other. Another fundamental difference is the fact that YARP does not provide any means for controlling modules and is designed to be used strictly as a slave framework. Vision requires the communication framework to be the main backbone of the system, whereas YARP is a more lightweight approach that does not assume control in order to be compatible with other possible libraries and frameworks in the system.

Player Player is another software framework that provides a communication means for managing the flow of distributed sensing and control data between sensors and actuators in robots. Player utilizes a message protocol implemented using multi-threaded TCP sockets. Player is in essence a set of protocols implemented on a client-server model which provides transparent network access to the devices employed within a robot. The socket based communication protocol of Player provides: easy of distribution of sensors, actuators and control processes; independence between modules (as each module can be implemented in any language and on any platforms); and the convenience of a client-server model for information exchange. Figure 2.2 shows the overall architecture of Player. The middle section of the diagram is the Player framework that creates the connections between the devices and control routines. The protocol specified by Player is a low-level schema for information exchange between actuators (servers) and control routines (clients). This communication model is not sufficient for vision based systems that require more flexible peer to peer (client to client) networking. Player focuses on the data management of many actuator-control pairs. However, it lacks the complex control required for the creation of complex networks of sensors for a single unified application like vision.

Orca Orca is an open source, general robotics framework that was developed based on Component Based Software Engineering (CBSE)[9]. The aim of Orca project is to provide an extensible de-centralized middleware for connection reusable components that is independent of a particular architecture and the communication mechanism. The project also aims to provide an online repository of useful standalone components that can be used by all robot developers.

Orca identifies and categorizes the following components of robotics: 'objects', which refer to units of data that are communicated in between modules; 'communication pattern', which specifies the data transfer model such as client-push; 'transport mechanism', which specifies the data transfer protocol such as TCP/IP; and 'components', which refers to the algorithm implementations and hardware interfaces.

In order to build a system using Orca, a developer connects one or several components together using the transport mechanism. Orca is most suitable as a vision communication mechanism. However, there are fundamental limitations that prevent it from being as effective for vision systems. The concept of a controlling module is missing from Orca as no one component is designated to setup and configure other components. Also, the level of

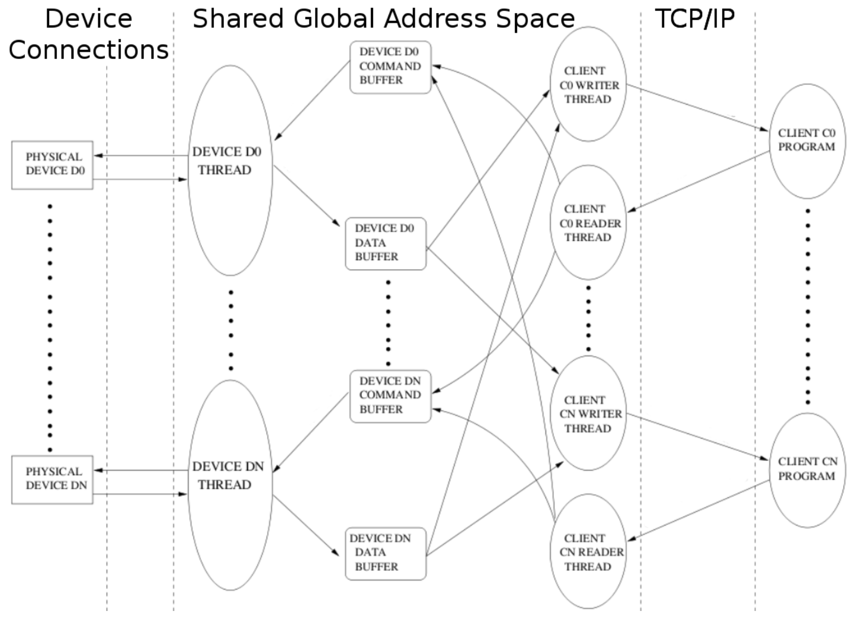


Figure 2.2: The scope and architecture of Player. Figure adapted from [10].

abstraction is not sufficient for vision developers as Orca users are forced to select communication mechanisms and indicate the communication patterns.

Robotics Frameworks Evaluation Makarenko et al. presented a study looking at existing robotics frameworks comparing the levels of abstraction of six existing solutions (Player, Orocos, Carmen, Orca, Yarp and OpenSlam)[26]. Makarenko et al. conclude that although each solution attempts to provide a framework for reusable components and is partially successful within its community, there is only an estimated 4% reusability across these platforms. Through the analysis of available solutions, Makarenko et al. illustrate that the level of abstraction provided within each of the solutions is insufficient. The authors organize the frameworks into three conceptual layers: Driver and Algorithm Implementations (DA); Communication Middleware (CM); and Robotic Software Framework (RSF). They indicate that while DA is the layer desirable for reusability, current solutions tend to couple it to the CM and in some cases, even worse, to RSF. The authors suggest that in the future, frameworks should be designed to have a thin middle-ware and RSF. Further, the levels of abstractions in future frame-

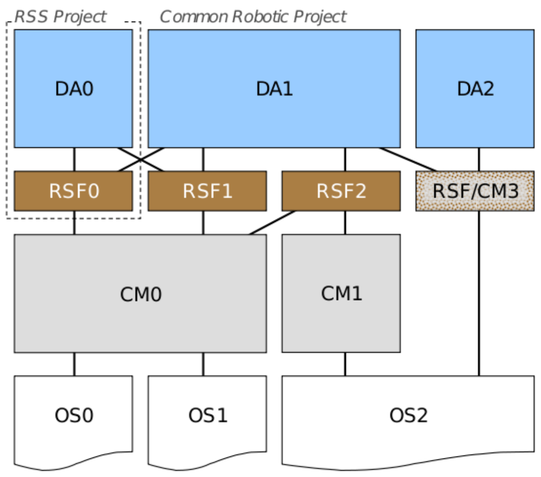


Figure 2.3: Increasing reusability through well defined component scope. Diagonal lines show that each component could be utilized in other frameworks. Figure adapted from [26].

works should decouple inter-dependency between layers in order to allow for a mix and match of layers amongst different frameworks. This decoupling would lead to maximizing code reuse in the community as a whole. Figure 2.3 shows the concept of mixing and matching graphically indicating how each decoupled component can be used with a number of other choices. We have adopted this concept in our design and implementation throughout this work.

Haptics

Another field that has dealt with similar issues as vision is haptics research in HCI. Haptics research inherently requires the use of many different sensors and performance is of critical priority, just as in any other Human Computer Interfaces. Pave et al. have presented the Real-Time Platform Middleware for Prototyping of Haptic Applications (RTPM)[34] in order to address some of these issues. RTPM is a framework for development of distributed real-time collaborative Haptic applications. RTPM provides abstraction over communication between its different distributed haptic, graphics, and control modules based on Remote Function Calls (RFC) model implemented by the Common Object Request Brokage Architecture (CORBA). RTPM medi-

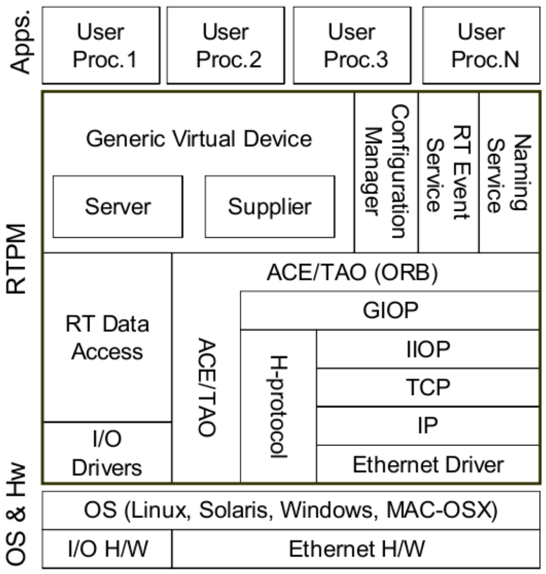


Figure 2.4: Architecture diagram of RTMP. Figure adapted from [34].

ates communication between user processes and the operating system while providing abstractions over communication details (as shown in Figure 2.4).

The main problem with the approach taken by RTPM in regards to communication for vision based applications is its strong client-server communication paradigm. Vision systems require more flexible communication patterns between components that support the creation of communication pipelines where the output of a module is directly connected to the input of another module without the mediation of the application. Using a strict client-server paradigm for vision would produce computational bottlenecks. In contrast to RTPM, Hive provides the ability to create complex peer-to-peer connections between modules as well as providing RFC for configuring the modules (as shown in Chapter 5).

Vision

The majority of activity in the area of data transportation with respect to vision systems consists of frameworks for distributed smart cameras, distributed sensor networks and generalized data communication solutions that developed out of specific applications. The following is an evaluation of the existing research in this field:

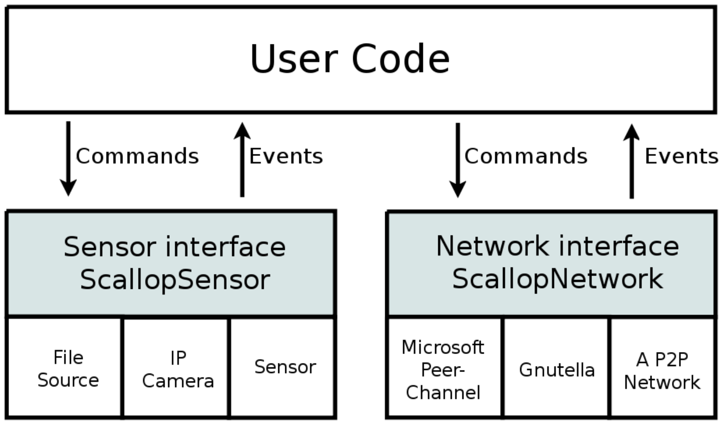


Figure 2.5: Architecture diagram of Scallop. Figure adapted from [37].

Scallop Scallop is an open framework for creating distributed sensor systems using the peer to peer connectivity paradigm[37]. Scallop provides a flexible communication middle-ware for sensor modules data transportation. Figure 2.5 shows the architecture of Scallop. The API mediates communication between user's code and sensors and other modules.

Scallop provides a plug-in mechanism for modules to be used within its framework which is required by vision systems. Scallop also avoids computational bottle-necks by focusing on a pure decentralized peer to peer communication model where each node communicates data directly to other nodes. The main disadvantage of Scallop is its lack of central control over modules. Due to this limitation there is no way for dynamic reconfiguration of the processing network which may be required for vision systems that need dynamic reconfiguration based on run-time data.

Cluster Based Smart Camera Framework Lei et al. present a generalized framework for communication in smart camera arrays that provides control and mediates data transportation between smart cameras[23]. This framework is based on a set of modules (nodes) running on a PC cluster. The nodes represent smart cameras as they contain a data source and a processing unit. The communication between these nodes is done through two channels: a message communicator and a data communicator. One of the nodes is designated as the master and it controls the operation of the other nodes (workers) by sending control messages through the message channel. All of the nodes in the framework are connected directly to each other forming a peer to peer network. The framework allows the smart camera array to perform a number of built-in tasks and allows for the extension of these tasks via modification of the nodes. The framework provides abstraction over device access, data communication and centralized control over the configuration of the nodes.

This framework provides a scalable peer to peer communication platform for inter-nodal data transportation as well as a central control mechanism for run-time configuration of the nodes. The main limitation with this framework is the homogeneity of its nodes. All of the nodes are designed to be utilized collectively to perform a single task, yet each node on its own is not a reusable entity. This methodology prevents the framework from supporting extension through a plug-in architecture of its nodes. All the nodes must be updated for the system to perform additional tasks beyond the default support.

RPV The RPV framework provides an API for developing vision systems by connecting a number of modules together on a cluster of PC's[2]. RPV provides the abstractions for gathering input data from sources and pipelining the operators to process the data. Using RPV, a programmer can develop distributed multi-camera systems with little overhead.

RPV provides abstraction over peer to peer data communication between its nodes and supports a plug-in model for the addition of source and processing nodes. However, RPV fails to address the need for a centralized control over the nodes and does not support run-time reconfiguration of the network of nodes.

2.2 Frameworks for Multimedia Development

There currently exists a number of popular frameworks for multimedia application development that target similar issues as vision development. Apple's QuickTime^{TM}, Sun's Java Media Framework, and Microsoft's DirectShow are three of the main players in this field. These frameworks focus mainly on capturing, decoding and rendering to the screen (playing) video (and audio). However, they provide a limited set of filters for manipulating this data and no support for data communication. The following is a discussion of these three frameworks in more detail:

QuickTime 7$^{\mathrm{TM}}$ QuickTime is a media framework developed by Apple Inc. for managing and handling various multimedia requirements[36]. In addition to its ability to manage audio, animation, and graphics, QuickTime provides functionality for capturing, processing, encoding, decoding, and the delivery of video data through a framework called QTKit. QTKit's view of vision data is based on the concept of video clips or as QuickTime calls it 'movies'. QTKit provides a set of classes for accessing vision data from sources (capture devices and files) that provide high-level abstractions over the source's low-level details. QuickTime also provides a very comprehensive, high-level mechanism for decoding and encoding video between a large number of different formats. There are two limitations with the QuickTime's approach with respect to vision based system development. The first issue is QuickTime's lack of support for data processing. Although QuickTime provides the use of CoreImage and CoreVideo frameworks which provide a small subset of built-in image manipulation routines, it does not provide any mechanism for advanced vision processing to be integrated in the framework. This limitation forces users to explicitly deal with the overhead involved in transferring image data in between the framework and the external processing module which significantly reduces the effectiveness of QuickTime as a vision based system development framework. The second major limitation of QuickTime is that it does not provide a means mechanism for the integration of a data transportation mechanism to address the communication between distributed tasks. We have addressed these limitations in our approach presented in Chapter 4.

Java Media Framework Java Media Framework (JMF)[21] is a cross-platform multimedia framework similar to QuickTime that provides capture, playback, streaming and transcoding of multimedia in a number of different formats for Java developers. The architecture of JMF consists of three stages: input, processing and output. The input stage provides routines for accessing video data from capture devices, files and network inputs. The processing stage deals with converting data using different codecs and adding common video effects. The output stage deals with rendering the video data, saving it to disk and sending the data via network. The fundamental limitations of JMF are similar to the QuickTime framework. The processing aspect is simplified to use intermediate filters and codecs (although JMF provides limited support for codecs compared with QuickTime) with no built-in support for extended processing. Also, the support for data transportation is limited to reads from and writes to the network.

DirectShow DirectShow[14] is a multimedia framework developed by Microsoft to provide a common interface for managing multimedia across many programming languages. DirectShow is an extensible filter-based framework that provides data capture, filtering, conversion and rendering of video and audio data. DirectShow interfaces with the Windows Driver Model in order to provide access to a large number of capture and filter devices. DirectShow insulates the application programmer from the details of accessing these devices; however, it also suffers from the same drawbacks as other multimedia frameworks as it provides no support for complex video processing and data transportation.

2.3 Frameworks for Vision System Development

There have been a number of frameworks that have been designed to target vision processing as whole. We discuss a few of these frameworks in this section and compare them with our approach.

OpenCV The Open Computer Vision library (OpenCV)[5] is a comprehensive and widely used vision processing framework. The overall design of OpenCV relies on declaring data type definitions for image and vision entities and providing functions for operating on and extracting data from them. OpenCV provides limited system development support (source access and image manipulation) for developers to easily create vision systems. OpenCV provides a framework for accessing data from cameras installed on the system that utilizes an OS specific framework such as V4L. OpenCV also provides a function based approach for image format conversion and resizing. These functions access images of different formats from disk and convert the data into OpenCV's native image class. OpenCV also provides routines that allow the programmer to resize images using a number of different interpolation methods, extract sub-regions of images to sub-pixel accuracy and extract specific channels from a multi-channel image.

