Phission:
A Concurrent Vision Processing System Software Development Kit for Mobile Robots

BY

PHILIP DAVID SCHAEFFER THOREN

B.S. UNIVERSITY OF MASSACHUSETTS AT LOWELL (2002)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF MASSACHUSETTS LOWELL

Signature of Author: ______________ **Date:** Sept. 19, 2007

Signature of Thesis Supervisor: ______________

Signatures of Other Thesis Committee Members: ______________

____

____

Phission: A Concurrent Vision Processing System Software Development Kit for Mobile Robots

BY

PHILIP DAVID SCHAEFFER THOREN

B.S. UNIVERSITY OF MASSACHUSETTS AT LOWELL (2002)

ABSTRACT OF A THESIS SUBMITTED TO THE FACULTY OF THE

DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF MASSACHUSETTS LOWELL

2007

Committee Members:

Thesis Supervisor: Holly A. Yanco, Ph.D.
Assistant Professor, Department of Computer Science
Fred G. Martin, Ph.D.
Assistant Professor, Department of Computer Science
William Moloney, M.S.
Associate Professor, Department of Computer Science

Abstract - Designing and coding a computer vision system can require a developer to know a great many details about capture, display and operating system application programming interfaces (APIs). There are also several programming languages, development environments, and operating systems from which to choose. The Phission software development kit (SDK) supplies a single software package that provides for the previously mentioned choices. Phission removes the majority of the learning curve in vision system implementation and design by encapsulating the available APIs into interchangeable software modules for capture, processing and display. Phission is a concurrent, thread-safe, extensible, platform independent, and modular continuous video processing software package written in C/C++. Phission supports Python and Java, has automated dataflow (between connected software modules) during runtime, and supports multiple development environments, capture, display, and operating system interfaces. Phission is meant for use on mobile robots, desktop systems, or in embedded applications. Phission has been used successfully in robotic applications. The success of Phission's implementation of the system requirements for a computer vision system SDK is shown through applications that assisted in lab research.

Acknowledgments

First, I would like to thank Professor Holly Yanco for giving me the time to explore this thesis to my own obsessive ends. She helped me take this thesis from idea, to rough draft and finally to this final document by editing over a hundred pages multiple times. She also let me develop the Phission over several years for lab research and other various course projects. I would also like to thank Professor Fred Martin and Professor William Moloney for reviewing and editing my completed thesis.

Mark Micire, Kate Tsui, and Michael Baker also deserve credit for taking time to provide corrections as well as advice for not only the thesis but also the defense slides. They have also contributed to the development of my Phission software by developing software that uses it for research.

I would like to thank my parents (Evelyn and Glenn) for providing me with food and a place to live since I was born, letting me live at home and letting me go my own way. They provided a lot of encouragement during the writing of this thesis. My father helped greatly by providing advice on writing and helping me outline several versions of my thesis, in detail, over the past couple years. Both my grandparents (Evelyn and Dean) and my brothers (Glenn and Matt) provided encouragement and support even when it seemed like my thesis was never going to be completed. I would also like to thank Emily who provided me support by listening to the latest developments in my thesis pursuits and encouraging my progress towards finishing it. She was also very patient with my constant disappearances for many hours on weekends and the late hours I pulled as a graduate student.

There are a few people (not already mentioned) who helped during the development of Phission by providing suggestions, testing it and working on group projects over the past few years in the Robotics Lab at UMass Lowell. Andrew Chanler, “Bobby” Casey, Brenden Keyes, and Munjal Desai have worked on projects of their own as well as competing along side myself at the 2005 AAAI Scavenger Hunt competition. I apologize if I forgot anyone, but you can take comfort in knowing I appreciate every minute of help I got over the past few years.

Table of Contents

1 Introduction....1
  1.1 Motivation....2
  1.2 Theoretical Framework....3
  1.3 Contributions of the Thesis....5
  1.4 Outline of the thesis....5
2 Applications of Phission....7
  2.1 The History of Phission....7
  2.2 AAAI 2005 - Scavenger Hunt....10
  2.3 FLIR Video Overlay....16
  2.4 Car Tracking....18
  2.5 Blackfin Handy Board....20
  2.6 Other Applications....23
  2.7 Conclusions....27
3 System Requirements....28
  3.1 Capture, Process and Display Facilities....28
  3.2 Image Support....29
  3.3 Extensibility....29
  3.4 Modularity....31
  3.5 Portability....32
  3.6 Concurrency....33
  3.7 Dataflow....34
  3.8 Conclusions....36
4 Related Work....37
  4.1 Software Categories....38
  4.2 Vision Software Packages....41
  4.3 System Design Documents....47
  4.4 Conclusions....50
5 Phission....51
  5.1 Overview and Purpose....51
  5.2 Example of Use....53
  5.3 System Design....54
  5.4 Application Layer....59
  5.5 Framework Layer....60
  5.6 Filter Layer....69
  5.7 Image Toolkit Layer....71
  5.8 Dataflow Toolkit Layer....73
  5.9 System Toolkit Layer....78
  5.10 Phission's Design Methods....85
  5.11 Conclusions....93
6 Future Work....96
  6.1 Design Patterns....96
  6.2 Data Event System....96
  6.3 Generic Data Processing....97

6.4 System Toolkit Layer....98
6.5 Dataflow Layer....99
6.6 Image Toolkit Layer....101
6.7 Framework Layer....102
6.8 Application Layer....103
6.9 GUI Layer....103
6.10 Conclusions....103
7 Conclusions....105
8 References....107
A.1 SegmentationExample....115
A.1.2 Directory Listing....115
A.1.3 src/SegmentationExample.cpp....115
A.1.4 Makefile....118
A.1.5 include/SegmentationExample.h....118

List of Figures

Figure 1: iRobot ATRV-Jr....10
Figure 2: AAAI 2005 Scavenger Hunt objects....11
Figure 3: ATRV-Jr's perspective of the green trail at the AAAI 2005 Scavenger Hunt....13
Figure 4: Soccer ball segmentation....14
Figure 6: Output of filters in car tracking algorithm....19
Figure 7: The Blackfin Handy Board....21
Figure 8: Input images for the neural network....25
Figure 9: A room with the AAAI 2005 Scavenger Hunt objects....26
Figure 10: Nearest Neighbor auto-segmentation output....26
Figure 11: Phission layer diagram....53
Figure 12: Segmentation input frame collage....58
Figure 13: Segmentation output frame collage....58
Figure 14: Outline showing the structure of a Phission built vision system....60
Figure 15: Data-centric signaling and transfer of data....74
Figure 16: Byte-stream method of data transfer....74
Figure 17: Node-stream method of data signaling and transfer....75

1 Introduction

Phission has been developed and used over several years with support from the UMass Lowell Robotics Lab [UML 2007]. The project began as a means to create a vision system that would be easier to extend, improve use of the capture resource, and provide a low latency response in robotic computer vision applications. It is used by projects within the lab for computer vision research on mobile robots including search and rescue, assistive robotics, and other advanced student projects.

The project evolved as the development of new computer vision applications demanded more features or changes within the design of Phission. The initial objective was to provide an extension or modification of the lab's existing vision system software. As more applications were written, the system requirements necessary in a computer vision system SDK became more clear. It progressed from a semester project into a long term initiative to provide software that satisfied the system requirements of a computer vision system SDK.

Creating a custom vision system usually involves many hours to learn the interfaces that acquire images and display images. Existing software supplies mechanisms for doing this, but requires code to explicitly capture data and explicitly display that data. Different operating systems have their own development environments and interfaces that require time to learn. Developing concurrent programs also introduces a large learning curve as well as extensive hours of debugging. It is clear that learning a variety of interfaces for acquisition and display on different operating systems with different system interfaces to create a computer vision system that is concurrent can be an overwhelming task.

Developing vision systems for a networked mobile robot can be difficult. Desktop support in a computer vision system SDK is important before deploying on a mobile robot because it permits prototyping and debugging, two very time consuming activities when programming. There are also applications of computer vision that are not based on mobile robots that may be more applicable to a desktop. The research presented in this thesis was spurred by a need for a system that allowed unrestricted development facilitated by a live desktop

implementation where the code could remain unchanged and function in the same way on a mobile robot.

This thesis presents a set of system requirements for a computer vision system software development kit (SDK). An SDK should provide a developer an efficient means of constructing vision systems. The system requirements are necessary to provide for the available interfaces (acquisition, display, operating system) without requiring a developer to actually learn the interfaces. Instead, a layered modular design that encapsulates the interfaces allows a developer to develop the software system once and easily adapt it to a variety of platforms. Phission is a product of these system requirements and was developed to satisfy them.

1.1 Motivation

There had not been much design development on the vision system used in the lab because it was quite new. One design problem yet to be addressed was the serial flow of processing. The vision processing code ran in the same execution context as the robot control loop along with the analysis of other sensors. This processor misuse occasionally caused the robots to collide with obstacles due to the overhead of the compute intensive vision processing code. The code prevented the analysis of higher priority sensors involved in obstacle avoidance until the vision algorithms finished.

Another problem was the amount of integration between the vision system and the robot control application written in Python. The vision system and the control application were inseparable and required an understanding of coding hooks to C functions for use in a Python program. Learning to hook C functions into Python requires an additional learning curve and more effort in addition to understanding the use of computer vision in robots and implementing new vision routines.

Learning the APIs for using C functions in Python is unnecessary when there is a software tool such as SWIG [2007] that can automatically generate the Python modules given a C/C++ header file for an interface. Extending the vision algorithms to include newer

algorithms was rather invasive because one needed to edit the file where all of the other algorithms existed. The long term maintainability of the software required that the vision system be separated from the robot control software. Also, it could be specialized as a vision processing system development kit and allow extension with userspace code.

The first step was to check if there was an existing computer vision software development kit that addressed the aforementioned needs. Initial research into existing software did not yield anything that would solve the problem as desired. Reasons for not choosing an existing work at the time, and even now, center on several issues. If the software is unmaintained, underdeveloped, or not packaged very well, then it might require just as much effort to start from scratch as it would be required to learn or update that software. Some software target a very different research scope such as telecommunications, and is designed with the data types of that world.

Finally, potential software solutions simply were not conducive to use on mobile robots in an embedded type control software used within the lab. If the software did not support Linux, then it would require porting to Linux. If the software was integrated with a GUI, then that interface would need to be displayed over the network as part of a robot control application. All of the issues created an opportunity to create a specialized computer vision processing system software development kit and resulted in this thesis research. The Phission software, that supports this research, provides a standardized means of implementing computer vision projects within our lab and for sharing computer vision work.

1.2 Theoretical Framework

Cameras are increasingly being used with mobile robot, embedded and desktop applications by developers new to the field of computer vision. One reason is the decrease in the cost of cameras and image acquisition hardware. Another reason is the availability of open source computer vision software. The images being processed could be from a simulated or real world environment and could be live or previously recorded.