OpenCV provides function level code reuse within the framework; however, it provides no easy way for users to provide higher level blocks that could be reused outside OpenCV. As demonstrated by Makarenko et al.[26] this approach to code reuse is not successful when examined on a scale that extends beyond the OpenCV framework. In addition to code reuse, OpenCV has other shortcomings and drawbacks that make it inadequate for larger scale vision system development. Limitations such as lack of support for distribution, multithreading, limited source access and image data manipulation, force developers to create custom frameworks (or utilize other existing frameworks) that employ OpenCV as a complementary framework.

OpenCV is an excellent representation of the current approach to computer vision development. OpenCV provides users with the tools necessary to create end to end vision systems, yet these tools cover only a subset of the vision problem. Figure 2.6 shows the components of the OpenCV framework. The computer vision task has been addressed by 3 libraries: CxCore, OpenCV and HighGui. CxCore defines a set of data types for representing common entities such as images, points and arrays and provides the functions that perform operations on these data types such as element access, copying, arithmetic, etc... OpenCV provides functions that implement computer vision algorithms as well as data access and manipulation. HighGui addresses the image and video I/O as well as window management and display. Although there is some degree of classification embedded in the categorization of the framework's tasks into these three classes, the functionality of these libraries (managing and processing data) overlap in a number of cases and are all presented to the user at the same level using the same function based API. It is clear from this superficial classification that OpenCV does not provide a strong conceptual separation between different classes of vision processing tasks. Consequently, as a result of the lack of conceptual categorization, OpenCV has major limitations and shortcomings preventing its extensive use as the major tool for system development.

Gandalf Gandalf is a function based computer vision and numerical analysis library that defines a set of data types relating to mathematical and computer vision operations. Gandalf consists of four packages: a Common package that defines simple structures and data types used by other packages; a Linear algebra package with a large number of routines for matrix and vector manipulations; an Image package that declares the image structure and provides low-level image manipulation routines; and a Vision package that provides a number of standard image processing, computer vision and geometrical routines.

Gandalf is similar to OpenCV in its approach to provide a function-based set of routines for processing numerical and image data. However, Gandalf focuses more on processing and less on providing functionality for dealing with reading, writing, manipulating image data formats, and displaying[17]. Gandalf has the same code reusability issue as OpenCV as it is also a function based framework. Gandalf can not be used as an all encompassing vision framework, because of its focus and limited scope; it does not provide

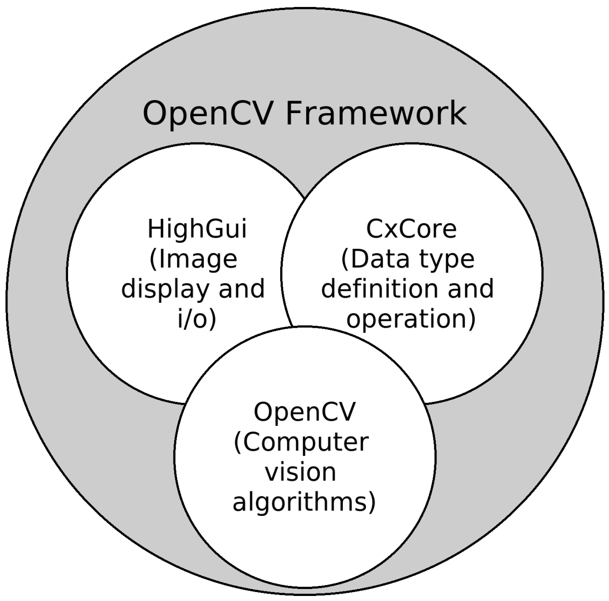

OpenCV Framework
HighGui (Image display and i/o)
CxCore (Data type definition and operation)
OpenCV (Computer vision algorithms)

Figure 2.6: The OpenCV computer vision framework internal components.

support for the data retrieval and preparation tasks. In its current rendition, Gandalf could be used as a processing component of the VU framework. However, it will need a much more standardized interface in order to co-exist with the other components in the framework. The VU framework presented in this thesis provides the standardized interface for interaction with frameworks such as Gandalf.

VXL VXL is a vision development framework that consists of a collection of C++ libraries for computer vision based development[43]. VXL contains four core libraries for numeric processing (VNL) that provides methods for matrices, vectors, decompositions, optimizers, etc, image access and manipulation (VIL), geometry definition (VGL), and other platform-independent vision related functionality (VSL, VBL, VUL). In addition to the core libraries, VXL contains a number of other libraries that cover a variety of vision problems. VXL provides a modular framework where each package could may be used as a lightweight unit independent of others.

The VXL package has an advantage over OpenCV and Gandalf in that it provides different libraries to address the different categories of the vision problem (reflected by its core libraries). The fundamental approach of VXL is however the same as the other two; to provide a function based set of libraries. The main disadvantage of this framework is that unlike the VU framework, it relies on users to explicitly manage and handle the input data preparation which includes access and manipulation. As we prove in Chapter 4, this level of detail is unnecessary for the application developer.

Matlab MATLAB is a numerically oriented programming environment that provides easy methods for matrix manipulation, plotting of data, implementation of algorithms and a number of other useful features[27]. MATLAB also contains a package that provides access to image file access and common image processing and analysis routines. MATLAB is commonly used for algorithm prototyping by researchers as it's easy interface and readily accessible image processing package allow for quick development. However, MATLAB is inadequate for any serious system development due to its poor computational performance and its inability to create end to end solutions. The scope of MATLAB is quite different than VU as VU targets the development of large deploy-able vision systems.

2.4 Conclusion

In this chapter we explored the two current approaches for utilizing existing frameworks to address the vision problem: using several standalone frameworks that address different components of the vision problem and using a single comprehensive framework (perhaps also in conjunction of other standalone frameworks) to address all of the sub tasks of the vision problem.

We reviewed a number of different frameworks that target specific sub tasks of vision system development in three categories: image data access, image data conversion, and transportation. Through a discussion of each framework, we showed that frameworks which provide a large set of functionality for data access and image data conversion exist, yet they overlap in scope and expose unnecessary low-level details to users. With respect to data transportation, we showed there are several frameworks; however, they do not address all the requirements of transportation of vision applications.

In the evaluation of the comprehensive frameworks we demonstrated that they have three major shortcomings: they do not provide all of the functionality needed by users; similar to the other frameworks they force the user to deal with low-level details of all the sub tasks of the vision problem; and they do not provide adequate support for high-level reusability.

The overall conclusion of the evaluation presented in this chapter is that the current approaches for vision based system development suffer from a variety of shortcomings which limit reusability and require a large effort on the part of developers.

Chapter 3

Classification of the Vision Problem

This chapter presents an overview of our methodology for addressing vision system development issues. It is based on a systematic classification of the vision problem into a number of decoupled sub-tasks.

As discussed in Chapter 2, current frameworks for vision based application development such as OpenCV, VXL and Gandalf focus on processing image and video data while offering intermediate support for retrieval, prepossessing and transportation of image and video data. Furthermore, they fail to provide any strong notion of classification of different sub-tasks within computer vision. All of the functionality of these frameworks are offered to users on the same level and through the same interface (function based). The lack of classification and abstraction in current approaches to vision based system development leads to limited component reuse and large development efforts due to the unnecessary exposure of low-level development details to developers.

In the following sections we directly target the lack of classification of the vision problem in current frameworks. We present a categorization of the various computer vision sub-tasks into a set of decoupled components which provides two main advantages over the traditional approaches: abstraction over low-level details, and increased reusability. Based on this categorization we define the scope of a framework for vision based development. We present our component based model and discuss the scope of each component.

3.1 Distinction between Data Management and Processing

The main objective of vision systems is to process image data via computational methods in order to perform one of two tasks:

- Extract high-level descriptions from images such as location of persons or objects etc. We refer to this information as meta-data in this context.

- Manipulate image data for example applying a smoothing filter or correct radial distortion etc.

The fundamental difference between these two types of processing is the type of output they produce. The first task produces high-level meta-data whereas the second task produces image data. We refer to the task of processing images which includes both analysis and manipulation of image data as *data processing*.

In vision systems, in order to perform the task of data processing, the system developer must address the following issues: retrieve the data from a data source that could be a camera, an image file, a video file or a number of other devices; deliver the data from the source to the module in charge of performing the actual processing; modify the format of the data to match the format expected by the processing module; and deliver the output from the processing module to the module in charge of storing, displaying or using the output for any other purpose. We refer to these tasks collectively as *data management* with respect to the vision problem. As described, data management is composed of a number of different non-trivial, yet necessary sub-tasks.

Although the data processing task has specific requirements for the format and communication protocol of its inputs and outputs, it is decoupled from the data management task as long as a standard interface is defined for communication between the two. This decoupling allows for the existence of a framework that implements the data management tasks while merely providing a standard interface for communication with modules that perform the processing. A framework with this limited scope allows developers to focus strictly on creating processing modules without addressing any data management. Using this model for vision system development means that the scope of data processing becomes well defined and quite thin, allowing for greater code reuse as demonstrated by Makarenko et al[26]. In Chapter 4 we present a framework that is based on the separation between data management and processing while highlighting the direct mapping between this classification and the scope of the framework. Furthermore, we proceed to validate this approach by demonstrating the advantages of the framework in Section 4.3.3.

The approach of separating management of data from processing has proved successful in other fields, a good example being computer graphics. Modern computer graphics programming is divided into two parts, a languages specification such as Open Graphics Library (OpenGL)[39] and utility frameworks such as Graphics Library Utility Toolkit (GLUT)[18]. OpenGL specifies graphics language which allows users to perform graphics tasks such as creating and manipulating polygons, shades and textures. Frameworks such as GLUT provide the data management functionality such as interaction from the user and displaying the graphics pipelines output to the screen. The abstraction provided by utility frameworks allows the language specification to be reusable, completely portable across platforms and accelerated using different graphics hardware.

The following section describes in more detail the scope of the functionality of the data management task and its sub-components.

3.2 Sub-Components of Data Management

As defined in the previous section, data management is the task of accessing, preprocessing and delivery of input and outputs to and from the processing task. Data management includes a number of non-trivial sub-tasks with respect to accessing image data from devices and managing data in the vision system. These tasks can be classified into three categories:

- Data Access
- Data Transportation
- Data Conversion

Figure 3.1 shows our classification of the computer vision. As the diagram indicates, there is a strong separation between data management and processing. Furthermore, it can be seen that the three sub-components within data management are also separated from each other indicating the strong decoupling that exists with respect to data management's internal components.

Although these three components are standalone and completely decoupled from one another, they can collectively be utilized to address the data management requirement of vision systems in an abstracted way. In Chapter 4 we present the VU framework directly based on this model that provides vision data management. We discuss in detail its conceptual design and implementation while demonstrating its benefits through a set of example applications developed on it.

The following is an overview of the three internal components of data management.

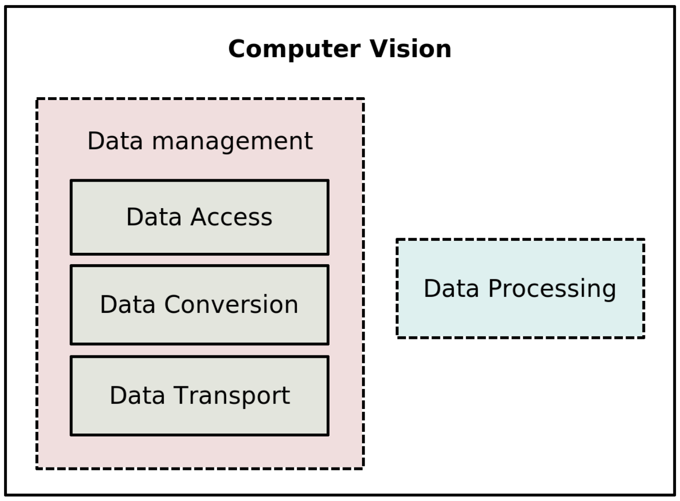


Figure 3.1: Graphical representation of our classification of the vision problem.

Data Access There exist a wide variety of devices and other media (e.g. files) that could be used as sources of image data for the processing module. The data access module addresses the task of obtaining data from these devices. More specifically, the data access module deals with configuration of the source and retrieval of image data in a standard format. A detailed explanation of the data access component is presented through our discussion of the Unified Camera Framework in Section 4.2.2.

Data Transportation In many vision systems, especially large scale systems, the components of the system such as source and processor modules are often distributed over a network or physically connected to several machines via a communication medium such as a bus. The data transport module addresses the need for inter-communication of data and control amongst the different modules of the vision system. We have developed a standalone framework for addressing the issues involved in data transportation for vision called Hive. Chapter 5 presents the conceptual design and implementation of this framework in detail.

Data Conversion Different sources employ a large variety of data formats and compression schemes to represent the image data. In order for modules to communicate data effectively, they need to agree on the communicated data types. The data manipulation module addresses the need for conversion of data formats in order to allow devices with different native representations of image data to communicate. A detailed explanation of the data conversion component is presented in Section 4.2.2.

3.3 Conclusion

In order to address the drawbacks of current frameworks for vision based application development, we presented a new approach that is based on a novel classification of the vision problem. We decomposed vision into two tasks: processing vision data and managing vision data. We further classified the data management task into: data access, which retrieves image data from sources; data transport, which delivers data in between the modules; and data conversion, which converts the between different data formats required by modules.

The classification of computer vision tasks in this way provides two major advantages: firstly, it reduces the developers' and researchers' work load by providing high level abstractions and allowing them to focus on development and extension of a particular task independently of others, and secondly, it promotes developing modularized, reusable code by removing inter-task dependencies via standardization of a clear interface between tasks.

Chapter 4

The Vision Utility Framework

In this chapter we present the VU framework for vision based application development. It is directly based on the classification of the vision problem presented in Chapter 3. VU provides the required data management functionality to vision developers via an API that abstracts the details of its sub-components.

The goal of this chapter is to verify that the approach of vision system development through separation of vision into sub-tasks is valid and that it provides the necessary abstraction over low-level data management details. More specifically, the framework will be evaluated in terms of the following criteria in Section 4.3.3:

- Is data capture from sources de-coupled from processing?
- Are the image data format details hidden from the user?
- Does the framework provide abstraction over inter-component communication?

We present the VU framework in three parts. We first present an overview of the framework which includes the development model, the conceptual design, and components of VU. Second, we discuss the details of the architecture of the VU framework and finally we present the proof-of-concept which includes an implementation of the framework, a vision system based on the framework, and an evaluation of the effectiveness of the framework.

The VU framework in its current build is designed to address a subset of the vision problem and it is intentionally limited to support vision systems that consist of a single source and a single processor. This decision was made in order to simplify the development task. The VU approach however generalizes to more sophisticated frameworks that support multiple sources and processors and connection patterns.

4.1 VU Overview

In this section we introduce the VU framework. The VU approach towards vision based system development is fundamentally different from traditional frameworks as it is based on a novel classification of the vision problem. We present this new approach by discussing the system development model, the modules of the framework and the communication model used.

4.1.1 System Development Model

VU allows programmers to create vision systems by developing *applications* that configure and connect *sources* and *processors*. In this model, sources are modules that produce image data, processors are modules that perform processing on data, and the application is the control center of the whole system. This development model preserves the separation between the data management tasks performed by the framework (as described in Section 3.2) and the data processing performed by the processing module.

VU provides a system development environment that encapsulates the data management functionality while providing an API that implicitly preserves the natural decoupling between the data management task and data processing task. Figure 4.1 shows the scope of the VU framework and its relationship to the application, source and processing blocks.