A vision system needs to support acquiring data, processing it by executing computer vision algorithms and possibly displaying it for user debugging, navigation or logging purposes. The actual algorithms are a peripheral concern here in that the system needs to provide data to the algorithms and execute them in a reasonable amount of time. There are many different types of image acquisition hardware and numerous application programming interfaces (APIs) that interface to that hardware. Regardless of number, the APIs serve the same purpose of providing image data for processing. There are many algorithms which can be applied to process an image and many software libraries available that supply these algorithms. Finally, there are also many software libraries or APIs for displaying images, each with its own design characteristics and procedures.

In mobile robots, cameras allow for perception of a visual world in an environment to allow actions such as obstacle avoidance, object recognition, target tracking and motion detection. Each one of these topics can be implemented as a specific vision system. When implemented on mobile robots, these systems are used to extract meta information from an image through use of segmentation. The meta information can support the ability of a robot control program in performing intelligent actions involved in such activities as manipulation or navigation of an environment [Russell and Norvig 2003 pp. 863-893].

Within the field of computer vision there are many subdivisions that can be explored. Some topics cover image analysis and processing algorithms that provide the brains to a computer vision application. The research discussed in this thesis focuses mainly on the system aspect of a vision system rather than the algorithms the system executes. Research revealed that no software was readily available to provide the ability to develop a vision system and deploy that vision system on platforms with differing acquisition, display or operating system APIs.

Software was needed to allow integration with a variety of robot control software (Mobility, Player/Stage and Pyrobot) and integration required supporting multiple programming languages (Python and Java) and operating systems (Linux and Windows). Our lab uses all of the aforementioned software, environments and platforms for Human-Robot Interaction, Assistive Technology, and Urban Search and Rescue research. The vision systems also

needed to be easy to use and capable of processing live data with practical frame rates for the applications researched by our lab.

1.3 Contributions of the Thesis

Phission provides an open source computer vision system software development kit that is concurrent, extensible, modular, and platform independent. It can be used on Linux, Windows or Analog Devices' Blackfin platforms and in desktop or embedded applications with very little difference in the code between each system. A layered and modular design provides plug and play capability for designing vision systems by allowing the linking of capture, process and display modules. Phission provides for automatic dataflow between the capture, process and display modules of a vision system and these modules are executed concurrently. These features are all available in a single software library that can be used in C/C++, Java or Python applications.

Phission provides a complete solution for rapid design of vision systems. These vision systems run at frame rate speeds that are limited by the number of filters in the system given the speed of the system's CPU. Phission was designed specifically with robotic applications in mind to allow for efficient live processing applications. No other open source package provides the extensive capabilities that Phission does in a single programming library meant for continuous live processing of data.

1.4 Outline of the thesis

Applications in which Phission was used are detailed in Chapter 2 to show the successful application of the software development kit and give an idea of the application scope. Chapter 3 outlines the system requirements and discusses, in more detail, the need for each requirement. The important theme that connects them together is that of making a developer efficient in researching or developing computer vision systems. Chapter 4 presents related works that are important to computer vision and the development of vision systems. Some related works were created after research for this thesis began, and they present architectural designs that can support the system requirements. Chapter 5 presents the Phission software in detail and finishes with a discussion of how the system requirements were satisfied. Future

work for the Phission software and the possibility of providing a better software development kit are almost unlimited but are briefly discussed in Chapter 6. Finally, the research contributions and conclusions are summarized in Chapter 7.

2 Applications of Phission

Phission has been used on mobile robots and in desktop applications to assist in the implementation and deployment of vision related applications. It was developed in concert with the demands of online/live processing applications. In this chapter we discuss several of these applications beginning with a time line of how they all contributed to the development of Phission. Larger applications/projects that have been used in published research are given a more thorough discussion in this chapter to provide supporting evidence of Phission's capabilities as a useful tool in constructing vision systems. These applications include the AAAI 2005 Scavenger Hunt, FLIR video overlay on USAR video, robotic arm visual servoing, car tracking, and a vision system for the new Blackfin Handy Board embedded system. These applications have all benefited from the ready-made components and component interfaces that Phission supplies. In addition, the applications benefit from the design guidelines for how Phission implements a vision system.

2.1 The History of Phission

Phission began as a solution to provide Pyrobot [2007] with a more efficient, concurrent, thread-safe and extensible vision system, because a search for existing software yielded no positive results. Phission's current design resembles its original design which included a capture module, pipeline module that executes filters and a display module. The original support included capture for only Video4Linux [2007], a few of the original filters present in Pyrobot's existing vision system (e.g. Gaussian, Canny, Sobel, etc.), an SDL [2007] display and pthreads [Nichols et al. 1996] for concurrency and thread-safety.

The plan consisted of making a vision system tool that was completely separate from Pyrobot so as to be specialized specifically for vision system creation. Each component of the system was going to be threaded because it was the simplest logical means to capture, process, and display concurrently. The C++ language was chosen because it was better for vision processing than Python (the language in which Pyrobot is written) and was object oriented which allowed for easier code reuse than C.

A good amount of the early development consisted of non-blocking spin loops and not the use of reader/writer locks. Threading was provided using the C interface of pthreads and was later made into a C++ class (from which the capture, pipeline and display classes now inherit threading functionality). Solving this problem was accomplished by creating a C++ class that encapsulated the signaling and transfer of data internally (see section 5.8.1). To remain completely thread-safe, everything was protected using mutexes. Once the system was stable and this class was integrated as a parent class of the image class, reader writer locks that greatly improved the performance of the entire application were implemented.

Eventually, it became necessary to start expanding the display capabilities of Phission. Being able to view multiple displays was not possible due to global variables and global state from the SDL library. Only one SDL display could be opened at a time. FLTK was the second display API class to be added in an attempt to allow multiple displays to be opened concurrently. Portability was a planned future addition to the Phission library and FLTK was chosen because it is a portable GUI library. However, the same issue occurred with FLTK as had occurred with SDL; global variables and global state prevented more than one instance of an FLTK display to be open. Finally, a new display was written using Xlib which allowed any number of displays to be open.

The first unplanned addition was a thread system management class. The original API required the developer or student to start the threads manually, which proved a bit confusing. From this confusion came a solution in the form of the thread system management class that takes care of properly starting and stopping threads.

The next unplanned addition was support for a Planar YUV format. The robots being used for Pyrobot applications were being run remotely and displaying raw RGB data over the network quickly impacted the ability of other students to use the robots. Planar YUV compressed the RGB data by removing a large percentage of the data required to transmit image frames over the network. To facilitate the transmission of the YUV image data, network sockets were added to the Phission library in the form of a server and client class.

The next big step in Phission's development came about as part of an Operating Systems course. Course projects required any student who had previously taken the course as an undergrad to develop code on Windows instead of a UNIX environment. All the code to complete homework for this course was written into the Phission classes. Any inconsistencies between the Pthreads and Win32 APIs for threading, synchronization and locking were removed or simulated using available components. For example, Windows does not have a condition variable that was being used by the Phission C++ thread class and did not support the canceling of threads in the same manner pthreads does. The functionality of a condition variable can be created using Win32 semaphores and locks and the cancel functionality was removed. With the port to windows came a display that used GDI and a capture class that used VideoForWindows. Windows support was provided using the Cygwin environment because it was the closest to a Linux environment whereas Visual C++ would be a drastic environmental change.

Java language support came from a desire to do image processing on the ATRV-Jr platform. The image processing was for Urban Search and Rescue overlay of infrared on a regular video stream. The interface to this platform is written in Java and uses the Java Media Framework [JMF 2007] to transmit video using the Real Time Protocol. Since Phission used SWIG [2007] to wrap its API into a Python module, doing the same for Java was simply a matter of removing any naming conflicts.

Support for the new Blackfin Handy Board was added to test Phission's ability to transparently port from Linux or Windows to a completely embedded system. VisualDSP++ uses a project file format for development which is different from the configure script and makefiles used in Linux and Cygwin. The port to VisualDSP++ took very little time after learning how to work within VisualDSP++'s environment. Support for capturing using the Parallel Peripheral Interface port on the Blackfin yielded a native capture facility. However, working with the PPI was not totally straight forward and a lot of effort went into debugging the actual capture process from an OmniVision camera.

Not all of the capture, platform and display support of Phission is being used by applications in our lab. For example, there are currently no applications which are making use of Phission on the Blackfin Handy Board. Windows is under utilized as a platform for development in our lab, too. However, the process of porting Phission caused inconsistencies and necessary abstractions to become apparent as well as platform incompatibilities. The applications that do exist can be moved from their current implementations onto these Windows and Blackfin environment.

2.2 AAAI 2005 - Scavenger Hunt

Figure 1: iRobot ATRV-Jr was used for the AAAI 2005 Scavenger Hunt operating platform.

The AAAI 2005 Scavenger Hunt was a mobile robot competition in which a robot had to follow a script of actions to find various objects using vision. Our ATRV-Jr robot system was entered into this competition to demonstrate the Phission vision system, a custom behavior system that controls the robot, and an interface developed as part of our lab's human-robot interaction research [Casey et al. 2005; Yanco et al. 2005]. Several behaviors for vision tasks were written using Phission to train on and track target objects. Given each of the objects, a behavior was coded to find that object. Many of the objects were single colors such as a

yellow beach ball, a set of colored bowls, colored cones, and colored construction paper (see Figure 2 for a photo of some of the objects). The construction paper was used to create a trail on the floor for which a possible real world application could be navigation through a warehouse using colored arrows. Given the characteristics of the objects, the implemented object recognition was limited to segmentation of a single color for an object. An exception to the single color object recognition was the soccer ball recognition code developed on-site and is discussed in more detail later.

Figure 2: AAAI 2005 Scavenger Hunt objects [Blank 2005].

Phission was used to show that writing code to implement a vision system within robot control code is beneficial in terms of reducing code complexity and providing mechanisms for faster robot reaction times. Robot control code can represent the vision problem using linkable modules which encapsulate a complex system that does not need to be completely understood by the developer. The behaviors of green trail following and tracking other objects consists of linking an input/capture module to a processing pipeline that contains a blob segmentation filter. The specific interfaces which acquire the images and the synchronization required to copy those images into the pipeline thread are encapsulated within the Phission components. Locating an object consists of loading the segmentation color values from a file and setting those values as the blob segmentation filter parameters. Loading color values from a file can even be done during the runtime of the vision system.

The Phission system constructed for this application consists of capture and blob segmentation filters in a pipeline system. The blob segmentation filter is given one color to

find each time an object is chosen from a list of available objects. The segmentation algorithm currently running can be stopped to change the color values if desired but is unnecessary since the blob parameters can be altered while the system is running. While the robot is in the color tracking mode, it will run the tracking behavior which checks if new segmentation data are available.[1] If there is no object identified in the segmented data, then the robot turns in place to scan the environment for the object.