4.1.2 Modules

The following sections describe the scope of the source, processor, and the application modules with respect to the VU framework.

Source

The representation of image data sources within VU is generalized to a black box with an interface that includes the configuration parameters and the output. Figure 4.2a shows the representation of the source block. In the VU framework all sources are abstracted as configurable virtual cameras. The configuration interface of the source block exposes the internal configuration of the source (such as resolution, exposure settings, white balance and so forth in the case of actual camera devices.)

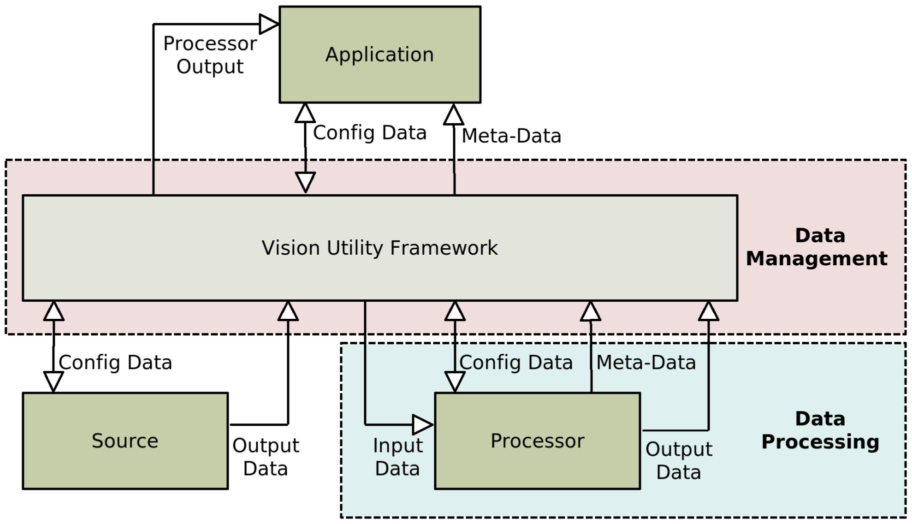


Figure 4.1: System development using the VU framework. The scope of the framework is clearly marked.

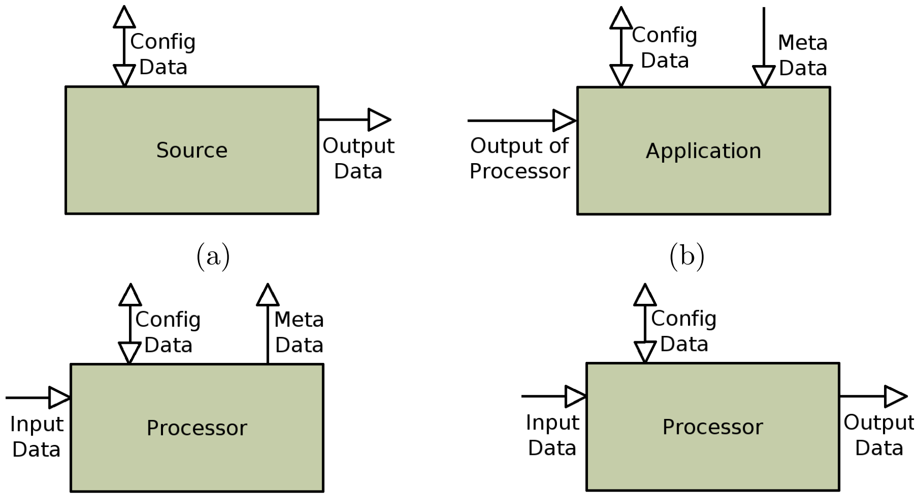


Figure 4.2: Representation of the a)source, b)application, c)analysing processor and d)image based processor modules in the VU framework.

Processor

As discussed in Section 3.1 we classify the processing task into two categories: extraction of meta-data and manipulation of data. Within VU the two types of processors are distinguished. The first type is called the *image analysis processor* and the second type is called the *image manipulation processor*. Figures 4.2c and (d) show the difference between the two processors. While they are both presented as black boxes with an input and configuration data, the image analysis processor produces meta-data whereas the image based processor produces image data. The input is image data and the configuration parameters allow customization of the algorithm to accommodate the nature of the input data. Meta-data is a high-level description extracted by the analysis process and the output data is an image. In practice it is possible for a processor device to be a mixture of the two models and provide both meta-data and output image data.

The following is a simple example which demonstrates the concepts behind the processor representation in the VU framework: A basic foreground extraction algorithm works by subtracting input images from a constant background image flagging the pixels that are greater than a threshold value as foreground and grouping these pixels by bounding boxes. In this case, the input is a pixel based image, the output is a binary image (foreground-1, background-0), the configuration is a threshold value and control over background model selection, and the meta-data is the list of bounding boxes. Since the outputs are both high-level (meta-data such as location and bounding boxes) and low-level (meta-data such as binary image), this processor is a mixture of the analysis and image based processor.

This simple model for the processing unit can scale to more complicated algorithms by presenting a broader definition of the input set and the parameter set. The overall paradigm of inputs, outputs, parameters and meta-data is still valid even if a processing block requiring a set of images as input and a parameter set selecting amongst multiple algorithms is presented.

Application

The application is developed by the users of the VU framework. The application uses the VU API to access, configure and connect sources to processors and retrieve meta-data. Figure 4.2c shows the representation of the application block within the framework. The application can receive the output of the source and processor as well as the configuration parameters and meta-data.

4.1.3 Communication Model

The communication model for the VU framework is based on asynchronous callbacks. A discussion of the decision to employ asynchronous callbacks is included in Chapter 5 where we present the architecture of the Hive framework.

VU provides two methods for communicating configuration and data amongst devices: remote function calls for configuration and retrieval of meta-data of devices, and receiving outputs from devices. Sources only support the first method whereas processors support both.

Remote function calls allow the application to read and write data to a device. The data can be configuration parameters or meta-data produced by the device. For source devices the application can call the remote functions at any time; however, for processor devices there is a synchronous callback which is invoked upon processing a frame of data. This callback blocks and allows the application to exchange data with the device. Through this callback the application is allowed to control, modify and verify resulting data of the processing module and possibly make decisions whether to move on to the next frame or reprocess (further process) the current frame.

In addition to the remote function calls, processors allow the transfer of their outputs to applications. The output here refers strictly to image data as other outputs are transferred via the remote function call mechanism described above. Limiting the output of the devices to image data is done in order to provide the functionality of image manipulation framework and allow for possible extension of the framework to multi-processing devices via cascading of the outputs.

4.2 Architecture

The VU framework provides the data management required for vision based application development through an interface that abstracts the details of the data management operations. The architecture of the VU framework therefore contains two layers: The Interface layer and the Core layer. As shown in Figure 4.3 VU's architecture consists of the interconnection of the following components:

- Interface layer
  - Application Interface
  - Driver Interface

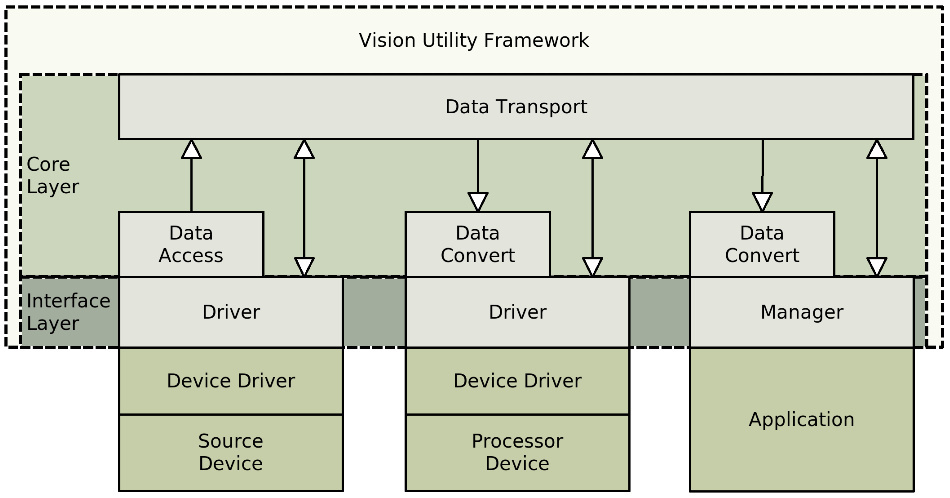


Figure 4.3: The VU architecture.

- Core layer
  - Data Transport
  - Data Access
  - Data Convert

The following sections provide a description of each layer and its subsequent components:

4.2.1 Interface Layer

The Interface layer contains two components; the *Application Interface* which provides the programmer access to the functionality provided by the Core layer, and the *Driver Interface* which provides a contact point for device developers allowing them to create drivers for existing sources and processors. Since the VU framework is based on asynchronous callbacks, in both instances the interface routines allow the user to register a number of callbacks to perform actions based on arriving events. The following is a discussion of the interface and the driver component of the framework.

Application Interface

The Application Interface layer provides routines to expose the functionality of the VU framework to the vision application programmer. It provides access and abstraction over the functionality of the Core layer to simplify the task of the developer. The following is a description of the services provided by the Application Interface:

- **Device query**: This functionality allows the programmer to query the devices (sources and processors) that are available and their general properties.
- **Device communication**: This functionality allows the programmer to send and receive data from devices in order to set or get configuration parameteras and meta-data.
- **Device interconnection**: This functionality allows programmers to create a vision processing context by connecting sources to processors.
- **Callback handling**: This functionality allows programmers to register handlers for the communication callbacks from each processor.

Driver Interface

The Driver Interface standardizes the communication between devices and VU while providing abstraction over the implementation details of devices. Devices can virtually be anything, ranging from software routines to hardware accelerated implementations via GPUs, as long as they provide a 'driver' that adheres to the defined interface.

Since communication between devices and the VU framework is strictly callback based, the Driver Interface provides the mechanism to allow developers to register their services in order for them to be VU compatible. These services must include handlers for configuration providing output and the main processing of the device.

4.2.2 Core Layer

The Core layer of the VU framework performs the data management services. As discussed in Chapter 3 data management consists of the following three tasks:

- **Data Access**: Configuration and retrieval of data from sources

- **Data Conversion**: Conversion and transformation of image data
- **Data Transport**: Data transfer mediation between modules

VU provides abstraction over the details of these tasks from the user through the use of the Core layer. The following is a discussion of each of the sub-components of the Core layer:

Data Access

Currently there are large variations of protocols and formats used for accessing image data, which is mostly due to the diversity of existing image sources. Analogue cameras, digital cameras, image and movie files, as well as imaging sensors based on mediums other than light (e.g. ultrasound) are just several examples of sources that are commonly used. Even within each group there exists a wide variety of devices with different representations in software and physical interfaces. This variability results in a lot of customized effort in order to configure and get data from these devices. In addition to the variability within devices, there exist numerous representations of image and movie data that introduces additional complexity.

This diversity has created a need for a general framework that provides uniform access to image sources regardless of access protocols, physical instantiation and native data type. The aim of this framework would be to get image data in native format from any image and video source in a uniform way.

In order to unify access to image data from devices and other mediums, we have created a model that presents image data sources as virtual cameras with a unified interface. We have named this conceptual approach the Unified Camera Framework (UCF). UCF defines and implements two sets of specifications: the *image specification* that specifies the format of the image data, and the *device communication protocol* that specifies a standard for device communication.

The image specification within UCF is the definition of the meta-data format that provides a complete description of its associated image data. This specification identifies the properties and the format of the image data allowing the consumer of the data to be able to decode and use the image data. Typical image properties include resolution and compression type. However, there are many other properties that need to be included in order to provide a comprehensive description of images.

The device communication protocol within UCF specifies the method for communication of data to and from devices. The communicated information consists of image and configuration data. This protocol would be implemented on top of the existing device driver in order to allow that device to be used as a UCF compatible device.

Data Conversion

Manually pre-processing images and converting them into the format required by the processing task is a non-trivial task. Pre-processing image data includes operations such as format conversion, resolution change, colour-space change and affine transformation. These tasks are non-trivial and require additional effort on the developer's part. The existence of a framework that performs these tasks seamlessly as a preprocessing step for a processing task is essential to the vision processing community. However, such a framework is still not available in a seamless manner. The main reason for the non-existence of such a framework is due mostly to the inadequate standard method for the representation of image data properties. Without a standard representation, it is very difficult to provide a framework for conversion between data of different formats.

We propose an image manipulation framework that allows for seamless transformation between data with different properties. This framework would rely on the image specification provided by the data access framework explained in the previous section. The reliance on the standard image specification makes this framework conceptually simple as it can be represented as a black box. The image specification would provide a uniform method for identification between the input and output of the framework and based on the output requirements the framework would apply the appropriate conversions and transformations. It is important to note that there is some overlap between operations that are considered to be part of this framework (data format conversion/transformation) and data processing. However, including functionality here does not preclude the processing task from performing similar operations to data processing. This is demonstrated in Section 4.3.3 with the resizing functionality of the processor.

Data Transport

Transportation of data between different source and processing modules is an important aspect of vision system development. The vision problem is inherently a distributed task due to its employment of multiple sensors. Communication between these sensors and processing modules is not a trivial task. As discussed in Chapter 2, there are no existing frameworks that target the data transport requirement of vision system development adequately. In order to address data transport we have developed an architecture for building distributed modular systems focusing on vision called Hive. We present the motivation and the complete description of the Hive framework in Chapter 5.

4.3 Proof of Concept

In this section we present an evaluation of the framework in three parts. In the first part we present the implementation details of the VU framework that we have developed in order to verify our approach towards vision based system development. In the second part we present a system that we developed using the VU framework. In the third and final part of this section, we present an evaluation of the framework, and discussion of the results of the application.

The current version of the VU framework is a proof-of-concept system that illustrates the validity of our approach; therefore, it is not a fully featured framework. The main limitations of this implementation are the following: VU only supports a single pipeline consisting of one source and one processor; VU uses the above frameworks that are also not fully featured and only provide a subset of the features. To implement a full version of this framework would require fully featured versions of the components stated above and is currently out of the scope of this thesis.

4.3.1 Implementation of VU Framework

The implementation of the VU framework closely follows the architecture presented above. We have implemented an API that exposes the functionality of VU. In order to provide the needed functionality to the users and maintain consistency with the classification presented in Chapter 3, we have developed three frameworks to address the data access, manipulation and transport requirements. The following is a discussion of the API of the framework as well as the implementation details of the components of the framework.

Interface Layer

In the VU implementation we created two separate interfaces: an application API and a driver API. The application API exposes the services of the framework to the vision system developer, whereas the driver API allows

Requirement	Methods
Device query	GetDevices()

Device configuration	SetParameter()
	GetParameter()

Device interconnection	CreateContext()
	SetContextSource()

Callback handling	SetIdleFn()
	RegisterImageOutputCallback()
	RegisterPostprocessCallback()

Table 4.1: This table shows the methods that provide the VU application functionality.

developers to implement drivers for devices to use in the VU framework. Both API's use the asynchronous callback model to interface with users. A complete list of the VU API is included in Appendix C which outlines the description, inputs and outputs of each VU routine.

Application API The application API supports device communication and control through a set of routines that implement the tasks outlined in Section 4.2.1. Table 4.1 shows the correlation between the requirements of the VU framework and the routines that provide the requirements. The GetDevices() routine provides a list of the available devices to the user. The user can query and set the parameters of these devices using the GetParameter() and SetParameter() routines. Device interconnection is not a direct call, but two calls. The first call (CreateContext()) starts a vision processing 'context' associated to a particular processing device. The second call (SetContextSource()) associates a source device to the 'context' hence producing the connection between the source and the processor. SetIdleFn() allows the user to provide an idle routine that gets called when the framework is not processing events. Using RegisterPostprocessCallback() the user can provide a routine that gets called when the processor has finished processing a frame of data. This allows the user to check the status of the system and possibly configure the processor or source during runtime. Using RegisterImageOutputCallback() the user can provide a routine that accepts a processor's output image in a specific format.