The largest segmented blob is assumed to be the given object. The size of the blob has to be large enough to not be considered noise. When an object is segmented, that object is first aligned in the center of the input camera image by adjusting the robot. Then the robot is instructed to drive towards the object when the object is aligned. If the object becomes misaligned, then the robot will adjust itself left or right while still driving forward. Safety behaviors prevent the robot from actually running into anything while the tracking is being performed. The robot assumes that it has reached the object if it sees the object centered in the middle of the input camera image and the front laser range sensor value shows some object (any object) breaking the lasers beam. That logic resulted in some false positives where the robot had reached the object and people would walk in front of the robot during the scavenger hunt.

The HSV color space [Wikipedia 2007e] was used for training and tracking the colored objects for the scavenger hunt. Phission supports both the RGB and HSV color spaces. Either color space can be chosen and usually depends on the environmental conditions. It may also be the case that a specific color is segmented better using RGB rather than HSV despite environmental conditions. The fact that HSV is not as susceptible to varying lighting conditions (as much as RGB) made having HSV as an option very useful. The lighting conditions within the competition area varied greatly every several feet due to overhead spotlights.

1 The phLiveObject interface provides an *update* call that will check if new data has been placed within the object that contains the segmented data. Non-blocking and blocking mechanisms are provided by the phLiveObject interface because waiting can hold up the rest of the control code. The non-blocking choice allows the segmentation data to be analyzed only when new data has been posted.

The impact lighting had on the segmentation was also reduced by the robot's fluorescent lights whose main purpose is for lighting dark areas in USAR applications. The lights occasionally resulted in better segmentation by evenly illuminating the objects that were close to the robot. However, reflective objects such as the beach ball became completely saturated when the robot was only inches away when using these lights. The HSV color space thresholds had to be adjusted to have greater thresholds for the saturation and value channels when using the robot's lights.

The concurrent design of the vision system allowed the blob segmentation processing to be performed while the other behaviors were being handled, such as the laser navigation. This concurrency was efficient in that the other robot sensors usually require less CPU overhead to process than the incoming images. Waiting for the vision system to finish was unnecessary because a future iteration of the behavior processing loop will eventually get the data when the behavior is executed. If the vision system was capable of processing faster than the behavior loop, then it could have still been efficient in terms of latency. Only the most recent information is usually necessary for the behavior. If other behaviors were slow in the check for segmented blobs, there would not be queued information causing inaccurate world models resulting from old blob data because it would overwritten with the most recent data.

Figure 3: ATRV-Jr's perspective of the green trail at the AAAI 2005 Scavenger Hunt.

Using Phission to construct the vision system for the scavenger hunt allowed integration with the behavior system during one weekend. Most of the time consisted of acquiring color training data, tweaking the color thresholds for proper calibration, and thoroughly testing the

system for stability. Much of the development was done on a desktop machine and proper deployment required testing on the mobile robot.

The tracking behavior was able to track down the yellow beach ball and other single colored objects easily. The system was very successful at following a green trail created from construction paper (see Figure 3). The trail was made to be a wavy path with a loop at the end that allowed the robot to return to its starting position. It drove down the path and turned around using the loop to find its way back to the path it just came from. The soccer ball recognition also worked but was not demonstrated for the judges. The green trail following earned the system an award for path planning.

2.2.1 Soccer Ball

A soccer ball recognition algorithm was coded for a Pioneer robot at the competition. Recognizing a soccer ball was meant to provide a Pioneer robot with a task to demonstrate the sliding autonomy system being exhibited at AAAI05 [Desai and Yanco 2005]. The system was coded in Python for use in the Pyrobot [2007] environment that was used with the sliding autonomy demo. It took about 3 hours to design, implement and test the system to recognize a soccer ball within the same operating environment as the scavenger hunt. It has since been recoded and commented for use as an example of soccer ball recognition.

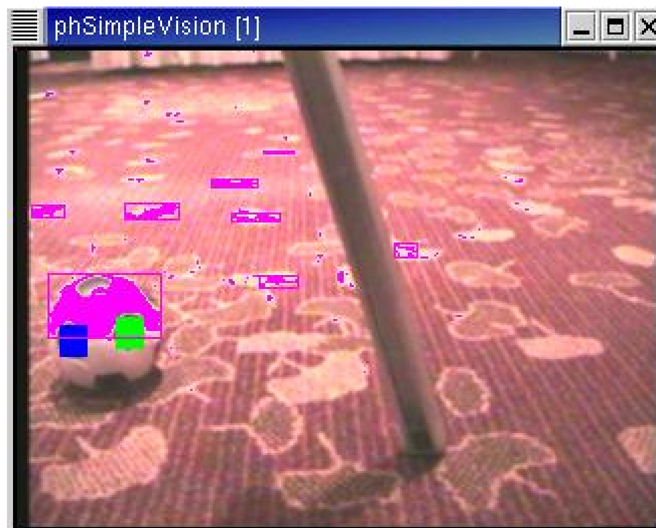


Figure 4: Soccer ball segmentation and recognition algorithm: Two black regions partially within a white region.

The recognition algorithm used looks for two different segmented color regions in proximity to each other. The two colors represent the majority color of the soccer ball sections and the singular color sections of the ball. When a large enough region of the majority color is segmented from the image, a search for the second color regions takes place. If two or three of the secondary regions are smaller than the first and lie within some threshold of the bounds created by the first region, then the first region is identified as a soccer ball.

There were two choices available, a colored soccer ball and a black and white soccer ball. The black and white option was chosen as the test of the segmentation capabilities. It was stated to be extra points if it was found in the scavenger hunt competition because it was an object not listed prior to the scavenger hunt event (shown in Figure 4). The soccer ball recognition algorithm could result in false hits if someone placed a white piece of paper with black regions in front the camera. False hits generated by such a method were not a concern because it was expected that no one would be running around to try to fool the algorithm. Further enhancement of the algorithm to recognize that the enclosing region is round would likely have reduced those false positives.

The Phission system constructed for this application consisted of a capture class, two pipelines, two blob filters, a couple draw rectangle filters and a display. In the C++ code, the capture class used was the V4LCapture class and the display class was the X11Display. In the Python code, it was the same but creation of the capture, pipeline and display classes are unnecessary since the phSimpleVison class was used to construct the simple vision system necessary. Each blob filter was initialized with a color representing one of the two regions to be segmented.

Planned future work was recognized through applications such as soccer ball recognition. The addition of a data pulse stream (similar to the one in MFSM [François and Medioni 2001, pgs. 35-49]) or a similar entity would allow one filter to attach all of its possible outputs to a single object so any component downstream could search for those outputs. The white and black region blob data information could be attached to the temporal object that gets passed between components (such as capture→pipeline, pipeline→pipeline,

pipeline→display). A generic soccer ball filter could be written to search for the named outputs from the blob filters and execute the soccer ball recognition algorithm on it. One of the drawbacks would be an increase in memory consumption but further discussion is left to the Future Work chapter.

2.3 FLIR Video Overlay

The FLIR overlay application reduces the transmission load that would occur with two separate images and reduces the cognitive load by fusing two images [Hestand and Yanco 2004]. The two images are the usual visible light image and an image captured from an infrared spectrum capable camera called a forward looking infrared (FLIR) camera.

Figure 5: FLIR Overlay showing heat from candles.

The method of overlaying the FLIR image on top of the visible image (see Figure 11) is to use an alpha channel. An alpha channel contains values for each RGB pixel and corresponds to a measure of transparency which gives a new RGBA format. The image to overlay on the original image is looked to for the alpha values. The overlay pixels are determined by the following equation (given in pseudo-code format for the red channel):

```
over[i].red = ((1.0 - flir[i].alpha) * orig[i].red) +
              (flir[i].alpha * flir[i].red)
```

The original pixel values are multiplied by one minus the alpha value and added to the FLIR pixels times the alpha value. Combining the two images in this manner allows for a level of transparency that highlights the visible image areas where the heat is detected rather than

completely blocking it with the FLIR image. A user can still use the image to drive while also looking for hot spots that may be a victim in an USAR environment.

A proper implementation would take into account the different lens characteristics of the two cameras and their orientations to properly align the two video inputs. For this application it was reasonable to determine the differences empirically using a heating pad to align to the two images. The FLIR image was shrunk before the overlay step because the lens characteristics cause it to have a narrower field of view than the visible camera. The pixels of the resized image were then applied to the region of the visible image that corresponds with approximately six to eight feet in front of the robot (the apparent focal point of the specific FLIR camera used).

The algorithm was implemented in C/C++ and compiled as a loadable Java module. It was executed on an ATRV-Jr robot being used for Urban Search and Rescue research and used the Java Media Framework RTP transmission facilities provided by the Java Media Framework for robust video transmission. In addition, the user interface was implemented in Java and required no modification to use the FLIR overlaying video since it continued to use the same method for receiving video.

The FLIR overlaying application showed the versatility of Phission because it could be integrated with the Java language. The application was able to be implemented as a standalone application in C/C++ and as a module at the same time. The standalone application provided the ability to debug and analyze the overlay algorithms. The FLIR example code is contained within the Phission code base as an example and is freely available. It has only been tested on Linux as it was the application domain. The examples are built by including a specially tailored example Makefile template. The makefile provides the ability to easily compile modules meant for any of the Phission supported languages (Java and Python). The Makefile that includes the special Makefile need only define the project files and that a module wants to be built.

2.4 Car Tracking

The car tracking application was an effort in "assistive robotics" and was meant for use on a robotic wheelchair or autonomous robots that operate in environments where they are required to cross streets. The car tracking application has yet to be deployed on a robotic wheelchair or autonomous robot for safety reasons. Allowing a mobile robot to drive onto a busy street using a street crossing system is in its beginning state of research could be disastrous. However, the initial research used the Phission toolkit to construct a vision processing system that tracked cars from recorded videos with close to real-time performance.

A different computer vision research tool called CVIPTools [2007] was being used prior to the use of Phission for the application. However, CVIPTools did not provide software for constructing the vision system and contained mostly computer vision algorithms. It did provide for reading images and displaying images but that has to be done explicitly. There was no support for reading from a video capture device. Also, there was no integration with threading libraries to automatically capture, process, and display captured images.

The method being used for the initial research consisted of grabbing a set of "good" frames from the video and storing them as files to be later read by CVIPTools. Although, using the "good" frames provided a means to begin the research and formulate most the ideas around the car tracking. However, that method did not allow for real world results. Real world results would require that the system process live data; a capability that CVIPTools did not provide and would have to be custom built if no toolkit could be found that provided the live processing capability.

Figure 6: Output of filters in car tracking algorithm. The final output (top-left) shows the system tracking a car approaching. The Mori scan (top-right) locates the bottom of the vehicle. A temporal mask filter is used (bottom-left) that removes noise pixels. A double difference temporal motion filter (bottom-right).

The car tracking research was being done at the same time Phission was being developed and researched. The existing code that used CVIPTools was separated into a set of filters (also known as processing classes) that allowed integration within the Phission system. The computer vision algorithms provided by CVIPTools were still used, but Phission added the ability to run these algorithms on live data. Phission removed the need for the developer to invest a large amount of time learning new programming interfaces (e.g., V4L, X11, or pthreads).

The architecture of the high level system components, thread-safe data transfer and synchronization is already designed so that the focus remained mostly on the purpose of the vision research. In addition, the use of Phission allowed for the car tracking application to be portable to different systems in the future whether it is a different operating system or a new capture facility. CVIPTools is the only dependency and would have to be ported as well.

The car tracking software system consists of a capture class, a pipeline with several filters and several displays. The following filters are processed in order (detailed in [Baker and Yanco 2005]): (1) Image differencing, (2) Noise filtering, (3) Edge extraction and thresholding, (4) Mori scan, and (5) Data reduction and history tracking. The output from several of the filters is shown in Figure 6. Double differencing Kameda [1997] is done for motion segmentation using an algorithm that takes three frames as input. The three successive frames produce two difference frames and the common differences in those two frames create one output frame. The noise filtering removes noise from the differencing algorithm. Edge extraction and thresholding is performed on the noise filtered motion image to get the outlines of any moving objects. The next filter is a Mori scan filter which looks for the bottom of cars using the idea that the darkest pixels in an image are beneath a car [Mori et al. 1994, pgs. 384-391]. The data reduction step analyzes the motion segments and Mori scan results to create bounding boxes around vehicles in the image. The final step labels these boxes and tracks them over time. The bounding boxes are placed with labels around the vehicles so that the display connected to the pipeline can provide a means to debug and view the results.

Phission allowed the developer of this application to move to live data processing quickly by providing all of the high level components necessary to do so. Only a basic understanding of threading and the transfer of data between modules is necessary. The developer can have confidence in the results of the algorithms by monitoring processed images as they are copied from the pipeline and displayed. The motion tracking and the "Mori Scan" could be run in their own pipelines. Separating those two tasks would require temporal stamping, temporal synchronization and multiplexing mechanisms which are not yet available. More research into systems which incorporate these facilities will allow future implementations to make better use of multiple CPUs and processing cores.

2.5 Blackfin Handy Board

Moving a working code base from one system to another system usually requires learning the specific APIs which are provided by the software development kits of the new system. New APIs can clearly provide frustration for a developer who wishes to take code from one

system and implement it again without completely recoding a new system and learning new capture, display or system APIs. After all, the goal is to research vision algorithms and implement an intelligent system which already requires extensive knowledge of vision processing algorithms and possibly artificial intelligence techniques.

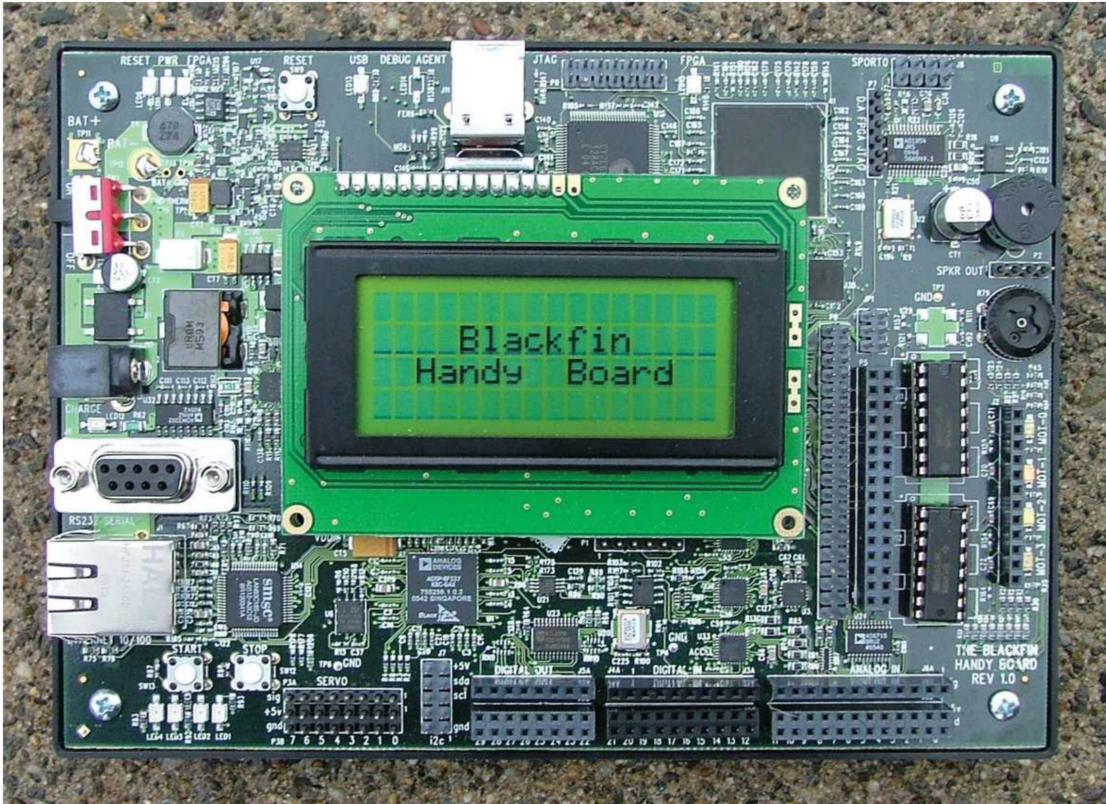


Figure 7: The Blackfin Handy Board from [BFHB 2007]

The Blackfin port of Phission provides for embedded applications because it supports the System Toolkit layer classes (see section 5.9) where the system has the same behavior on both a Linux and Windows system. The Blackfin EZ-Kit system was used for development of the Phission port and contains a Blackfin 537 processor clocked at 600MHz with 10/100Mbps Ethernet, 64MB of RAM [EZ-Kit 2007] and an OmniVision 7620 CCD camera. The Blackfin processor is a combination 32-bit RISC processor and 16-bit DSP core with a top clock speed of 600MHz [Blackfin 2007a]. The Blackfin has a Parallel Peripheral Interface that provides a convenient interface to control LCDs and Vision sensors using DMA and internal clock signals [Blackfin 2007b, sec. 7].

The development environment is the VisualDSP++ IDE which provides a visual development system capable of compiling code, loading the code onto the target and debugging or profiling the code while it is executing on the target in addition. The port to the Blackfin is targeted for the new Blackfin Handy Board, “a hand-held robot controller inspired by the original MIT Handy Board controller [Martin and Chanler 2007].” Applications can be developed on the desktop and then adapted to the VisualDSP++ environment (once most bugs have been resolved) and should result in greater productivity and less frustration.

The design of the Phission system provides for reuse of a vision system developed for one system to be easily reimplemented on another system using the pluggable top layer components. The majority of system dependent code is encapsulated within these portable classes. Any system which uses these Phission classes and refrains from system specific APIs can be easily migrated. A capture module for configuring and acquiring data from the OmniVision 7620 camera sensor is ready to swap with other capture classes such as the Video4Linux, VideoForWindows or Avcodec source modules.

The Handy Board system does not provide for a native display module. However, the Phission network display module (`NetDisplay`) runs well on the EZ-Kit and will be the main source of visual debugging from the Blackfin. With the availability of a network the processing outputs can be monitored using a simple network client that connects with the network source class. The network source class adheres to the protocol used with the network display class.

It is possible for the filter code to make use of specialized hardware which would change the scope of system dependent code. It is to be expected that filter classes will eventually contain specialized code for different hardware. The current algorithms that are distributed in Phission run on the Blackfin system as they are written in a system independent C/C++.

Porting a vision system developed with Phission from Linux or VisualStudio on Windows to the VisualDSP++ requires a minimal amount of work. A developer will have to create a

VisualDSP++ project. A bit of bootstrapping of the vision system code into an application that runs on the Blackfin and compiles within VisualDSP++ is needed. The bootstrapping can be thought of as an outer shell whose role is to execute to the vision system code.

The `NetDisplay` has defaults which do not require extra configuration but the network port can be specified for which a network client on a monitoring system, such as the desktop system used for development, will connect. In the case of the `phOmniVisionSource`, there are a few calibration settings which do not apply as they do with the `V4LCapture` component. These calls are simply ignored and have no effect. Some components such as the `VFWSource` (VideoForWindows) capture component have no API to calibrate the camera.

The ability to use the Handy Board shows how a vision system developed with Phission on a desktop system can be easily brought to the Blackfin. While it takes some new coding to bootstrap the code into a VisualDSP++ application, the majority of the project is left unchanged. Reuse of code resulting from the efforts of vision processing algorithm and application research is improved. A researcher also does not have to develop code that interfaces with the OmniVision capture module as it has already been supplied. A portable network streaming image JPEG server provides the processed image output monitoring capabilities that are otherwise missing from the Blackfin Handy Board.

2.6 Other Applications

Other applications Phission has been used for include a streaming HTTP webcam, static 20 second webcam, motion detection logger, Lamport Distributed Mutual exclusion system, teaching vision on mobile robotics, and Machine Learning research into Neural Network recognition of objects. These applications are either very simple in design or have seen limited success due to course time restrictions.

2.6.1 Webcam applications

The streaming HTTP webcam uses a simple daemon that captures images, saves them to a JPEG format which is recognized by web browsers and a Perl/CGI script to continually push new JPEG files to the browser. The speed at which the script pushes the images depends on

the speed of the client browser as it is a TCP connection. Continuous pushing of the images to the browser creates a streaming webcam. In addition to the Perl script pushing images, it also provided a form with buttons that allowed a user to send messages to a text to speech program which talked to the lab and buttons to move a custom pan-tilt servo.

The 20 second webcam consisted of three parts. On the front-end was a web page that refreshed an image of the lab every 20 seconds. There was a capture daemon that used Phission for capturing and saving images to JPEG files every 10 or so seconds. Finally, a back-end upload script copied the lab scene images to the web server on which the web page was hosted.

A motion detection program was also written to protect the lab refrigerator and as back up security in the lab. This program was also a continuously running daemon program and used more features of the Phission library. In addition to the capture and JPEG file saving, the motion program used image differencing, thresholding and blob segmentation filters. The program constantly differenced frames and thresholded the results to obtain a black and white image that highlighted motion. The motion regions were then segmented to create blob data. The main part of the program would wait for the Phission vision processing subsystem to update a blob data object with blobs of a certain size. When this object is updated it has the original image so it can highlight the motion regions and can then be saved to file for later inspection.