Requirement	Methods

Callback handling	RegisterProcessorFn()
	RegisterGetConfigurationFn()
	RegisterSetConfigurationFn()
	RegisterDataReciever()

Output production	SendOutput()
	GetParameter()

Table 4.2: This table shows the methods required to device developers in order to write drivers for various devices

Driver API The Driver API provides a standard interface to mediate the communication between the VU framework and devices through asynchronous callbacks. Table 4.2 lists the API methods for the Driver. For a device to be used in the VU framework, it needs to implement a driver that provides handlers for these routines. The RegisterProcessorFn() allows the developer to provide the main processing routine of the device which is called repeatedly by the framework. RegisterDataReciever() allows the developer to specify the properties of the incoming images (device's input) and provide the routine that receives it. RegisterGetConfigurationFn() and RegisterSetConfigurationFn() allows the developer to provide routines that responds to the applications requests for retrieval (GetParameters()) and assignment (SetParameters()) of internal parameters. The driver developer would also need to expose the parameter types to the application developer via a header file for these functions to be utilized.

Core layer

In order to implement the functionality of VU we have developed three frameworks that implement the sub-components discussed in Section 4.2.2. The VU Core layer simply makes calls to these frameworks based on internal events or user requests from the Interface layer. The implementation of the data access and the data convert components is done in a very lightweight manner as proof-of-concept and are covered here. However, for the data transport, we have developed a full framework that is presented in Chapter 5.

We have defined a light version of the image specification and device protocol. The image specification defines a number of data types that describe the content and the encoding of the image. Table 4.3 summarizes the image specification that we are using in our implementation. As can be seen we

Properties	Options
Image Compression	Raw JPEG PNG
Pixel Depth (Bytes)	1, 2, 3, 4
Pixel Type	Grayscale RGB BGR HSV YUV CMYK
Image Origin	Top Left Bottom Right

Table 4.3: This table provides a summary of the image properties included in our proof of concept image specification

have limited support for data types, but this description can be extended to include many other image and video formats. The device access protocol is based on a uniform set of asynchronous callbacks to the driver.

In order to perform data conversion we have developed a set of libraries to convert between different image types as defined by Table 4.3. We have wrapped the data conversion component in a simple function that takes an input image header, an output image header and input data, and provides data in the format requested by the output header. This is a very simplified view of data conversion and to implement a fully featured package requires further research.

4.3.2 System Development using VU

This section presents the proof-of-concept application that has been developed using the VU framework. In order to create a direct contrast between the VU framework programming paradigm and the current vision application development paradigm we have chosen to present a subset of OpenCV's functionality in this section.

We describe the components that make up this proof-of-concept system (the sources and the processor) and present the application program that uses these components. We then present the result of the application and contrast this approach to the conventional method of development.

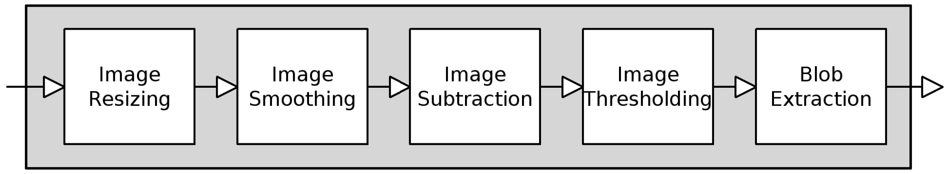


Figure 4.4: OpenCV based VU framework processor.

Sources

For this application we use two sources that employ very different physical protocols: An Axis 207 network camera that employs TCP/IP over Ethernet and a Logitech Quickcam pro 3000 that uses USB as a communication medium. We have developed drivers for these two cameras for use as sources with the VU framework. These drivers adhere to the data access image specification data and retrieval standards presented in Section 4.3.1.

Processor

We have designed a VU framework processor that performs a number of different tasks implemented using the OpenCV framework. The following is the functionality we have chosen to include in this processor:

- Resizing
- Smoothing
- Image Subtraction
- Image Thresholding
- Blob Extraction

These sub-processing components have been pipelined and collectively constitute the processor that performs foreground object detection. However, the user has control over activating components of the pipeline and by utilizing VU's functionality could perform any combination of these components on an image by processing it through the pipeline multiple times. Figure 4.4 shows the processor and its components.

In order to be utilized as a VU device, the processor needs to provide three components: a header file that includes the configuration and metadata types of the device; a set of handlers methods for the VU callbacks;

and a header file that provides functions to allow the application to access and configure the devices parameters and meta-data.

Code Snippet 4.1 shows the code that performs the task of providing handlers to for the callbacks. This code provides function pointers as handlers for the incoming image, get and set configuration, and the processing routine. Note that the SetDataReceiver call passes the input_header object that specifies the properties for the incoming image. The framework takes care of converting the incoming image to match those properties automatically.

CodeSnippet 4.1 VU device interface code. This snippet shows the process of registering callback handlers.

```
// Declare VUDevice
VUDevice vu_device(VU_DEVICE_PORT);

// configuration of VUDevice
vu_device.SetProcessorFn(OpenCVProcess);
vu_device.SetConfigurationFn(GetConfiguration);
vu_device.SetConfigureFn(SetConfiguration);
vu_device.SetDataReceiver(input_header, RecieveImage);
```

Table 4.4 shows the configuration parameters of the OpenCV processor constituting the second requirement for a VU device. In addition to the control structure (Active Modules) that determines the active components of the pipeline, each component of the pipeline has a set of configuration parameters that is exposed to the application developer. By manipulating these configuration parameters the pipeline and its components can be configured.

Code Snippet 4.2 shows the third and final requirement of the VU device; an example of a function that utilizes the set and get configuration functionality of VU application API to modify and access a particular device's internal parameters and meta-data. This code shows the function that sets the status (active or not-active) of each component on the device. It takes in the VU object, the processor device and the configuration data and configures the device using the appropriate 'command' and VU's setParameter routine.

Parameters	Configurations
Active Modules	Resizing Smoothing Image Subtraction Thresholding Blob Detection
Smoothing	Smoothing Type Smoothing parameters
Image Subtraction	
Thresholding	Threshold Value
Blob Detection	Minimum Size Maximum Size

Table 4.4: This table shows the parameters of the OpenCV processor

Application

The role of the application module is to configure the source and connect it to the processor. For this application we would like to display the processor's output on the screen under a number of different configurations to test the framework.

Assigning the Source The application can assign one of the two available sources to the processor. Depending on the user's input one of the two sources is chosen and configured to provide images at a specific resolution (e.g. 640x480 or 320x240).

Configuring the Processor In order to fully exercise the processor we have developed the application to take inputs from the user in runtime to configure the processor. Using the GLUT framework we are capturing key strokes from the user and setting appropriate flags that activate or deactivate components of the pipeline upon the invocation of the post process callback handling routine.

Displaying Processor Output In order to display images we are using the GLUT framework to create a 640x480 RGB window. To receive the output of the processor in this format we register a handler for the incoming image and specify the desired image properties (640x480 RGB). The image

CodeSnippet 4.2 This is an example of wrapper code that sets the activation of opencv components.

```
bool cvDeviceSetComponents(VU *vu, VU::Device processor,
                           opencvComponents &config_data)
{
  int psize = sizeof(commandType)+sizeof(opencvComponents);
  byte *pdata = new byte[psize];
  commandType parameter = CV_SET_COMPONENT_STATUS;
  memcpy(pdata,&parameter,sizeof(commandType));
  memcpy(pdata+sizeof(commandType),
         &config_data,
         sizeof(opencvComponents));
  bool res = vu->SetParameter(processor, pdata,
                              psize, VU::DeviceConfig);
  delete[] pdata;
  return res;
}
```

receiving handler copies the data into the GLUT display buffer and calls the display function.

4.3.3 Results and Evaluation

This section presents an evaluation of the VU framework based on the criteria presented in the introduction of the chapter. Through the following evaluation we demonstrate the following:

- The details of accessing data from sources is de-coupled from the processing module.
- The details of image data format of the source and processor are hidden from the user of the framework.
- The details of physical connection and communication of the source and processor module is hidden from the user of the framework.

In the following sub-sections we present and discuss the results in detail as they apply to these three points. In addition to directly evaluating the framework with respect to these three points we present the results of the overall system in order to verify the VU framework's ability to configure and communicate to the processor in run-time.

Source Access Abstraction

As presented in the previous section we have developed VU drivers for two different cameras: a USB camera and a network camera. In order to test the source access abstraction of the VU framework we have developed an application that connects each source to a processor and displays its output. In order to simplify the task and focus on the task at hand (source access abstraction) we have configured the processor to perform no operation on the image hence the output of the processor is the same as its input.

The two cameras have different native formats. The USB camera produces raw RGB images whereas the Axis 207 camera produces JPEG images. Although both cameras have a number of different configurable internal parameters, for demonstration, we only configure the resolution of the cameras.

Figure 4.5 show the output of the processor with the different cameras set at different resolutions, and Code Snippet 4.3 shows the code responsible for assigning a source to a processing pipeline. As can be seen from the code, the application programmer does not deal with any low-level details such as memory allocation, addressing or format change. The user simply picks a source from the list, has the option of setting the properties and assigns the source to the processor. This interface will remain the same regardless of the source; for example, image and movie files could also be represented as virtual cameras.

Image Detail Abstraction

An important contribution of VU is the abstraction it provides over the image representations inside the system from the user of the framework thus removing any effort that is associated with image data manipulation and transformation on the user's part.

In order to evaluate the framework's ability to perform this abstraction, we provide a number of different image resolutions and formats to each component of the pipeline. Since each component of the pipeline (the processor and the application) can only perform operations on a specific data type, the framework should handle the format changes to cater to the needs of each component. We evaluate the application developer's awareness of the change in image formats throughout the system.

The processor used for this evaluation is designed to strictly accept images with resolution of 640x480 RGB format and depending on its configuration can produce images of any resolution. The application has been designed to reconfigure the source to produce images with resolutions of

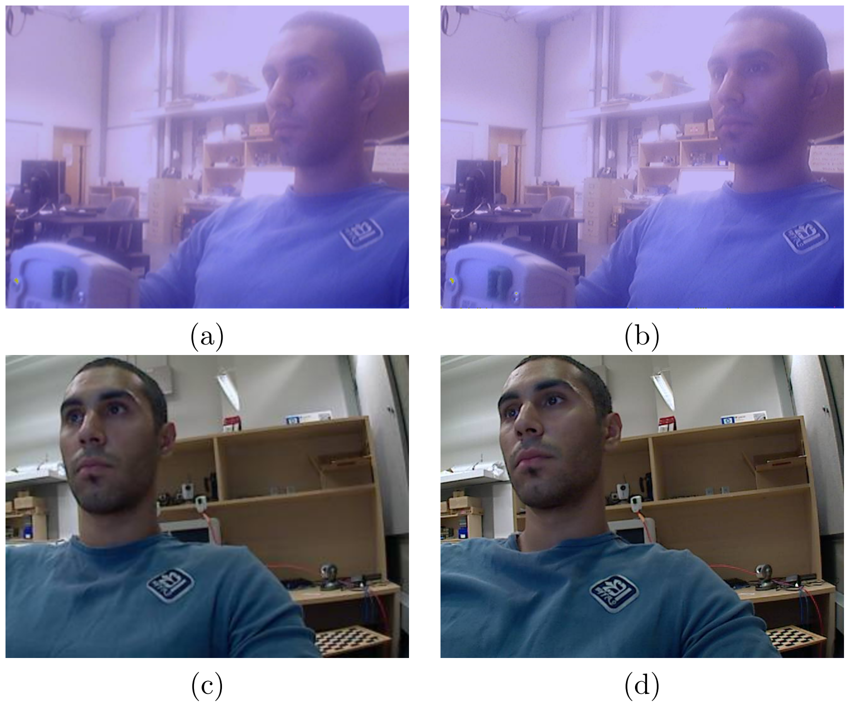


Figure 4.5: Output of the processor with the a) USB camera at 320x240, (b) USB camera at 640x480, c) AXIS camera at 320x240, (d) AXIS camera at 640x480.

640x480, 352x288 and 320x240 in both raw RGB and JPEG format and strictly display images of resolution 640x480 in RGB space.

We exercised the system by testing it under a number of different scenarios.

In the first scenario, the source is configured to provide images of resolution 640x480 in RGB space, and the processor is simply passing the same image to the application that displays it at the same resolution and type, hence eliminating the need for any format change (as shown in Figure 4.6a)). In the second scenario, the source is producing images of resolution 320x240 of JPEG format, the processor requires raw 640x480 images hence requiring a transformation by the framework to match the required data type for the processor (as shown in Figure 4.6b)). In the last scenario, the source is still producing 320x240 JPEG images and the processor has been configured to convert the images to raw 100x200 RGB format. Since the application ac-

CodeSnippet 4.3 VU code for setting up the source of a processing pipeline.

```
/* using the QCP camera */
vu.SetContextSource(devices.at(0));

QCPSettings qcp_settings;
qcp_settings.width = 320;
qcp_settings.height = 240;
res = QCFSetProperties(vu, devices.at(0), qcp_settings);

/* using the AXIS camera */
//vu.SetContextSource(devices.at(1));

//AxisSettings axis_settings;
//axis_settings.width = 640;
//axis_settings.height = 480;
//res = AXISSetProperties(vu, devices.at(1), axis_settings);
```

cepts only 640x480 RGB the framework converts the image automatically to match the required format for the application (as shown in In Figure 4.6c)).

Code Snippet 4.4 shows the code for registering image receiving handlers for both the application and the processor device driver. Using this API, the user simply states the required properties of the incoming image and the VU framework takes care of the conversion to that type. Using this framework all sources and processors are compatible and can be connected using the 'plug and play' paradigm.

Communication Abstraction

To demonstrate VU's ability to provide abstraction over the inter-module communication details, we present and compare the code for two systems developed on the VU framework. The components of the first system (source, processor and application) are on a PC running Linux OS. For the second system the source and application are running on the Linux PC and the processor is running on a Windows PC that resides on the same network.

The only difference in the two applications from the developer's perspective is the selection of the different processor from the list. The system developer treats both processing modules identically as the actual physical

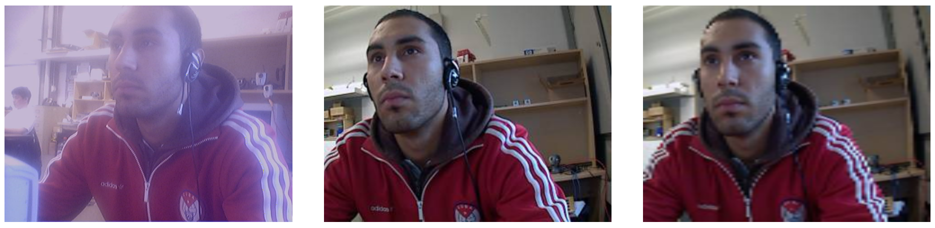


Figure 4.6: Output of the processor with a) Camera and processor raw RGB 640x480, (b) Camera JPEG 320x240 and processor raw 640x480, c) camera 320x240 JPEG and processor raw RGB 100x200.

CodeSnippet 4.4 Code for registering an incoming image handler.

```
UCF::ImageProp img_prop;
img_prop.width = width;
img_prop.hight = height;
img_prop.size = width*height*UCF::Byte3;
img_prop.image_format = UCF::Raw;
img_prop.pixel_depth = UCF::Byte3;
img_prop.pixel_type = UCF::RGB;

// VU Application
vu_application.RegisterImageReceiver(img_prop,image_receiver);
```

connection is completely hidden from the application developer via the data transport component. More discussion regarding data transport is presented in Chapter 5.

Run-time Processor Control and Configuration

To demonstrate the run-time control and configuration of the processor from the application we present the results of the application under a number of different controlling commands from the user. Figure 4.7 shows the screen shots of the system output with a number of different processor components activated.

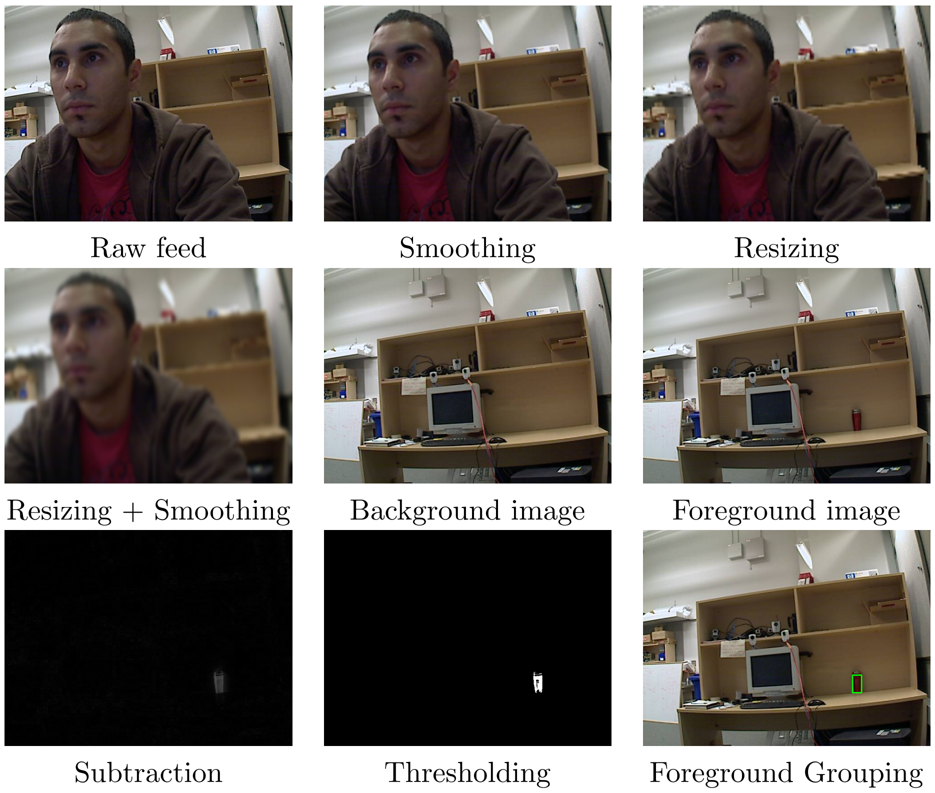


Figure 4.7: The result of the VU processor.

4.4 Conclusion

In this chapter we presented our approach of addressing the vision problem. Our approach is directly derived from the classification of the vision problem presented in Chapter 3. We presented the VU framework which provides the data management component of the vision task to vision developers while providing abstraction over the low-level details of data management sub-components.

We presented a proof-of-concept implementation of each data management sub-components and demonstrated how the VU framework utilizes these components and provides transparent access to their functionality to vision based application developers. We presented an application developed using the VU framework that utilizes two different sources and a processor based on the OpenCV framework.

We demonstrated that by using the VU framework, the user is not required to explicitly address the details of accessing sources, converting image data and transporting data in between modules.

Chapter 5

Hive Framework

In this chapter we present the Hive framework as a solution to satisfy computer vision's requirement for data transportation as described in Chapter 3. As presented in Chapter 4, Hive serves as the basis for the transport component of the VU framework. However, Hive is a standalone framework designed to provide the modularity and distributivity needed in vision system development.

The main goal of the Hive framework is to mediate *reusability* by providing *abstraction over inter-module communication* in a *platform independent* way. There is a direct link between support for data transportation and reusability that has inspired the Hive framework. The existence of a data transport mechanism is necessary to achieve code reusability since without a standard method for data communication between vision components there is no easy way of defining a standard interface for components to make them reusable. Reusability, abstraction and platform independence form the basis of our evaluation at the end of this chapter.

Data transportation for vision based systems is a complex task and to create an all encompassing solution it requires that has a large set of requirements be fully met which is well beyond the scope of this. We have mainly focused on satisfying a set of key requirements in developing Hive while not addressing hard real-time or global synchronization requirements. Following is a list of the requirements addressed by Hive:

- **Abstraction and Encapsulation**: Low-level communication details should be hidden from the user. This feature also decouples the implementation of the framework from its API.

- **Plug-in Interface**: Hive modules should be standalone (not require any other modules) and reusable to emulate the successful operating system's paradigm of plug and play.

- **Flexible and Low overhead Communication**: The communication protocol needs to provide low-overhead direct connection between modules while being extensible and flexible to accommodate any possible distribution pattern for vision systems.

- **Centralized Control**: The modules and connection between the modules should be controlled centrally to allow for dynamic reconfiguration.
- **Cross platform**: The framework needs to be platform independent in order to be promote use, portability and interchangeability of modules. This requirement allows heterogeneous sensor systems to be easily developed.

This chapter presents an overview of the Hive frameworks, the architecture of Hive, and a discussion of the implementation of its components, highlighting how the architecture satisfies the mentioned requirements.

5.1 Hive Overview

In this section we are introducing the Hive framework by presenting its application development model, its components and its framework model.

5.1.1 System development Model

Hive is a modular framework based on the concept of encapsulated and distributed processing. Hive systems consist of a single application and a number of drone modules which allows developers to rapidly create vision systems by reusing modules. Hive provides control routines to the application and mediates all of the communication between the modules by creating a structured peer-to-peer network. Using Hive's communication model, drones can be set up as a pipeline, a distributed network, or any combination of the two. Figure 5.1 shows a simple Hive system that performs human tracking using a background subtraction and a tracking drone. The image data in this case is provided by the camera drone. The application creates the data connections between the drones for processing and receives the output of the tracker. The following sections present the components of Hive: the application, the drone and swarms.

5.1.2 Modules

The following is a description of the different types of modules that make up Hive systems:

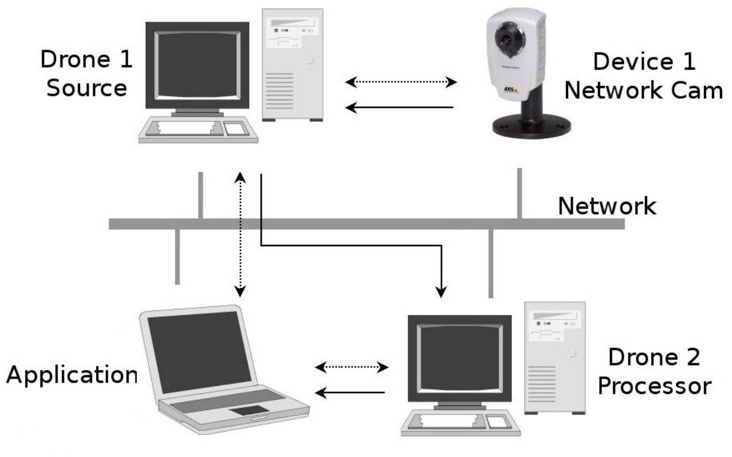


Figure 5.1: An example swarm set up using Hive to accomplish a vision processing task. Adapted from [1]

Application

The application module is the control center of Hive based systems. The application provides access to Hive modules to the system developer. The role of the application is to configure and connect drones together to create single or multiple swarms.

Hive supplies an Application API for the system developer that provides access to functions that configure drones' internal parameters and create data connections for drone-to-drone and drone-to-application communication. In addition to the initial configuration and setup, the API allows an application to connect drones to itself to receive data from drones. This data may be the results of the vision processing component of the system that is used by the Hive application or it can be run time information about the status of each drone. Hive applications can reconfigure drones or the swarm dynamically at run time if needed.

The application accesses drones using a universal addressing scheme. Using this addressing model, drones can be connected to the same physical machine as the application or be scattered across the network.

Drone

Drones are independent, reusable modules with a well defined interface that carry out one or several specific tasks. Drones can be based on a physical device such as a graphics card or a software routine. However, regardless of the nature of the drone, the interface remains constant. The drone's interface consists of the specification of the input data types, the internal configuration parameters, and the output data types. The configuration parameters of a drone allow a programmer to customize its function. For example, a camera drone's parameters can be the resolution and frame rate. Drones can be pure data sources (such as cameras), pure data processors (such as a background subtraction module), or a combination of the two (such as smart cameras).

Hive provides a simple interface for developers to create drone modules that run a software routine or interface with a physical hardware device. The Driver API provides routines for receiving data, performing processing, sending and receiving configuration parameters from applications, and sending data to other modules in the system. This API allows developers to easily create drones which can be connected to any other drone that adheres to its interface.

Swarm

Hive swarms are a set of inter connected drones controlled by a Hive application. Swarms consist of sources that produce data and processors that process the data. They can be set up in a variety of configurations to accomplish a single or a series of complex vision processing tasks. An application can set up multiple swarms simultaneously using different drones or employ a single drone in more than one swarm.

5.1.3 Framework Model

The Hive framework is a hybrid of two well known architecture patterns: Pipes and Filters (stream processing) and event driven architecture. Each of these patterns have a set of advantages and disadvantages. By combining them we are leveraging the benefits of each approach in order to satisfy Hive's requirements.

The Pipes and Filters model for stream processing is a popular model for signal processing that allows for parallel processing and distribution. In this model *filters* are processing units with well defined inputs and outputs and *pipes* are connectors that transfer stream data among filters. The simple representation of filters (inputs and output description) in this model provides modularity and easy reusability. Furthermore, as filters are independent units, they may be utilized in order to achieve parallel and distributed processing. Benefits of Pipes and Filters with relation to computer vision has been demonstrated in [16] where the authors use a modified Pipes and Filters model to successfully develop several vision based systems. The main disadvantage of the Pipes and Filters model with respect to vision based system development stems from the simplicity of the filters. As the model does not provide high-level management of filters, there is no mechanism for interactively accessing and manipulating the filter parameters. We have addressed this issue in Hive by modifying the communication model of the "filters".

Event Driven Architecture (EDA) describes a model of communication between components that is based on production and consumption of *events*. An event in this sense is defined as a change in state that is exchanged between a system's components[8]. The use of EDA for vision is not necessary; however, it provides a number of features that make it a very suitable model. The main advantage provided by this model is that system components are very loosely coupled since the event creator has no knowledge of the recipient(s). Due to the loosely coupled nature of its components, EDA can be easily distributed.

The Hive framework's architecture is based on a modified Pipes and Filters model which utilizes an event driven architecture for communication between components. This approach provides the advantages of the Pipes and Filters in addition to interactive control over the components of the framework. The details of Hive's architecture are presented in Section 5.2.

5.2 Architecture

Hive is based on a layered architecture inspired by the success of other layered designs such as the OSI model[44]. As successfully demonstrated by the OSI model, the layered architecture decouples services offered by the framework and provides abstraction and encapsulation of the implementation details of each service. In general layered architectures promote modularity and reusability as lower layers can be used by several instances of the upper layers, thereby reducing the complexity of development.

Figure 5.2 shows Hive's layered architecture for both the application and drone. There is a division in the architecture between the application and drone in the interface and Service layers. This distinction is necessary since Hive provides two API's with different functionality; however, the abstraction provided by the Communication layer is identical for both modules. In this section we present a detailed discussion of the motivation and design of

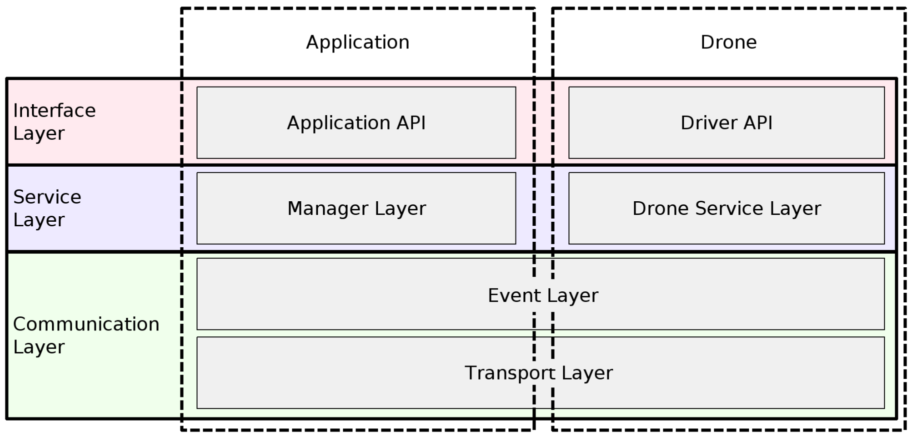


Figure 5.2: The layered architecture of Hive for both applications and drones.

each layer.

5.2.1 Interface Layer

The Interface layer provides Hive's application programming interface to users via asynchronous callbacks. Callbacks are chosen in this instance because they work well with the asynchronous event based nature of the system. The interface layer could also be implemented via blocking or polling methods. However, in these two cases the overhead on the user is increased as the user would need to manage the blocking call or poll for events manually. Hive provides two different API's to users: the Application API and the Driver API. Both API's share a subset of routines for setting up data handlers and the main method as well as a number of specific routines. The next three sections describe the Interface layer by discussing the API's for both the application and drones.

Application Specific API

The fundamental difference between applications and drones is the application's ability to manage drones and create swarms. The Application API provides the functionality to set and get the internal configuration parameters of individual drones. This can be done prior to creating a swarm or during the operation of a swarm. To create swarms the Application

API provides methods for connecting drones to each other. These connections specify the data type and can either be synchronized or streaming. The difference between the two connection methods is that for synchronized connections there is a request each time the data is required by the receiving drone. However, with streaming connections the data is sent whenever the sending drone has it available. There are also methods for connecting drones to the application and vice versa. The application can also send data and notifications to drones directly. The API for the application exposes the functionality provided by the Service (Manager) layer to the application developer.

Driver Specific API

The driver specific API is a thin layer that provides counterpart routines to the Application API and a drone's output. The Driver API allows drones to provide function handlers to respond to the set and get configuration requests. These handlers are invoked by callbacks when initiated by the application. The API also allows the drone to provide outputs by creating data of a certain type which is transported by the Communication layer to any drone that has registered to receive data of that type.

Common API

Since communication for both application and drones is done via asynchronous callbacks, registering callback handlers is the same for both interfaces. There are two types of callbacks for communication with modules: the *main routine* and *incoming data*. The main routine is a method that is invoked repeatedly by the Service layer and often carries out the main processing task of the module. For example, a camera drone's main routine is responsible for retrieving a new frame and passing it as output to the Event layer. The incoming data handlers are routines that are registered by the modules to process a specific data type. These routines are invoked by the Service layer whenever a data of that type arrives.

5.2.2 Service Layer

The Service layer provides the functionality for the Interface layer for the drones. The Service Layer's functionality can be categorized into the following: using the Communication layer to send appropriate commands and data to other modules in order to achieve tasks required by the Interface layer; and responding to incoming commands and data by invoking the appropriate handler routines (registered via the Interface layer).