2.6.2 Course Projects

Phission is mainly used in mobile robotic or vision applications but the facilities it provides does not limit it to this scope. During an Operating Systems course, there was a project that required the implementation of a Lamport Distributed Mutual Exclusion System [Singhal and Shivaratri 1994 pp. 125-128] among several different machines. The Lamport application provided an opportunity to develop and test more of the Phission library that would otherwise normally not see much execution. A Machine Learning semester project was another similar project that experimented with training a neural network using video and processed video input nodes.

2.6.2.1 Lamport Distributed Mutual Exclusion

A Lamport system passes time stamped messages in a fully connected system of network nodes to determine which node will have access to a sink node that receives some type of data. The threading, synchronization and network components of the system were the only components necessary to implement a client in the Lamport system. The result of implementing this project was the removal of a few software bugs, stabilizing of the Cygwin/Windows port and enhancing the thread-safe network socket classes (`phSocket` and `phServerSocket`) to allow access to the sending and receiving methods by two different threads at the same time. This project also served to show Phission is not a one trick pony in that it can be used as a portable software layer for other types of threaded processing subsystems.

2.6.2.2 Machine Learning: Neural Networks and Auto Segmentation

Figure 8: Input images for the neural network. These were cropped by the training program given the hand calibrated coordinates (x1,y1) and (x2,y2) that represent the top left and bottom right of a rectangle enclosing the object.

The Machine Learning semester project used Phission for the implementation of a neural network (NN) processing system. The NN system is meant for the recognition of objects during live operation of a robot (see Figure 9 for an example of a room with objects). The objects are automatically segmented (see Figure 10) from the scene using a prototype auto segmentation algorithm provided by the `blob_Filter`. The project intended to use a neural network to identify trained objects in a scene while a mobile robot was driving around a room.

Figure 9: A room with the AAAI 2005 Scavenger Hunt objects; montaged frames from a movie used for live input to the neural network recognition system.

The project ended up with two programs. The first was a program that trained on the database of object images using a neural network library [FANN 2007]. This program output a file that described the neural network and could be loaded by the second program. The second program was meant for using the trained neural network and running segmented regions of live video frames through the neural network to attempt identification based on the trained network.

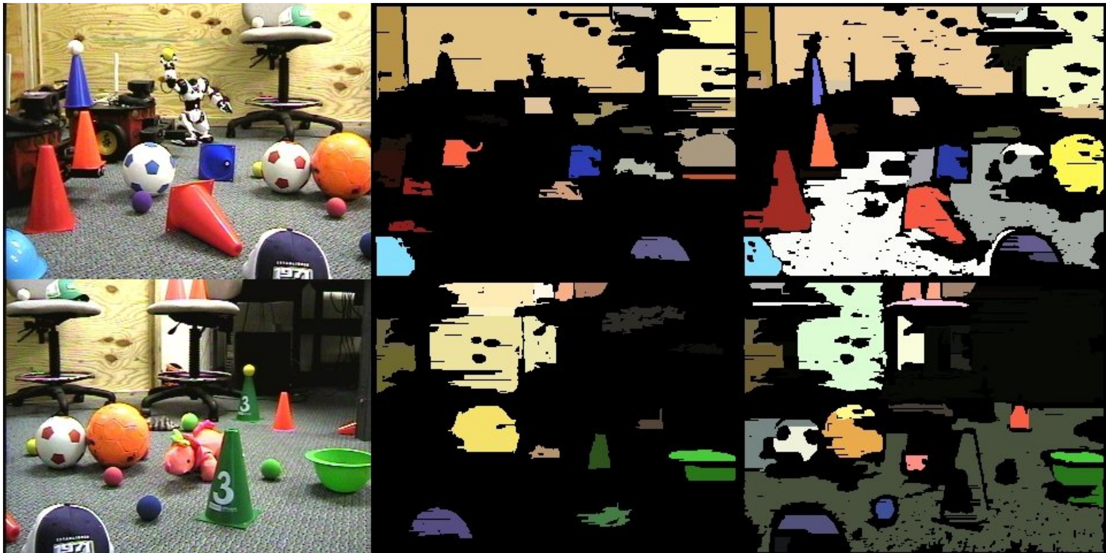


Figure 10: Nearest Neighbor auto-segmentation output (currently a prototype / alpha-stage algorithm) of a scene (left) using the HSV (middle) and RGB (right) color spaces. Uses a nearest-neighbor type algorithm to group similar neighboring pixels into individual blobs.

Some of the underlying algorithms in the automatic segmentation were ultimately the problem with many of the issues that resulted in the inability of the neural network to reliable identify objects. There is segmentation algorithm research which will be incorporated in the future to improve the segmentation accuracy. While the system may have not worked, Phission allowed most the time to be spent on the algorithms and the neural network design. Without Phission, the basic vision system components would have needed to be coded before any machine learning research or segmentation algorithm development could have been done.

2.7 Conclusions

Phission has been used to implement a variety of basic and mildly complex systems. The applications have shown Phission to be useful in removing the majority of effort required in implementing a live data processing vision system. Phission has been incorporated into a custom robot behavior system, car tracking application, back end vision processing for a USAR research robot, semester projects, and several different webcam applications. Each application has improved the system stability and shown Phission to be an incredible time saver with its ready-made components. The applications support robust live processing without a need for major redesign of the high level components, which provides supporting evidence in favor of the Phission system design and components. The ability to port Phission to other development environments, such as the Blackfin Handy Board, without requiring any changes to application level code shows the design of the operating system level components to be valid and useful in adding value early on in versions of a code base.

3 System Requirements

This chapter outlines and describes the system requirements of a vision processing system software development kit. These requirements were established through the application of vision systems to solve problems in a lab environment. The reasons for each system requirements are explained through their contributions to vision system research and development.

A vision system has three main tasks: capturing, processing, and displaying image data. These facilities must be provided and built on the foundation of the system requirements to provide an efficient vision system software development kit capable of live processing applications. The system requirements include extensibility, modularity, portability, concurrency, providing for automatic dataflow and coherent packaging.

An efficient SDK must be unrestrictive, provide all of the vision system facilities and allow for sharing of research and software. It must also allow a developer to quickly and simply perform the task of constructing a vision system. Efficiency, as it is used is this thesis, does not refer to the runtime of the software system. Rather, it is meant to express its ability to effect the productivity of the developer and the quality of the research produced. Without an efficient vision system SDK, developer research may be stifled by limited execution environments or an ineffectual research and development time line. It may even be abandoned later, due to restrictions such as only running on a specific operating system, or interfacing with a particular capture technology or display GUI. An efficient SDK is accomplished by providing for all of the system facilities with all of the system requirements satisfied.

3.1 Capture, Process and Display Facilities

From the highest viewpoint, a computer vision system accomplishes a task by capturing, processing and displaying image data. Acquisition of images in a live environment is usually provided by a hardware video acquisition device. Processing is done by applying computer vision algorithms on the acquired image data. Displaying requires a graphical user interface

facility that interfaces with the video hardware to display an image. Facilities for each domain must be available in a computer vision system SDK.

3.2 Image Support

A computer vision system software development kit should include code that provides for image representation to which other third-party image types can be mapped. It should include at least one format, but a variety of formats and conversions between those formats would be useful. Support for the representation of images is necessary because the components provided need to have a common model which ties them together. Image utility functions, such as resizing or cropping, are not necessary. However, they can be useful for such things as reducing the data size input to a processing filter or allowing a display window to fit the image data to a user requested resize.

3.3 Extensibility

Extensibility requires developing the SDK while taking into account unknown future applications [Wikipedia 2007b]. It is an efficient design attribute of any software development kit and provides the ability to extend it within the scope of its field of application. It is a mindset as well as a software engineering practice. Extensibility can partly be supported through the use of generic interfaces. Those interfaces allow access and control to the modules of the vision system. Each capture, display, or processing component can be implemented using the generic interface without knowing actual specifics of each related capture, display or processing interface.

The extensibility attribute ties into several other necessary attributes including modularity and portability. In fact, most of the system requirements are connected to each other. Extensibility (as used here) covers the ability of the software to be extended to a variety of programming interfaces, languages, development environments, multiple operating systems and other software projects. Supporting multiple operating systems is also covered under the portability requirement. Multiple languages and other programming interfaces are also covered by the modularity requirement. Extending or integrating with existing software

projects is a feature of an SDK that satisfies the requirement of being a coherent software package.

3.3.1 Multiple Programming Languages

Language extensibility allows the developer to choose other languages by providing wrappers to the native language of the vision system's SDK. The reasons for the choice of a secondary language over the native language could be based on technical merits, personal experiences, or an existing software project's demands. Java may offer better portable user interfaces, or Python may be preferred for the ability to write code quickly. A choice of language will allow a greater number of developers to make use of the SDK without requiring them to learn the native language in great detail. A developer can still create custom extensions to the SDK in the chosen language or in the native language of the SDK.

There is at least one drawback of language extensibility, which happens when another developer wishes to use the code developed in a secondary language with any language other than that secondary language. Any application code or algorithms developed in the secondary language may have to be ported back into the native language (for wrapping) or into one of the secondary languages. This drawback is not a concern of the current thesis research. It is left up to the developer to either write the extensions in the native language or back port the third party code.

3.3.2 Multiple Development Environments

The ability to use a particular operating system, programming language, or interface may be restricted to an individual development environment. Some embedded hardware platforms might not support compilation outside of a single IDE (e.g., VisualDSP++ for the Blackfin processor [VisualDSP++ 2007]). There are some display and capture APIs which are simply implemented using a certain IDE (e.g., VS2005, DirectX). Making the SDK extensible should permit it to work in any development environment. The two main types of development environments encountered are usually “Makefile” (e.g., GNU Make, Jam) or “Project file” (e.g., Eclipse, VisualStudio, VisualDSP++) type systems. Most software can be brought into any new development environment with some amount effort. An SDK that

already supports as many of them as possible is likely to have changed to make it easier for porting into other development environments. The way the SDK is packaged should provide no road blocks for developers to use it.

Not every individual developer is expected to be fluent with every development environment. Those who have never used a Linux system may not be familiar with makefiles or the Autotools [Vaughn et al. 2007] that make up a command line development environment. Linux developers may prefer a makefile over a visual integrated development environment project manager. An efficient vision processing system SDK must allow for the varying tastes of each developer, the demands of existing software, software interfaces, and different hardware by providing support for several development environments.