5.2.3 Manager Service Layer

Similar to the Service layer, the Manager Service layer provides the functionality of the API provided by the Interface layer for the application. The Manager layer uses the Communication layer to send and receive data to drones. In order to set and get drone parameters, the Manager layer sends commands to a specific drone and blocks until the expected response event is received. In order connect (or disconnect) drones together, the Manager layer sends commands to the recipient drone, instructing it to request (or cancel the request) for data from the sending drone. The Manager layer also provides routines for sending data and notification of a specific kind directly to a drone.

Drone Service Layer

The drone specific functionality of the Service layer is minimal; mainly the registration and invocation of the callback routines that handle the configuration of the drone. In addition to configuration, the Service layer also passes the output data to the Event layer for potential delivery to other modules.

Common Services

The Service layer maintains and manages the callback handlers for both the application and drones. It provides routines for registration (from the interface layer) and invocation (from itself or the communication layer) of these handlers. Each handler is registered for a specific data type, and is invoked by the Service layer when data of that type is received through the Communication layer. The callback handlers are queued and called one at a time to maintain simplicity and prevent synchronization issues on the user's part.

5.2.4 Communication Layer

The Communication layer manages the transfer of data and commands between modules in Hive via a peer-to-peer network using an event based model. Event based communication is suitable for Hive since vision data is represented and processed as discretized packets. The communication model is based on a publish/subscribe model which uses an asynchronous messaging model. The peer-to-peer model is chosen to avoid communication bottlenecks that can occur in client-server models where the data from all components is directed to a central server for processing. Combining the flexibility of an event based mechanism with the efficiency of peer-to-peer communication allows for a system that is loosely coupled and scalable to very high number of modules.

The Communication layer consists internally of the Event and Transport layers. The following is a discussion of the details of these two layers.

Event Layer

The Event layer provides the functionality needed for managing connections between drones as well as sending and receiving events. The event based communication between drones is based on a publisher/subscriber model, where the sender does not explicitly send messages to receivers but the message is delivered to the recipient if it has registered for the message's data type. This paradigm decouples the communicating modules allowing the sender to operate at its maximum potential regardless of the receiver's performance. The event based communication provides flexibility and extensibility for the Communication layer.

The Event layer handles two types of events: network events and module events. In both cases the Event layer uses the functionality of the Transport layer to deliver events to their destination. Figure 5.3 graphically demonstrates the flow of data and events through the different layers of the Hive framework.

Network events are commands to manage interconnections between modules. These events are issued by the application module to drones as requests for creating or removing uni-directional data pipelines between two modules. The 'connect' event instructs the recipient drone to register itself as interested in data of a specific type on the drone that will produce the data, hence creating a uni-directional pipeline from the sender to the recipient. There are two types of pipelines: persistent, which transports events continuously; and non-persistent, which only transports a single event.

Module events carry data or notifications between drones. These events can be sent directly from the application to drones or transferred between drones using the uni-directional pipelines (as described above). The data transferred between drones is managed by the Event layer as described previously; drones produce output data of a certain type and pass it to the Event layer which then delivers the data to any module that has registered

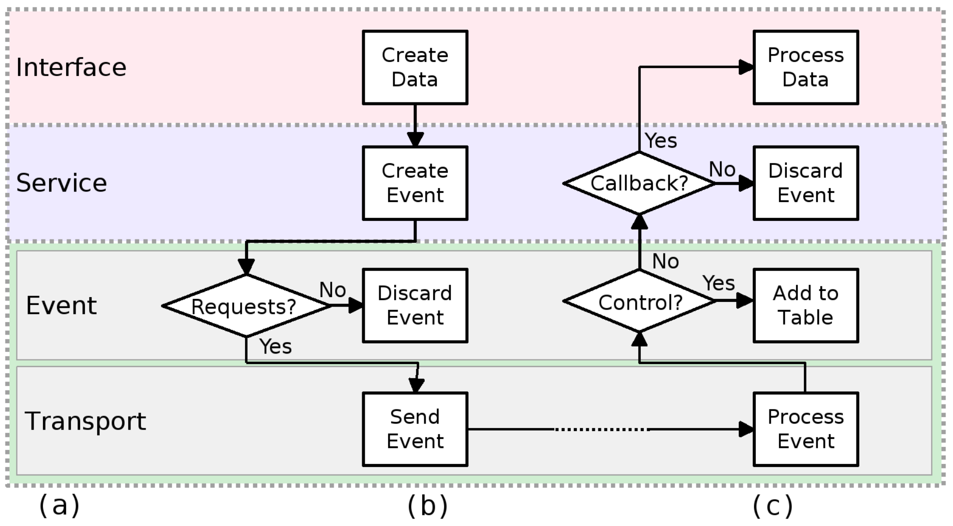


Figure 5.3: The flow across layers associated with sending (b), and receiving (c) data across the network.

for data of that type.

Transport Layer

The Transport layer can receive packets from and deliver packets to other modules. The send and receive functions are complementary and implement the delivery of incoming and outgoing events as complete packages. The receive routine is implemented as an asynchronous callback to the Event layer. The Transport layer was designed to implement a peer-to-peer network in a simple and lightweight manner. The communication is based on TCP/IP to allow modules to run anywhere on the network.

5.3 Proof of Concept

The main objective of Hive is to provide the data transportation and standardization required for vision (and other sensor) based system development that meets the following requirements:

- Component reusability
- Platform independency

- Abstraction over inter-component communication

In this section we focus on presenting a number of drones and systems developed using Hive to validate the framework by showing how it meets the above requirements. In order to focus on Hive's features we have isolated the transportation component in the following systems by dealing with the image data access and conversion within each drone. Furthermore, we utilize different platforms and sensors to show Hive's platform independency and ability to uniformly access modules regardless of underlying hardware or software platforms.

5.3.1 Implementation

We have developed a full version of the Hive framework to evaluate our conceptual VU framework presented in the previous section.

The implementation of this version of Hive is in C++ using the Boost library. Boost is a free peer-reviewed and portable set of C++ source libraries that provide standard API's for tasks such as networking, thread management and timing etc.[4]. The development of Hive follows the layer architecture closely to provide the same abstraction levels in code. C++ was chosen because it is the most widely used language for vision system development. Boost libraries allow us to easily provide a C/C++ interface for Hive and provide the platform independency we require.

5.3.2 System Development Using Hive

This section presents a diverse set of reusable modules (drones) implemented using Hive that form the building blocks for Hive systems. Each drone's functionality is described as well as its inputs, outputs, configuration and processing method. The drones are categorized into the following three types: data capture, processors, and visualization and storage.

Data Capture

This section presents a number of data capture drones that act as a starting point (sources) to swarms in Hive. We discuss the functionality and the abstraction of these drones. Note that the drones that produce images in this section follow the UCF image specification described in Section 4.2.2.

AXIS Network Camera : AXIS network cameras are standalone units that interface directly to the network via Ethernet and host a web-server that provides access to the camera's internal parameters and image data via the AXIS VAPIX protocol over TCP/IP[11]. The AXIS drone abstracts the VAPIX protocol by implementing the translation between Hive commands and protocol to VAPIX. This drone can run on any machine that is connected to the network which the camera resides on.

Inputs:	*None*
Outputs:	*Colour image*
Configuration:	*Camera settings; Output format (JPEG or RGB)*

Logitech Quickcam pro 3000 : Quickcam pro 3000 is a USB webcam that has a number of configuration parameters for image quality and resolution. Logitech provides a driver that allows access to image data and the configuration. The Hive drone for this device implements the translation between the Hive and native protocol device. The only difference from Hive's perspective between this source and the AXIS network cameras presented previously is the configuration parameters.

Inputs:	*None*
Outputs:	*Colour image*
Configuration:	*Camera settings; Output format (RGB or HSV)*

Image Sequence : Often when testing algorithms the same sequence is used for evaluation purposes. In some cases the actual data capture device might not be available and pre-recorded data is needed. The image sequence drone provides this functionality by loading an image sequence from the disk thus allowing seamless switching of data sources (e.g. from a live camera to an image sequence stored on any computer on the network). The drone itself can load any image sequence stored on its local machine. The root name of the sequence and the frame rate to supply data are given as configuration parameters.

Inputs:	*None*
Outputs:	*Colour image*
Configuration:	*Root filename; Frame rate*

Video Files : This drone fulfils the same purpose as the image sequence drone, but for video files. This is currently implemented using OpenCV and thus supports the codecs installed on the system. The frame rate used is the same as that used for the video file.

Inputs:	*None*
Outputs:	*Colour image*
Configuration:	*Filename*

Fastrak : Spatial position is an important aspect of many vision applications. Vision based algorithms for estimating 3D position require intensive processing and are often inaccurate. This task can be performed easily and accurately using tracking hardware. The Polhemus Fastrak[35] is a magnetic device that performs real-time six degree of freedom tracking of sensors. Fastrak provides the 3D position and orientation of each sensor (pitch, roll and yaw) relative to a base station. The Fastrak drone implements routines for getting and setting configuration of the tracker, start and stop routines and the get data routines. The only parameter on the tracker is the number of connected sensors. The device allows up to four sensors to be connected simultaneously. The start and stop routines allow the application to control whether the drone is producing data.

Inputs:	*None*
Outputs:	*3D position and orientation for each sensor*
Configuration:	*Number of active sensors*

Processing

This section presents the processing drones that we have developed using Hive.

Background Subtracter : Many algorithms in Computer Vision make use of background subtraction (or foreground extraction) as a precursor to the main computation. For example, silhouette images are required for visual hull construction and are useful for tracking, object detection and virtual environment applications. This drone provides eight different methods of background subtraction, ranging from simple frame differencing to more sophisticated techniques[33]. Algorithm selection and parameter setting can be altered via drone configuration.

Inputs:	*Image*
Outputs:	*Foreground image; Alpha matte*
Configuration:	*Algorithm selection; Parameters of algorithm*

Face Detector : Finding faces in an image has become an ubiquitous application seen now as standard on many compact cameras and also available on some camera phones. This drone makes use of the face detection supplied with OpenCV, which utilizes a cascade of boosted classifiers using Haar-like features[5, 24]. For each input image the drone produces an array of rectangles corresponding to regions possibly containing a face.

Inputs:	*Image*
Outputs:	*Array of rectangles*
Configuration:	*Algorithm parameters*

Colour Point Detector : Locating colour points in images is a useful method for tracking objects of interest. This drone finds the center of a region in an image that corresponds to a certain colour. The image is first thresholded against the required colour and then the pixels left in the image are grouped into blobs. The centers of the blobs are then calculated for those that meet the preferred size criteria.

Inputs:	*Image*
Outputs:	*2D position of the coloured areas*
Configuration:	*RGB value of point; Min and max size of regions*

Convolution : Convolution is a common operation in computer vision that is the basis behind many filters. We have developed two drones: one that implements 7x7 convolution using software and another that implements it using CUDA on a graphics card[12]. The interface to both drones is identical; however, the performance is substantially different. In the graphics card implementation, the image and kernel are loaded into the graphics card memory and operated on using parallel processors.

Inputs:	*Image*
Outputs:	*Convoluted Image*
Configuration:	*Convolution Kernel*

Visualization and Storing

Displaying and storing image data is a vital part of vision based system development. The following describes drones set up to accomplish these tasks:

Live Video Viewer : This drone provides a display for incoming images and annotation tools to draw shapes (from other drones such as the Face Detector). Multiple instances of this drone can be tied to different drones providing real-time feedback at each stage of a swarm's computation, which is useful for debugging during development. For example, in the Face Detection application described in Section 5.3.3 separate viewers can be connected to the camera, the background subtractor and the face detector to monitor algorithm results.

Inputs:	*Image; Rectangles; Points*
Outputs:	*Video to screen*
Configuration:	*None*

Image Sequence Capture : As discussed above for the Image Sequence drone, capturing data from cameras is important for offline processing or algorithm development and testing. This drone accepts images and stores them directly to disk, saving them with a file name given via configuration. The Image Sequence Capture drone also supports saving images to video files instead of image sequences.

Inputs:	*Image*
Outputs:	*Images to disk*
Configuration:	*Root filename*

Video Capture : Video capture works in much the same way as Image Sequence Capture, although the incoming images are saved to disk as a video file. The file name for the video is supplied via configuration, as is the video compression format.

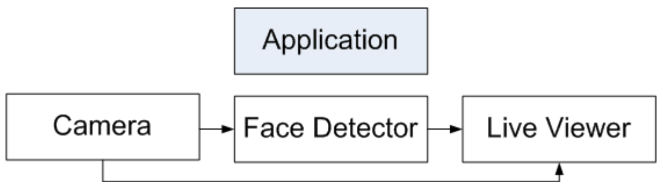
Inputs:	*Image*
Outputs:	*Video to disk*
Configuration:	*Filename; Video format (codec)*

5.3.3 Results and Evaluation

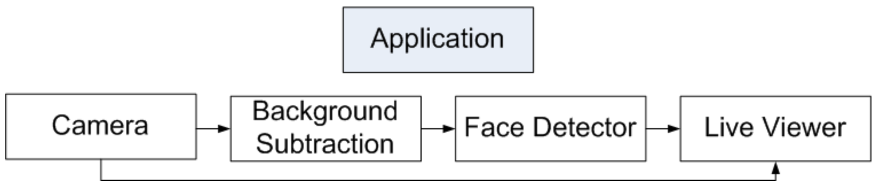
In this section we present a number of systems that have been developed using the Hive framework and the drones presented in Section 5.3.2. We present the results of these systems through which we demonstrate the component reusability, platform independency and seamless communication of the Hive framework. The systems in this section emphasize that given a base set of drones, system prototypes can be constructed quickly and swarms can be dynamically connected to test different configurations.

Face Detection

We have implemented a face detection system using the AXIS Camera, Background Subtracter, Face Detector and the Live Video Viewer. The system operates in two ways: the first detects faces on the original camera images and the second performs face detection on a foreground extracted image. Results demonstrate the improvement in accuracy and performance by incorporating a background subtraction system. The system itself shows the simplicity and flexibility of development using Hive.



(a) System 1



(b) System 2

Figure 5.4: **Face Detection**: Flow charts showing the connections in (a) direct detection and (b) the addition of a Background Subtractor drone. Taken from [30]

Both methods of face detection use a ‘dumb’ application to connect the various drones together. The application is termed ‘dumb’ because it does not need to do any computation or result collation itself as the drones perform all the processing.

The first method uses the application to connect an AXIS Camera to both the Face Detector and the Live Video Viewer and then connects the Face Detector to the Live Video Viewer (as shown in Figure 5.4(a)). Using Hive and the predefined drones, this amounts to under thirty lines of code (including configuration parameters). To obtain real-time performance the face detector is configured to be less accurate and faster. However, this results in more false positives. All of the drones and the application module for this setup run on one PC with Windows XP OS for this demo.

For the second system a background subtracter is inserted between the AXIS Camera and the Face Detector in order to reduce the number of false positives while maintaining real-time performance. In this instance, the Background Subtracter drone is running on another PC running Linux OS connected via the network. This new system, shown in Figure 5.4(b), removes identified faces from the background (such as photographs) as well as reducing the number of false positives.