3.4 Modularity

Modularity is a method of designing the SDK so that the separate interfaces for capturing, displaying and processing images can act as black boxes [Wikipedia 2007c]. The vision system is constructed by linking the black boxes together, but apart from transferring data they do not interact with each other. Each capture, display or processing component acts separately to perform its own given task and should be swappable with other components of the same type. They only communicate using thread-safe transfers of data when necessary. The vision system can also becomes a modular construct where it can be developed for a specific task and included in a large application without interfering with normal program flow.

Creating a module requires encapsulation of code and allows the SDK to be more portable. Modules can be chosen for compilation, or compiled out, depending on system settings. For a capture module, the programming interfaces are used to acquire image data. Display modules use interfaces which provide a means of sending data to a target such as a screen or network client. Processing modules provide access to algorithms, process data, and may provide their own data outputs whether image or meta-information (such as segmented object coordinates).

The efficiency of the SDK is increased because a developer has ready to use black boxes that perform some action that reduce the time to implementation. It should reduce the slope of the learning curve on a developer to achieve a fully functional vision processing application. These modules can represent a significant amount of design, debug, and development time on the part of the SDK developer(s).

The design of the modules should use consistent mechanisms provided by the extensible generic interface. An example is the set of common routines that provide control over the brightness or size of a capture device. Extensions or exclusions to the common routines are allowed. An example of an extension would be for a network display which requires the setting of port information that a normal screen display would not need. An example of an exclusion would be if a capture device does not support the setting of brightness values, then the implementation could just stub out the function or method.

3.5 Portability

Providing portability allows for better design through the refining of the SDK interfaces to hide the differences between platforms. It is impossible to know all of the systems on which the SDK will possibly be used. A vision system built using a portable SDK can be coded for one platform and then brought to another platform with minimal effort required by the developer. Designing it to be portable will allow it to support new platforms with little extra effort in the future. Portability is facilitated by using extensibility and modular design techniques. The design of the system is done to allow for future ports. It permits compiling only modules (capture, processing, display, etc.) that are supported by the platform. In addition, the interfaces to the components of the vision system are encapsulated so the public interface introduces no platform dependency that hinders porting.

Portability adds a significant amount of inherent value to a developer's efforts. Instead of writing platform dependent code, a portable or platform independent API allows a developer to code once and run on all of the platforms supported by the SDK. If the development kit were to only run on one system it would severely limit the usefulness of any developer research that is based on it. A developer who is restricted to one operating system (for one

reason or another) may have to port the vision system. Having an already portable vision system benefits others performing related research in addition to oneself. Such a system can allow the state of the art to increase more quickly. Considering that research labs can contain a heterogeneous mix of different hardware and operating systems, building portability into the SDK will allow the same application code base to run mostly unmodified among these platforms.

Threading software interfaces (e.g., pthreads, Win32, VDK) differ from one operating system to another (e.g., Linux, Windows, VDK). Synchronization, capture, display and development environments also differ from system to system. There are software libraries that provide portable threading interfaces (e.g., Boost, the Apache Portable Runtime, pthread-win32, etc.), but this introduces another dependency to the applications. The SDK would be restricted to the specific design of the portable thread library and the supported operating systems of that portable threading interface. Porting will be required when a new operating system is desired that is not supported.

Integrating concurrency into the vision processing SDK interface allows a number of benefits. Integration allows threads to be designed in a way that allows for providing an extended interface. An extension can make the system more flexible (e.g., Phission's `wakeup` methods) by providing behavior that is useful but not standardized. Getting the system to execute on an unsupported operating system requires implementing the same control logic and similar functions calls within the portable system components provided by the SDK. It requires the same amount of effort as supporting two different portable third party threading and synchronization libraries without the added dependency.

3.6 Concurrency

Concurrency is necessary to achieve the best utilization of resources in a given system. An image acquisition can be taking place while an image is being processed and another image is being displayed. Concurrency is a necessary feature of vision processing. The concurrency can be achieved using multiple processors or time slicing a single processor. SMP, hyper-threading and multi-core systems can provide for true concurrency and allow the vision

system to scale without requiring the developer to change anything. Concurrency is built into the design of the system components through modularity and becomes an implicit feature of developer efforts. If the SDK does not provide for the concurrency, then it will be left up to the developer to supply his own.[2]

Lack of concurrency would result in a serialized processing system. This type of application captures a frame from the video hardware, processes the data, and displays any processed output. After displaying data, the system returns to acquire another frame from the capture device and repeats the loop. This design results in long processing latencies because the next frame could have already been captured and ready when the processing was finished.[3] Instead, the application has to wait the entire time the hardware is acquiring and transferring the frame from the hardware.

Concurrency requires the protection of any data that are accessed by more than one thread. Coordinating the sharing of data and copying of data between threads requires mutual exclusion features. The concurrency requirement is not solely satisfied by providing for threading or multi-processing. Rather, it requires that condition variables, semaphores, mutexes, and reader/writer locks be provided for completeness. Those synchronization features can then be incorporated in any upper layer structures that are meant for access by more than one thread.

3.7 Dataflow

Providing the capability to automatically move data between components is a necessary requirement of a vision system SDK. Dataflow is a result of using concurrency and modularity together. There is a need for image data to go from component to component

---

2 Concurrency could be custom coded to time-slice a single process by triggering capture and display tasks using timer signals. Ready-made solutions are more practical.

3 Perhaps the hardware is able to queue capture requests. Common display and capture video hardware can have memory transactions performed with Direct Memory Access(DMA). It is possible that as soon as an acquisition was complete another could be queued without requiring significant CPU overhead. When the processing and displaying are done, the next frame would be ready and waiting or almost complete. If processing takes more time than it takes to capture a frame or two in a serial application, queued frames become old data. It ends up not being a very practical solution and concurrency resolves queuing old information by allowing new frames to replace old ones through constant updating.

because a capture class that acquires image data is useless unless that data can be processed and displayed. The passage of an image from capture to processing to displaying is defined under the term dataflow [Wikipedia 2007a]. Dataflow should be automatic and coordinated by the modules using a specific protocol. Otherwise the developer would have to write the code that explicitly moves the data from one component to the other and that would break the modularity of the individual components.

There is plenty of software that allows one to process individual images but not many packages directed towards live input processing. However, applications that will use a vision processing SDK should be targeted towards eventual implementation in a live processing system. The demands of such a vision system require that data flows through the vision system automatically.

There are many different options for the transfer of data between modules that include pushing, pulling, indirect transfers [Manolescu and Nahrstedt 1998, pgs. 84-91], simple memory copy operations, and memory reference counting (this is not an exhaustive list). The number of options and which options are provided are not as important as the existence of at least one option that gets the data moved through the system. Having the one option provides the basis for implementing optimized dataflow operations later.

For example, data could be copied from one thread to another thread using simple mutex operations to prevent race conditions. An optimization to that solution is to provide reader/writer locks that allow several threads to be copying the data at the same time while preventing race conditions with the data writing/source thread. Another example is the use of CPU intensive memory copies to move the data from the private space of one thread to that of another thread. An optimization to the memory copying would be the use of memory reference counting and/or stream structures (such as those that are part of the MFSM/SAI architecture) to pass the data along and remove the majority of memory copies needed.

3.8 Conclusions

A vision system SDK cannot simply provide a means of supporting the requirements. It must actually provide the variety of extensible interfaces, provide support files for different development environments, provide the ability to generate loadable secondary language modules, implement several capture and display APIs, be used with a variety of algorithm libraries, implement support for more than one platform, and have support to execute on more than one operating system. The requirements are all interdependent in different ways and so any individual requirement cannot be removed. It is necessary for a vision processing system toolkit to maintain a certain set of system requirements to be efficient in constructing vision processing systems. The requirements give a large inherent value to developer research by requiring the SDK to supply everything needed to create the vision system. An SDK that contains some but not all of the system requirements is less efficient than one which has all of the requirements.

4 Related Work

This chapter is an overview of related work in the field of computer vision systems and constructing those systems. During the history of computer vision, there have been a great number of subfields that have developed, which is clear from the amount of resources available online and offline. This chapter does not provide an exhaustive list or complete description of all of these fields. Also, there will not be a discussion of single purpose ad hoc vision systems that constitute the majority of vision research, even though they are the intended applications targeted by a computer vision system SDK. Instead, the focus is directed to software and research that is useful for constructing general vision processing systems for which the previous chapter outlined the system requirements. It turns out that all of the system requirements are rarely provided or defined by an individual related work. Partial fulfillment is not enough to provide an efficient vision system software development kit because it requires significant effort to develop the missing features when implementing an application.

To the best of my knowledge, there is no software that has provided an implementation of a software development kit that assists in constructing general computer vision systems and supplies the full scope of the system requirements. There are a few software packages that do aim to provide software for creating computer vision systems; however, there is always at least one requirement missing. The drawbacks are usually associated with the effort it takes to construct a vision system based on the related software. The ability of the vision software to make a developer more productive could be improved by providing more support for creating a vision system framework.

Available software encompasses a large range of categories from general software toolkits to full blown scientific GUI applications. There is also a significant amount of related work that is not directed at computer vision but rather multimedia, providing portable toolkits for threading, video capture or displaying video. These related works contribute solutions that may later be integrated into a vision system SDK and should not be overlooked.

4.1 Software Categories

4.1.1 Image Processing Toolkits

An Image Processing Toolkit is software that provides structures for image representation and includes groups of functions for processing image data. Some of these toolkits provide capture, display, GUI, or image loading facilities in addition to their collection of algorithms. Those attributes allow the toolkits to be useful for developing vision system applications but still require more effort to add dataflow and concurrency. They generally are limited to the toolkit scope of software and do not provide the resources for constructing a vision system. Some examples of image processing toolkits are CVIPTools [2007], Gandalf [2007], LTI-lib [2007], CImg [2007], libvideogfx [2007], Mimas [2007a], and VIPS [2007].

“Toolkits don't impose a particular design on [an] application” [Gamma et al. 1994 pg. 26], which precludes the software in this category from being labeled as a computer vision system software development kit. Missing from these kits is threading or automatic dataflow mechanisms. These toolkits could be extended to include the components necessary to provide a computer vision system SDK. However, their focus has been to provide the implementation and optimization of a large number of computer vision algorithms involved in image format conversion, analysis, and filtering. Usually they provide a specialized feature such as the ability to processes large (measured in gigabytes) images on RAM restricted machines with support for parallel algorithm processing [VIPS 2007].

4.1.2 General Image Editing Toolkits

A general image editing toolkit is software that provides for a large number of image manipulation routines for purposes such as editing and composing images. They can provide the ability to read a vast variety of file formats as well as drawing primitives such as circles, lines, and boxes. Utility features include resizing, rotating, cropping, and some conditioning filters that include blurring or sharpening. These toolkits do not provide dataflow, but are specialized to the image editing field. Such toolkits are useful for integration into a computer vision system as opposed to providing for such systems. In addition, they are more useful for altering the visual information that passes through a vision system rather than interpreting it

as a general image processing toolkit. A few examples of general image editing toolkits include IM [2007], ImageLib [2007], ImageMagick [2007], DevIL [2007], and VIAGRA [2007].

4.1.3 Multimedia Toolkits

Multimedia toolkits are software packages oriented toward the entertainment aspect of application development such as adding sound or video to applications for playing music or developing video conferencing applications, respectively. They include timing features to ensure correct playback of audio or video streams. The filtering that is performed by these systems could include computer vision analysis, but filtering in such systems is intended for visual effects or conditioning images rather than analysis. There is also a class of applications used for Video DJing that are mainly for entertainment and performance of which a few have underlying systems with attributes similar to a multimedia toolkit.

Multimedia toolkits also tend to provide components for image capture, dataflow and display. The modular design and timing constraints present in multimedia applications is also relevant to image processing. Features provided by multimedia toolkits can be used to benefit computer vision system kits. Some examples of Multimedia Toolkits include GROUPKIT [Roseman and Greenburd 1992], GStreamer [2007a], and DirectX [2007]. Two video DJing applications include Veejay [2007] and LiVES [2007].

4.1.4 Scientific Numerical Processing Environments

Scientific numerical processing environments are usually provided in the form of an Integrated Development Environment (IDE). Some of them provide libraries that can be linked to without the need for invoking the IDE. They usually provide vast function sets for scientific analysis of image data, but are not limited to vision processing as they are intended to analyze almost any type of data. These applications are not targeted at providing a computer vision system SDK meant to execute in a live environment. Some examples of applications in this category are MATLAB [2007], SciLab [2007], and Octave [2007].

4.1.5 Segmentation

Segmentation is the analysis and deconstruction of an image into regions. These regions crop the visual field to a specific area that could contain an individual feature or object. An example of this is the color segmentation that groups color regions together to provide a more coarse representation for the scene. These segmented regions can be passed into a shape analysis or interpreted as part of object detection routines. Generally, the software found for segmenting does not provide a modular system that can be cleanly incorporated into an existing application. Segmentation is usually provided as a complete application that would require work to separate and understand the components of the full system. Such systems are best suited for adaptation into a program that uses a computer vision system SDK as the backbone of the processing. The modules or filters created from such integration could benefit others attempting to create reliable segmentation software. Examples of segmentation applications include CMVision [2007], as well as Felzenszwalb and Huttenlocher [2004].

4.1.6 Continuous Media Processing

Some continuous media processing software packages include MFSM/SAI [François 2000; François 2007], VuSystem [Lindblad et al. 1994, pgs. 307-314; Lindblad and Tennenhouse 1996, pgs. 1298-1313], CVTK [2007], S2iLib [2007] Myron (WebcamXtra) [2007], CVTool [2007], and Integrating Vision Toolkit (IVT) [2007]. Next to general data processing and multimedia toolkits, continuous media processing software kits are the closest set of development packages that attempt to provide all of the outlined system requirements (see Chapter 3). Modular design is a common theme where sources, filters, and sinks are connected to create the desired custom vision system.

Dataflow is not provided by all of the software kits in this category but it is present in a few. Most software in this category tends to be restrictive in their extensibility, supporting only a particular operating platform, a single programming language or a particular development environment. These restrictions make their scope of application either desktops, compiled programming languages or a proprietary development environment, respectively. Concurrency is not a standard feature among such software either. Design features of these

systems should not be ignored as they are usually well tested systems that are simply restrictive.

4.2 Vision Software Packages

4.2.1 OpenCV

OpenCV is a portable computer vision package that includes four different components for real time computer vision. It is being developed by a team of world-wide developers [OpenCV 2007a]. The first component is CxCore which supplies many generic types for the rest of the libraries which include structures for trees, lists, queues, sequences, images, and the functions that operate on them. The second component is CvReference which includes all of the processing and analysis functions. The third component is the CvAux library which contains experimental or obsolete components of the OpenCV project. The final component is the HighGUI library which supplies support for displaying, capturing, and saving images.

The HighGUI component is provided as an additional library to allow for creation of “experimental setups” [OpenCV 2007b]. HighGUI is the closest that OpenCV comes to providing for the system components of a computer vision system software development kit. It provides for an extensive list of capture interfaces and has a portable display. It does not aim to be a computer vision system SDK, but rather a quick means to implement a computer vision application. There is no provision for threading or dataflow; the image acquisition, processing and display must be done explicitly.

OpenCV is a great resource of standardized implementations of computer vision functions. It would be best integrated into a computer vision SDK to supply the real-time filter and analysis functions. It is possible that a fifth library specifically meant for computer vision systems could be added to the OpenCV collection. The OpenCV Canny edge detection algorithm is included as a filter in Phission (see section 5.6) to show the possibility of integration with a computer vision SDK.

4.2.2 Modular Middleware Flow Scheduling Framework

The Modular Flow Scheduling Framework (MFSM) [François 2007a] is meant for "immersive, interactive applications." MFSM implements the Software Architecture for Immersipresence (SAI) which is a term coined by the designer that is derived from the application field. This architecture is a model for "parallel processing of generic data streams." The architectural components of SAI are very straightforward and are made up of sources, cells, streams and pulses. Sources generate data and cells create pulses of data sent along active streams. Active streams connect cells or "processing centers" together, where each cell adds data to the pulse structure and passes it along to downstream cells. Passive streams exist to connect sources to cells and to retain any persistent data within the application. Active streams are meant for volatile data.

This work is the outcome of analysis into processing systems for multimedia and related applications [François and Medioni 2000, pgs. 371-374; 2001, pgs. 35-49][4]. MFSM is possibly the most relevant current work relating to vision system processing software development kits (as outlined by the system requirements in Chapter 3). The SAI design supports most of the system requirements and MFSM implements those components.

The software is organized as different "modules" which are separately packaged. They build on the MFSM package and add specific functionality such as image representation and processing. It has integrated third party libraries for standard processing algorithms within cells that are then used to construct the vision system. OpenCV [2007a], Unicap [2007] and OpenGL [2007] are used to provide for portable image representation, capture and displays, respectively.

4.2.3 VuSystem

VuSystem is a computer vision system for "compute-intensive multimedia" applications. The design of VuSystem consists of in-band and out-of-band contexts that are the vision system and the main application areas, respectively. The vision system is separated into source,

4 MFSM was found during the later stages of research and provides direction for continuing the research into computer vision system SDKs.

filter, and sink modules. Each module connects to another using a data protocol that transfers data without buffering by using a “ready/not-ready” protocol. If the upstream module is producing data faster than the downstream can consume it, this protocol causes the upstream module to wait with old data until the downstream module can receive it.

The papers that describe the VuSystem [Lindblad et al. 1994, pgs. 307-314; Lindblad and Tennenhouse 1996, pgs. 1298-1313] were the basis for the system requirements during early stages of the research for this thesis. It is a very basic design that employs the common components (source, sink, and filter) and links them together for automatic dataflow. Separating the system into the in-band and out-of-band regions is a logical design that allows the vision system to be a single purpose module as is desired by the system requirements.[5]

The VuSystem uses the X Window System Toolkit (Xt) for scheduling in an unthreaded single process application. Depending on the Xt library inherently ties VuSystem to the platforms that Xt supports. In addition, the holding of old data in upstream modules allows the actual processed data output to be latent compared with the input time of the data. Systems can usually acquire data faster than it can be processed.

Threading the vision system will allow a recently acquired image to be processed as opposed to an older frame that was held in the upstream module. This requirement is important for mobile robot applications to prevent the robot from having an out of date world model. The targeted application scope is that of “computer-participative” multimedia, which is not too far from the target applications for robotic deployment. The system still lacks many of the desired features that will make the best use of resources on newer computing platforms such as multiple cores. Finally, it has not been updated since the late 1990's (according to the distribution directory [VuSystem 2007]), making it out of date with current interfaces for capture and display.

5 Many other research initiatives, such as MFSM/SAI, reference the VuSystem as a related work.

4.2.4 Integrating Vision Toolkit

The Integrating Vision Toolkit (IVT) [IVT 2007] is a software package that includes classes for capturing, processing, displaying, synchronization, threading and processing algorithms. It has a generic interface for each type of component that allows use of varying interfaces on Windows and Linux with a common interface. The internal image formats are RGB and grayscale. They include methods to convert between the OpenCV image type and their own internal image type. Much of the GUI support uses the Qt [2007] toolkit which is a cross platform toolkit.

IVT is another example of a software application which came into existence more recently. It is a promising step in the right direction, but it lacks the constructs for automatic dataflow or built in thread-safety. These requirements could easily be added by the development team working on IVT. The IVT developers took the approach of using several portable toolkits for capture and display support. In addition, they only provide support for a makefile or a Visual Studio project file and do not include build support that uses the Autotools [Vaughn et al. 2007] which are the standard for cross platform *nix[6] development. IVT includes platform specific types in the interface or header files which may cause issues for language modules generated using SWIG.

4.2.5 MIMAS

Mimas [2007] is a C++ toolkit for developing real-time vision applications that is mainly for Linux and uses a variety of development tools that are standard to many professional projects. It includes several capture and image reading support classes as well as display or output classes. Mimas is currently limited to APIs available for Linux and the APIs themselves are not portable. It makes use of templates for representing the image types, which is standard practice in many scientific toolkits.

6 *nix is used to denote any operating system “UNIX-like”[Wikipedia 2007d]. Two examples include Solaris and Linux. It is used here to be any UNIX style environment including Cygwin [2007]. It does not include the Windows operating system.

Mimas is another example of a toolkit which has sought to supply many of the requirements for a computer vision system SDK but still lacks a few of the necessary features. It does not build threads into the system which reduces its effectiveness (by requiring the developer to manually code in the concurrency support), and there is no support for automatic dataflow. These features would have to be developed on a per project basis rather than being an inherent feature of the software kit. However, Mimas has a significant amount of development effort behind its algorithms and applications. These features could easily be adapted into a computer vision system SDK.

4.2.6 DirectX

DirectX [2007] is a multimedia framework made up of components for user command input, processing, capturing, displaying and anything else media related. It is the main interface for developing games on Windows platforms, and there is hardware acceleration for DirectX components as well. The component of interest for this document is the DirectShow API. It is the current API for acquiring, processing and displaying of video or audio. DirectShow features a system by which one creates a filter graph by connecting sources, filters and rendering components. The data are moved through the system automatically during runtime and supports a push or pull protocol between filters. The system uses threads internally to execute the system in an efficient manner.