The second system demonstrates Hive’s platform independency and ab-

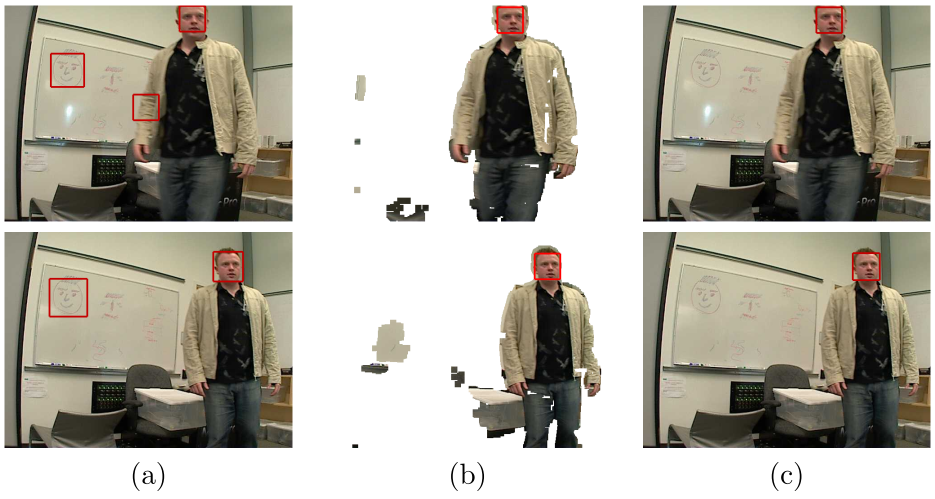


Figure 5.5: **Face Detection**: First row uses fast method, Second row the accurate method. Taken from [30]

straction over the physical location and connection of drones. The application does not need to address any communication or platform specific details.

The results of the two systems are shown in Figure 5.5, with and without background subtraction, and at two levels of accuracy. To obtain real-time performance the Face Detector is set to find faces with low accuracy, which increases the rate of false positives (Figure 5.5a)). Attaching a Background Subtracter to the system removes large regions of the image (Figure 5.5b)) where false positives can appear as well as removing static faces (such as photographs) from the scene. Figure 5.5c) shows the final result using the second system. The second row of images displays results for the system with the Face Detector in high accuracy mode.

The addition of the Background Subtracter drone to the system is simple and shows how systems can be enhanced or tested by inserting additional processes using Hive.

Quality of View Analysis

This system extends the previous real-time Face Detection algorithm to create a system which analyzes the quality of the given views in a multiple camera network. The quality evaluation is set to the number of faces in each view and the application automatically switches to the view with the most

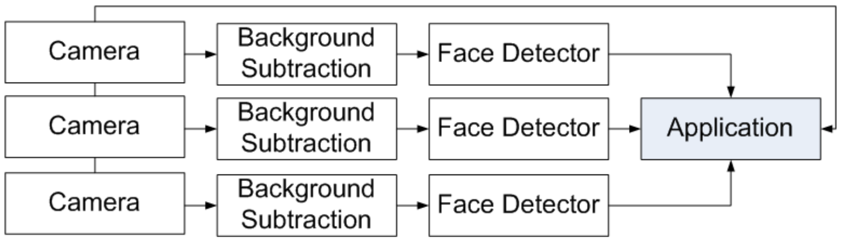


Figure 5.6: **Quality of View Analysis**: The flow chart for the Hive sensor network for analysing the quality of views via face detection. Taken from [30]

faces. This system for example could be used for home video podcasting in one-person shows; using multiple webcams the system will automatically change to the view the presenter is looking at.

The system connections are shown in Figure 5.6. Three AXIS Camera drones are each connected to a Background Subtracter drone which is in turn attached to a Face Detector drone. The cameras are also connected to the application to provide the images for the chosen view. As the feeds come in from the Face Detectors, the number of faces in each view is compared and the view with the most faces is chosen. Its images are then routed to the applications display.

This example demonstrates the reusability provided by Hive. We show how a sophisticated system can be built quickly using Hive from a set of base drones. Figure 5.7 shows the screenshots of the Quality of View Analysis system running for three scenes.

Multiple Camera Calibration

Applications such as tracking, augmented reality, camera calibration and human computer interface require a mapping between image pixels and 3D positions in the world. This system uses a magnetic positioning device to obtain the intrinsic and extrinsic camera parameters in order to calibrate cameras. There are various methods for computing camera calibration; we have developed a multiple camera calibration system based on the Tsai calibration method[42].

Using Hive we utilize the Colour Point Detector and the Fastrak drones. For this system we use a green marker on the Fastrak sensor to locate it in the image giving an image point to 3D point correspondence. To perform

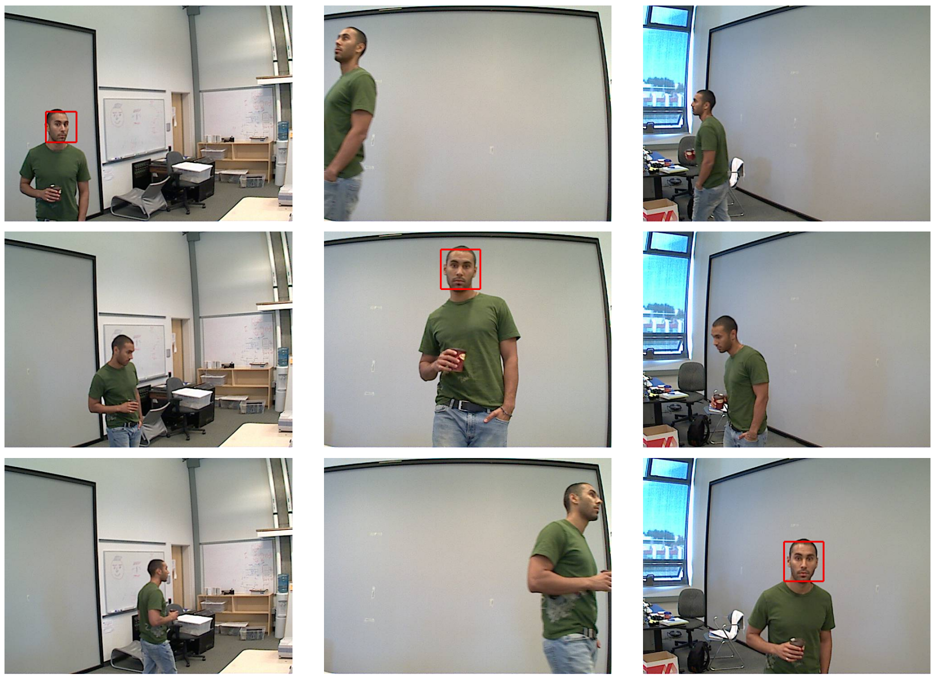


Figure 5.7: **Quality of View Analysis**: Each row represents a snapshot in time from each of the three cameras. The red boxes in the top-left, center and bottom-right images show positive detections and the view chosen by the system. Taken from [30]

calibration, the marked sensor is moved around in the field of view of each camera to produce a data set which is then processed using the Tsai method to calculate the intrinsic and extrinsic parameters.

Figure 5.8 shows the interconnection of drones in the multi-camera calibration system. The application is connected to one Fastrak and three sets of the Colour Point Detector and AXIS Camera swarms. The application couples the 3D sensor position from the Fastrak drone with the 2D location of the colour point from the Colour Point Detector drone and runs the calibration routine. The resulting calibration parameters are written to disk for each camera. Figure 5.9 shows the annotated images for each camera. Note that extension to more cameras is trivial, requiring an additional swarm for each camera.

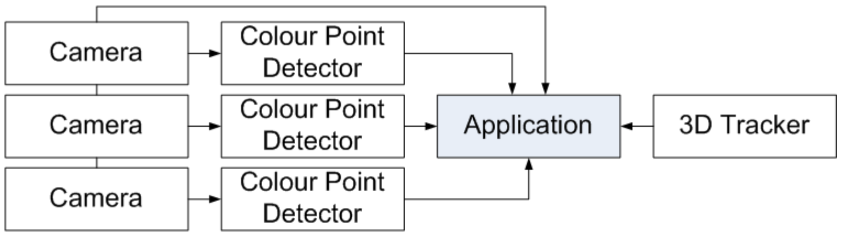


Figure 5.8: Drone interconnection for camera calibration. Taken from [30]

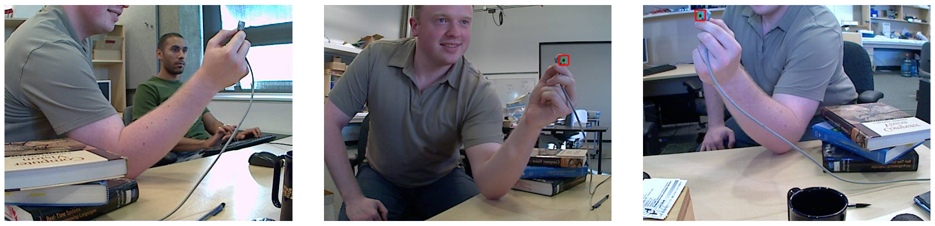


Figure 5.9: Feed from cameras 1, 2 and 3 during data point collection for calibration. Taken from [30]

Augmented Reality

The insertion of virtual objects into a real scene has many applications in entertainment, virtual reality and human computer interaction. We have implemented a real-time augmented reality system using the Fastrak drone and multiple camera drones that provides jitter-free virtual object insertion which is accurately represented in the different camera viewpoints.

Figure 5.10 shows the interconnection of drones for this system. We use the multiple camera calibration described above to calibrate the cameras to the Fastrak's coordinate system. The calibration system provides the location and orientation of each camera in the tracker's coordinate system as well as the camera's intrinsic parameter (focal length and the principle point). The calibration data is used to construct a model of the cameras and the coordinate system in OpenGL.

Given this model, a 3D object can be placed in the scene using the correct position and orientation supplied by the Fastrak sensor and rendered in the image plane of the modeled camera. This rendering is superimposed on the actual camera feed to produce the images that contain the virtual object. Figure 5.11 shows the frames from the three cameras before and after the

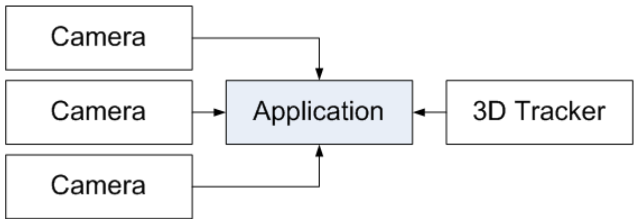


Figure 5.10: Drone interconnection for augmented reality. Taken from [30]

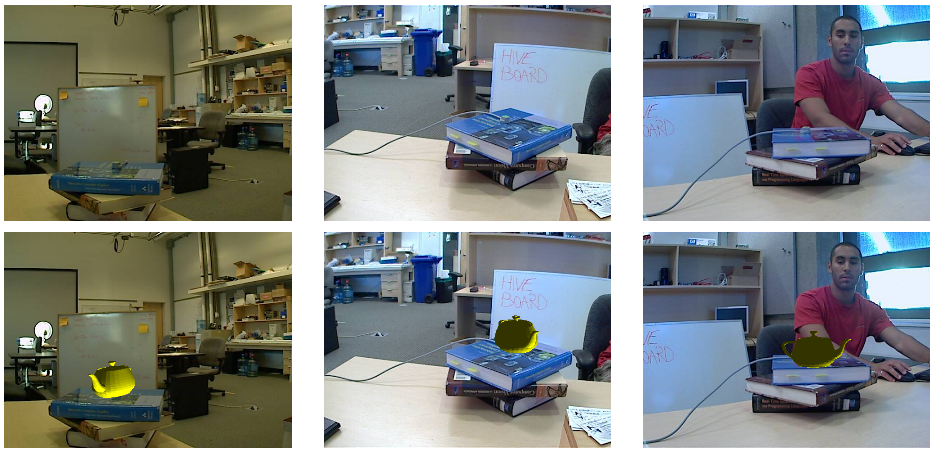


Figure 5.11: Original feed from the cameras vs. augmented reality. Taken from [30]

placement of the augmented reality object. Figure 5.11 shows the setup used for the calibration and the augmented reality system.

5.4 Conclusion

In this chapter we presented the architecture and the programming model of Hive; a component based framework for mediating communication and services needed for development of distributed vision systems.

We presented the Hive API which allows developers to create reusable source and processing modules called 'drones'. Using control modules called 'applications' the 'drones' can be connected in virtually any configuration to create sophisticated vision systems. We described the architecture of Hive as a set of layered services to provide abstraction and to decouple the architecture from implementation. We showed that by defining a clear interface for each layer, Hive provides increasingly high-level services that uses the functionality of underlying layers without relying on particular implementation. At its lowest level, Hive provides a flexible, low overhead Communication layer that follows the peer-to-peer interconnection model to allow for the creation of systems that are distributed and scalable.

We present a number of source and processor drones that have been implemented using the Hive framework. We show that using these drones, we can create a number of different systems easily in order to show modularity and reusability.

Chapter 6

Conclusion

In this chapter we revisit the material presented in this thesis in three parts. Firstly, we revisit the problems that we identified in the introduction and addressed throughout the thesis. Secondly, we summarize the contributions that have been made with respect to those problems. Thirdly, we propose a number of suggestions for the continuation of this work and the future direction of this area of research.

6.1 Problems with Current Approaches

The major focus of the work presented in this thesis is to address two important requirements that are not fully addressed in current approaches to vision based application development: abstraction over low-level details and high-level module reuse.

We identified that the underlying reason that these two issues have been previously overlooked is the lack of classification and conceptual abstractions in current approaches to computer vision based system development.

6.2 Contributions of this Work

In order to address the lack of conceptual abstraction we firstly separate the vision problem into the data management task and the processing task. We further decompose the data management task into the following decoupled components:

- **Data Access**: Configuration and retrieval of data from sources
- **Data Transformation**: Conversion and transformation of image data
- **Data Transportation**: Data transfer mediation between modules

We proposed that a framework for vision development should provide the data management functionality which consists of the data access, transformation and transportation sub-tasks.

Based on the above decomposition we presented VU framework, a framework that provides the data management functionality to vision developers while providing abstraction over low-level details. We demonstrated how the VU framework simplifies the developers task through abstraction of device access, image data details and communication between components.

In addition to the VU framework, we presented the Hive framework, an event based framework for developing distributed sensor systems that provides simple high-level methods for the communication, control and configuration of the reusable components. We discussed the details of design and architecture of Hive as well as a number of modules (sources and processors) and applications developed using Hive. We showed that even though Hive is completely independent of the VU framework, it forms the basis of the data transportation component of VU.

6.3 Future Direction

During the course of the research presented above, a number of future directions have emerged for the continuation of this research topic. We summarize these directions here.

6.3.1 Extension of Vision Utility Framework

We have identified four components of vision processing and presented the functionality and scope of each component. However, we only focused in detail on one of the components (data transportation). We have proposed an approach for the remaining components (data access and data transformation) and shown that this approach is valid for a subset of the functionality, providing a comprehensive solution. However, it is a non-trivial task that still requires in-depth research.

The VU framework in its current rendition has been designed as solely a proof-of-concept framework that supports a limited set of functionality. The main limitation of VU is that it only addresses a single vision context with one source and one processor. In order for this framework to be utilized as a successful vision based application development framework, it would need to support contexts with multiple sources and processors. In order to achieve this, the API of the framework needs to be extended to fully exploit the transport component by supporting more flexible connection configurations amongst the modules. Another feature that could be added to the VU framework is the addition of an auto-discovery feature for devices.

6.3.2 Extension of Hive

There are a number of future extensions in order to improve the Hive framework both in terms of added functionality and improved performance. We discuss two immediate possible additions to the framework: support for task distribution and an extension to the transport layer.

Hive currently provides the mechanism for transparent communication of modules. A possible extension to Hive that would further exploit this mechanism is to add build-in support for task distribution using the idea of pools of drones. The framework could accommodate dynamic allocation of drones to tasks based on the workload. The framework could provide the synchronization and control means for distributing the task and gathering results with minimal effort on the application developer.

Currently the sole transportation mechanism of the Hive framework is TCP/IP over Ethernet. However, the transport layer can be extended to support a number of different mediums such as shared memory and physical buses (e.g. USB and FireWire). Shared memory can drastically increase the performance for communication between drones running on processors that share memory banks, whereas USB and FireWire could extend the framework to be used for applications that require specific connectivity between components.

6.4 Conclusion

In this thesis we explored the current methodology for computer vision based system development in order to determine the underlying cause of the inadequacy of the existing frameworks. We proposed that the fundamental limitation with the current approach offered through various frameworks is the lack of conceptual high-level classification of the vision problem into smaller sub-components.

In order to address the lack of sub-task classification in computer vision we proposed a decomposition of the vision problem into the following decoupled components: data access, which addresses retrieval of image data from sources; data transformation, which addresses format conversion of image data; data transportation, which addresses the communication of data between modules in a vision system; and data processing, which addresses the analyzing and manipulation of image data.

Based on the above classification we presented a framework that provides the functionality of the data access, data transformation and data transportation components through an API that abstracts the details of each component from users. We described the programming model of this framework and presented an application as a proof-of-concept to validate this approach.

We focused on the transport component of the above classification and presented Hive, a standalone event based framework for developing distributed vision based systems that provides simple high-level methods for the communication, control and configuration of the reusable components. The main objectives of Hive are to promote component reusability, platform independency and abstraction over communication. We presented a set of modules and applications to validate the Hive framework.

The vision system development approach presented in this thesis could fundamentally change the way vision development is approached and could help advance the vision community as a whole through abstraction, standardization and promotion of code reuse.

Bibliography

[1] Amir Afrah, Gregor Miller, Donovan Parks, Matthias Finke, and Sidney Fels. Hive: A distributed system for vision processing. In *Proc. of the Int. Conf. on Distributed Smart Cameras*, September 2008.

[2] D. Arita, Y. Hamada, S. Yonemoto, and R. Taniguchi. Rpv: a programming environment for real-time parallel vision - specification and programming methodology. In *Proceedings of 15th International Parallel and Distributed Processing Symposium*, pages 218–225, 2000.

[3] 1394 Trade Association. Iidc 1394-based digital camera specification. Technical Report 1.3, 1394 Trade Association, 2000.

[4] Boost C++. http://www.boost.org/.

[5] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library.* O'Reilly Media, Inc., 1st edition, October 2008.

[6] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Oreback. Towards component-based robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 163–168, 2005.

[7] H. Bruyninckx. Open robot control software: the orocos project. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 2523–2528, 2001.

[8] K Chandy. Event-driven applications: Costs, benefits and design approaches. Presentation at Gartner Application Integration and Web Service Summit 2006, California Institute of Technology (2006).

[9] Szyperski Clemens. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Ltd., 1998.

[10] T.H.J. Collett, B.A. MacDonald, and B.P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Australasian Conference on Robotics and Automation*, 2005.

[11] Axis Corporation. Axis communication application programming interface:
http://www.axis.com/. Technical report, AXIS, 2008.

[12] NVIDIA Corporation. Nvidia cuda compute unified device architecture: Programming guide. Technical Report 1.0, NVIDIA Corporation, 2007.

[13] DevIL. http://openil.sourceforge.net/.

[14] Direct Show.
http://msdn.microsoft.com/en-us/library/ms783354(VS.85).aspx.

[15] David Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2003.

[16] Alexandre R. J. Francois. Software architecture for computer vision. In *Emerging Topics in Computer Vision*. Prentice Hall, 2004.

[17] Gandalf. http://gandalf-library.sourceforge.net/.

[18] GLUT. http://www.opengl.org/resources/libraries/glut/.

[19] ImageMagick. http://www.imagemagick.org/.

[20] Apple Inc. Core image programming guide. Technical report, Apple, 2008.

[21] Java Media Framework API.
http://java.sun.com/javase/technologies/desktop/media/jmf/.

[22] John Krumm, Steve Harris, Brian Meyers, Barry Brumitt, Michael Hale, and Steve Shafer. Multi-camera multi-person tracking for easyliving. *Visual Surveillance, IEEE Workshop on*, 0:3, 2000.

[23] Cheng Lei and Yee-Hong Yang. Design and implementation of a cluster based smart camera array appliaction framework. In *Proc. of the Int. Conf. on Distributed Smart Cameras*, September 2008.

[24] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *Proceedings of International Conference on Image Processing*, volume 1, pages 900–903, September 2002.

[25] R. C. Luo, Y. Chin-Chen, and L. S. Kuo. Multisensor fusion and integration: approaches, applications and future research directions. *IEEE Sensors Journal*, 2:107–119, 2002.

[26] Alexei Makarenko, Alex Brooks, , and Tobias Kaupp. On the benefits of making robotic software frameworks thin. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.

[27] MATLAB. http://www.mathworks.com/.

[28] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: Yet Another Robot Platform. International Journal on Advanced Robotics Systems. In *International Journal on Advanced Robotics Systems*, pages 43–48, 2006.

[29] Gregor Miller. *High Quality Novel View Rendering from Multiple Cameras.* PhD thesis, University of Surrey, UK, CVSSP, SEPS, University of Surrey, Guildford, GU2 7XH, 2007.

[30] Gregor Miller, Amir Afrah, and Sidney Fels. Rapid vision application development using hive. In *Proc. International Conference on Computer Vision Theory and Applications*, February 2009.

[31] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2436–2441, 2003.

[32] Netpbm. http://netpbm.sourceforge.net/.

[33] D. H. Parks and S. Fels. Evaluation of background subtraction algorithms with post-processing. In *IEEE International Conference on Advanced Video and Signal-based Surveillance*, 2008.

[34] George Pava and Karon E. MacLean. Real Time Platform Middleware for Transparent Prototyping of Haptic Applications. In *12th International Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 383–390, 2004.

[35] Polhemus. Fastrak :
http://www.polhemus.com/. Technical report, Polhemus, 2008.

[36] Quicktime. http://developer.apple.com/QuickTime/.

[37] Pekka Saastamoinen, Sami Huttunen, Valtteri Takala, Marko Heikkila, and Janne Heikkila. Scallop : An open peer-to-peer framework for distributed sensor networks. In *International Conference on Distributed Smart Cameras*, 2008.

[38] Michael H. Schimek, Bill Dirks, Hans Verkuil, and Martin Rubli. Video For Linux v4.12: http://v4l2spec.bytesex.org/v4l2spec/v4l2.pdf. Technical Report 0.24, Linux, 2008.

[39] Mark Segal and Kurt Akeley. The OpenGL Graphics System. Specification 3.0, The Khronos Group Inc., 2008.

[40] A. W. Senior, A. Hampapur, and M. Lu. Acquiring multi-scale images by pan-tilt-zoom control and automatic multi-camera calibration. *Applications of Computer Vision and the IEEE Workshop on Motion and Video Computing, IEEE Workshop on*, 1:433–438, 2005.

[41] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[42] Roger Y. Tsai. An efficient and accurate camera calibration technique for 3d machine vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 364–374, 1986.

[43] VXL. http://vxl.sourceforge.net/.

[44] H. Zimmerman. OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection. In *IEEE Transactions on Communications*, pages 425–432, 1980.

Appendix A

Previous Publications

The material presented in Chapter 5 (Hive Framework) has been previously published in the International Conference on Distributed Smart Cameras 2008 and International Conference on Vision Systems and Application 2009 [1, 30].

Appendix B

Statement of Co-Authorship

Parts of the work presented in this thesis were completed in collaboration with other researchers. The following list outlines my contributions:

- Identification and design of the research program:

  Collaborated on the classification of the vision problem.

  Determining the Scope of the Vision Utility framework.

- Performing the research:

  Design and implementation of the Vision Utility framework.

  Implementation of the VU proof of concept application.

  Evaluation of the VU proof of concept application.

  Scope definition and layered design for Hive's architecture.

  Implementation of Hives Event, Service, and Interface layers.

  Implementation of the Face Detections, Multi-camera Calibration, and Augmented Reality applications and drones.

- Data analysis:

  Evaluation of the Vision Utility Framework.

  Evaluation of the Hive framework.

- Manuscript preparation:

  Preparation of the entire thesis document.

Appendix C

The Vision Utility Framework's Application Programming Interface

The following is a description of the routines that constitute the VU API:

C.1 Application API

- **ContextID CreateContext(Device processor_id)**:

  Functionality: This method creates a VU processing pipeline that is associated with a specific processor.

  Arguments: Processor device

  Return: ID of the processing pipeline created

- **void SetContextSource(Device source)**:

  Functionality: This method allows the application to assign a source to the context that is already created using the CreateContext() method.

  Arguments: Source device

  Return: void

- **bool SetParameter(Device dev, void *data, int size, ConfigType type)**:

  Functionality: This routine allows the application to set the configuration parameters of a device or its driver (determined by ConfigType arguement). The driver parameter consists of manipulating the communication mechanism between the device and application.

  Arguments: Device ID, configuration data, size of the configuration data, target (driver or device)

  Return: Result of the operation

- **bool GetParameter(Device dev, void *data, int size, ConfigType type)**:

  Functionality: This routine allows the application to get the configuration parameters of a device or its driver (determined by ConfigType arguement). Similar to SetParameter() the driver parameter controls the communication mechanism between the device and application.

  Arguments: Device ID, configuration data (to be written into), size of configuration, target(driver or device)

  Return: Result of the operation

- **void SetIdleFn(IdleFn VU_idle_function)**:

  Functionality: This routine allows application to register an idle function that gets called by the framework repeatedly when there are no other events being processed.

  Arguments: Idle function routine

  Return: void

- **void RegisterPostprocessCallback(VUCallBack fn)**:

  Functionality: This routine allows the application to register a method that is invoked every time the processor has finished a cycle of processing on an input. The processor blocks until this call returns.

  Arguments: Callback handler routine

  Return: void

- **void RegisterImageReceiver(UCF::ImageProp prop, VUCallBack fn)**:

  Functionality: This routine allows application to register a handler for incoming images from the processor. This routine allows the application to specify the properties of incoming images. Images are converted to match these properties upon being received by the system.

  Arguments: Incoming image properties, handler routine

  Return: void

- **void PostReprocess()**:

Functionality: This routine signals the processor of a context to reprocess an input. This routine is called within the post-process callback by the application.

Arguments: void

Return: void

- **void PostDone()**:

  Functionality: Similar to PostProcess(), however this call signals the vision context to move on to the next input data.

  Arguments: void

  Return: void

- **void Start()**:

  Functionality: This routine starts the vision system. This routine requires that a vision processing context be set up prior to its invocation.

  Arguments: void

  Return: void

- **void MainLoop()**:

  Functionality: This routine blocks forever and allows the framework to invoke the registered callbacks in response to actions of the vision processing context.

  Arguments: void

  Return: void

- **void Cycle()**:

  Functionality: This is a non blocking counter-part to the MainLoop() call. This routine relies on the application for getting called repeatedly.

  Arguments: void

  Return: void

C.2 Driver API

- **void RegisterProcessor(NoArgCallBack fn)**:

Functionality: This routine allows the device driver to register the main processing method of the device. This routine gets called by the framework when there is an input awaiting processing.

Arguments: Processor function

Return: void

- **void RegisterGetConfiguration(DataCallBack fn)**:

  Functionality: This routine allows the device driver to register the routine that retrieves the configuration of the device. This routine is invoked by the application through a callback.

  Arguments: Get configuration routine

  Return: void

- **void RegisterSetConfiguration(DataCallBack fn)**:

  Functionality: This routine allows the device driver to register the routine that performs configuration setting of the device. This routine is invoked by the application through a callback.

  Arguments: Set configuration routine

  Return: void

- **void RegisterDataReciever(UCF::ImageProp img_prop, DataCallBack fn)**:

  Functionality: This routine allows the device driver to register a method that receives incoming image data. This routine allows driver to set the properties of the incoming image. This is an optional routine and is only used for processors.

  Arguments: Image properties, receiver routine

  Return: void

- **void SendOutput(UCF::ImageProp image_prop, char *data)**:

  Functionality: This routine allows a device to send image outputs to the application.

  Arguments: Image properties, image data

  Return: void

- **void Wait()**:

Functionality: This routine blocks forever and is used once the driver registers all the appropriate callback handlers in order to allow the framework to invoke appropriate callbacks to respond to incoming events.

Arguments: void

Return: void

Appendix D

Hive's Application Programming Interface

The following is a description of the routines that constitute in the Hive API:

D.1 Application API

- **bool SetConfig(ModuleID &id, byte *data, int size)**:

  Functionality: This method allows the application to set the configuration of drones. The application programmer must have the drone's header file that specifies the configuration specification for each drone.

  Arguments: Drone's ID, configuration data, size of the configuration data

  Return: Boolean result

- **bool GetConfig(ModuleID &id, byte *data, int size);**:

  Functionality: This method allows the application to get configuration data from a drone. The application programmer must have the drone's header file that specifies the configuration specification for each drone.

  Arguments: Drone's ID, size of the configuration data

  Return: Boolean result, Configuration struct

- **bool Connect(ModuleID &target, ConnectOptions &options, Connection::Type type)**:

  Functionality: This method allows the application to create data pipelines between drones, an overloaded method is also provided to connect drones to the application. The connection options provided by the user supplies the method and the data type for the connection.

The data type specifies what kind of data should be sent through this pipeline as a drone could produce a number of different outputs. The method specifies whether the data is 'synchronized' or 'streaming'. The difference here is that for synchronized connections there is a request each time the data is wanted by the receiving drone however with synchronized the data is sent whenever the sending drone has it available.

Arguments: Source drone ID, destination drone ID, connection options

Return: Boolean result

- **bool Disconnect(ModuleID &target, ConnectOptions &options, Connection::Type type)**:

  Functionality: This method allows the application to remove existing connections between drones, it is also overloaded to drone-to-application connections. Disconnect() requires the connection type in order to remove only the required connection, as there may be several connections between 2 modules.

  Arguments: Source drone ID, destination drone ID, connection options

  Return: Boolean result

- **void SendDataToDrone(ModuleID &mod, DataType &type, Data &data)**:

  Functionality: These methods (SendNotificationToDrone) allow the application to send data to specific drones. The difference between the two methods is that SendNotificationToDrone() only transfers the data type, where SendDataToDrone() also sends the data payload.

  Arguments: Drone's ID, data type, data(SendDataToDrone)

  Return: Void

- **void SetMainFn(MainFn main)**:

  Functionality: This method allows the module to register a main handler routine. This routine is called by the service layer repeatedly.

  Arguments: Main Routine

  Return: Void

- **void RegisterHandler(DataType &datatype, HandlerFn handler)**:

  Functionality: This method allows the module to register a callback routine that is called when the module receives data of a specific type. The data type is passed as an argument.

  Arguments: Data Type, handler function

  Return: Void

D.2 Drone API

- **void SetConfigFns(ConfigureFn GetConfig, ConfigureFn SetConfig)**:

  Functionality: This method allows the drone to provide the callback routines that get and set the internal parameters of the drone. These are the functions that are invoked when the application's getconfig/setconfig is called for a drone.

  Arguments: Handlers functions for configuration setting and getting

  Return: Void

- **void NewData(const DataType &type, const Data &data)**:

  Functionality: This method is used by the drone to send the output to other modules in Hive, This routine only requires the data type and data payload as the destination of this data is determined by the event layer.

  Arguments: Data Type, data

  Return: Void

- **void SetMainFn(MainFn main)**:

  Functionality: This method allows the module to register a main handler routine. This routine is called by the service layer repeatedly.

  Arguments: Main Routine

  Return: Void

- **void RegisterHandler(DataType &datatype, HandlerFn handler)**:

Functionality: This method allows the module to register a callback routine that is called when the module receives data of a specific type. The data type is passed as an argument.

Arguments: Data Type, handler function

Return: Void