The basic design includes the ideas that are present in many other related software packages. This software is an example of a closed source software package that is proprietary software. It is developed by Microsoft and is inherently tied to the Windows operating system. Porting it to other platforms is not a feasible approach for a developer who simply intends to create a vision system. Extending to arbitrary data processing may be possible, but it is not intended for this purpose. Rather, a system could be built to use DirectX components for supporting the Windows platform because DirectX itself is not built to be an efficient computer vision system SDK.

4.2.7 LibVideoGfx

LibVideoGfx [2007] is a low level video processing C++ toolkit for Linux with support for various compressed file inputs and Video4Linux support. It supports displaying images using X11 with support for the Xv extension. The common images formats of RGB, HSV and YUV are available. The library uses templates extensively for the representation of the image type. LibVideoGfx is supported in Cygwin for file reading, image displaying and processing. There are MMX optimizations for some operations within the code such as copying data from V4L into an image buffer. LibVideoGfx has been used as a research tool for real-time segmentation of sports events. Finally, the software is packaged with autotools and makefile support for detecting features, compiling and installing.

LibVideoGfx lacks any concurrency or any thread-safe structures. The interfaces to the X11/Xlib are exposed through the header files when they could be more thoroughly hidden within the source files. The extensive use of templates may be confusing to novice C++ programmers and also may cause SWIG to not correctly wrap the library into other languages. In addition, there is no dataflow provided by library. Dataflow should not be expected because it is meant for processing rather than vision system construction.

4.2.8 GStreamer

GStreamer [2007a] "is a framework for creating media applications."[GStreamer 2007e, Chapter 1.1] They draw some of their inspiration from the DirectShow design where components plug into each other and direct the flow of information. They have designed the framework so that it can be extended to any type of data, in addition to the normal audio and video support. It is built on the GTK+ and GDK libraries and conforms to the Glib 2.0 object model [GStreamer 2007b, sec. 2.2.2]. The system provides a unified application programming interface for input plug-ins that supply audio and video from various sources (e.g., Video4Linux, AVI decoders, MPEG decoders). Building on Glib allows them to take advantage of a significant amount of previous work. Two examples are a fast memory allocations and reference counting mechanisms that reduce allocation time and remove memory copying, respectively.

A media application is created by linking GStreamer elements together and adding them to a pipeline. GStreamer uses the terms source and sink (as do many other related works) to refer to an element that sources data or is the destination of data, respectively. Connections are made between sources and sinks through pads (which are synonymous with plugs or ports in other frameworks) and data flows from a source pad to a sink pad. A filter is used to refer to an element with both source and sink pads which are capable of processing the media or performing some other action such as multiplexing several data streams into one.

GStreamer is modular in design, provides mechanisms for dataflow, and has plug-ins for capturing (or acquiring) data as well as plug-ins for displaying data. Internally, GStreamer makes use of threads to allow the collecting of several elements into a pipeline that runs in its own thread. The software package is portable, extensible and packaged very well. It consists of the main framework with easily installable extension packages. It does depend on several third party packages that may not be supported on every system but it is able to compile only the extension components that are supported. GStreamer is a very well designed framework that takes care to make sure it is runtime efficient as well as easy to extend.

GStreamer is supported on *nix type systems, OS X and Windows [2007c, Chapter 23]. There are a decent number of third party applications which use GStreamer [2007d]. A processing system that is useful for robotics could be written with GStreamer as the supporting framework library. Such a system is possible because a filter element can allow any type of input and any type of output on a pad which could include locations of segmented objects. A vision processing system that uses GStreamer as the framework (or simply used for acquisition and display) has full access to all of the input and output plug-ins available for GStreamer.

4.3 System Design Documents

4.3.1 A Scalable Approach to Continuous-Media Processing

Manolescu and Nahrstedt [1998, pgs. 84-91] present a discussion into the design of the components associated with a system for continuous-media processing. The components

make up a “media-flow architecture” with components that include sources, filters, and sinks. These are connected to each other to pass payloads from the source to the filter and ending at the sink. Protocols for passing payloads include push, pull and indirect methods.

Push methods are initiated by an upstream or source component and only complete when the downstream component has accepted the data. The Pull method is the opposite where a downstream module initiates data movement. The indirect method is performed by providing a shared memory resource which any downstream component can access and copy. The indirect method may have a signaling mechanism to let downstream components know when new data are available.

Finally, the paper details the two different partitions of an application that uses the media system processing system: control partition and processing partition. The control partition is the area of the program that affects the flow of the system such as the main application code or the thread scheduler. The processing partition is the system itself and the code governing the generation or movement of data.

This paper presents the basic design of a generic data processing system for media. It accomplishes some of the objectives of other research such as MFSM and the VuSystem. There is no software product, that could be found, provided by this paper, and its intent was directed towards implementation of compression algorithms such as MPEG. Even though it is not directed at robotics or computer vision, the patterns presented by the paper are useful in directing the design of such a system. The three protocols for data transfer and manner in which the payloads are passed are crucial observation that a computer vision system software development kit can use. For example, MFSM uses a payload called a Pulse and passes data using a push method. The VuSystem permits both the push and pull protocols for transfer of data between its systems components.

4.3.2 DAVE

DAVE [Mines et al. 1994, pgs. 59-66] stands for Distributed Audio Video Environment and is meant for developing distributed multimedia applications. It supplies a “plug and play”

programming interface to audio and video components. This interface allows remote devices to be accessed as if they were not remote. It is intended for video conferencing, media archival, remote process control, and distributed learning.

The application domain of DAVE is not the important aspect but rather its design goals. Most of the design goals are common to the system requirements of a computer vision system software development kit. DAVE provides for device and media extensibility. The media extensibility is accomplished using a message hierarchy from which any data type or message can be derived. The device extensibility is provided using object-oriented techniques and a high level of abstraction. The resulting plug and play modular components allow easy development by application developers without needing low level device knowledge. A common theme of connecting source to filter to sink is done with DAVE as well.

DAVE does not provide for threading but does provide the ability to spawn processes. In addition, its goal of providing a distributed environment provides the feature of concurrency. DAVE does not provide for operating systems other than UNIX variants. However, the research is still relevant as it shows the existence of many of the system requirements in a single package. It lists these requirements for similar reasons.

4.3.3 AIBO Vision Workshop 2

AIBO Vision Workshop 2 (AVW2) [Lovell 2004, pgs. 268-274] is an environment in which to develop vision processing systems. It provides for offline development within the AVW2 environment to profile and evaluate the system. It allows the creation of a custom vision pipeline using either provided filter components or user written components. Its inspiration is the DirectShow method of creating a vision pipeline, a design that appears in many related works mentioned here. Data are passed from source to sink using a method similar to MFSM where source data are read-only, new processed data are added to that and then passed along to downstream modules. This design allows for data fusion that the first version of AVW did not provide.

The internal format for the AVW2 processing is YUV2 and requires that any other formats be converted to this format. Providing a single internal format make things simple but is restrictive when it comes to the advantages of color matching in other formats. The environment is not provided as a software development kit; instead it is an integrated development environment that supports outputting binary code suitable for AIBO systems. Development is restricted to using the AVW2 environment which restricts the extensibility of the system to different environments.

The representations for the vision processing system and the dataflow features are similar to that of other systems and provide multiple pieces of the system requirements. However, because it is restricted to a development environment and only supported on AIBO systems, it has not been shown to be extensible. It does not provide features to allow it to be integrated with other languages either. Nevertheless, it attempts to provide similar goals (described by this document) to provide a solution to create vision systems easily, but it does so in a restrictive manner.

4.4 Conclusions

The related work and related groups of work discussed show that the system requirements exist and are valid design features. The system requirements are rarely present as a whole group within a single software package. Our discussion of the relevant software outlines why a missing requirement has negative effect on efficiency for a developer. By identifying all of the system requirements as necessary and showing how a missing requirement causes development difficulty or restrictions, an efficient computer vision system software development kit can leverage the research of others to make sure that it provides for development efficiency.

5 Phission

Phission was developed to satisfy the system requirements of a vision system software development kit, outlined in Chapter 3. It provides image data representation and the ability to capture, process, and display live or recorded image data. These capabilities are built on a portable lower layer that includes threading, synchronization, and data movement. Phission employs a number of design methods that were outlined from the beginning and others which were added during development (see section 5.10). The project has been under development for several years during which it has been used in computer vision applications and lab research (see Chapter 2). This chapter describes how the system requirements were satisfied through a description of the project in terms of layering, briefly detailing the components of each layer, outlining the design methods, and discussing the history of Phission.

5.1 Overview and Purpose

Phission is a concurrent, cross platform and multiple language vision processing system software development kit written in C/C++, targeting applications on the desktop as well as robot platforms. The project employs a set of design methods that are not exclusive or unique to Phission, but have driven the foundation of the minimal system requirements for a vision processing SDK. Phission is used for constructing software based vision processing subsystems meant to run concurrently with a main control loop that manages said subsystems, built with components from the Framework layer (see section 5.5). It contains image representation data structures and several image utility functions (see section 5.7).

Phission currently supports development for Windows, Linux and Blackfin/VisualDSP++ 4.5 systems with hooks provided for Java and Python languages. It is also a well developed and debugged software package with good documentation and plenty of example programs to demonstrate its use. It is packaged as a single software development kit with target applications mainly in mobile robotics, real or simulated. Phission is also fully capable of non-robot desktop operation, which is essential because developing on a live robot can introduce negative development issues such as slow compilation times, network or serial connection communication limits, or incomplete feedback of visual information necessary for debugging.

The main benefit of the desktop environment is its wide availability compared to robot platforms; desktop environments allow initial vision research to be prototyped quickly without the setbacks that developing on a robot can introduce. Use of commodity video acquisition devices such as webcams or television capture cards combined with the Phission library can start the research into image processing algorithms with little investment in time or money.

The Phission SDK is best described as using a layered design. Each layer is built using lower layers (see Figure 11). Lower layers do not reference any upper layer classes or functions. These layers include (from top to bottom) the Application layer, the Framework layer, the Filter layer, the Image Toolkit layer, the Dataflow Toolkit layer, and the System Toolkit layer.

- At the top, the **Application layer** includes classes which can encapsulate the framework classes into an even easier to use class.
- The **Framework layer** includes classes for capture, filtering or processing, processing pipeline, display and overall system management.
- The **Filter layer** adds a filter class that provides an abstract interface to execute any derived filter. This layer also includes all of the specific filter implementations such as blur and segmentation filters.
- The **Image Toolkit layer** is made of two parts. One part implements the basic image types and provides all of the image functions. The other part encapsulates these into a single image class that inherits from a data class defined in the data toolkit layer.
- The **Dataflow Toolkit layer** is a small layer which defines classes that provide mechanisms for thread-safe data synchronization, signaling and transfer between threads.
- At the bottom, the **System Toolkit layer** provides the portable interfaces to threads, synchronization classes, time functions, standard integer types, and other code utility features.

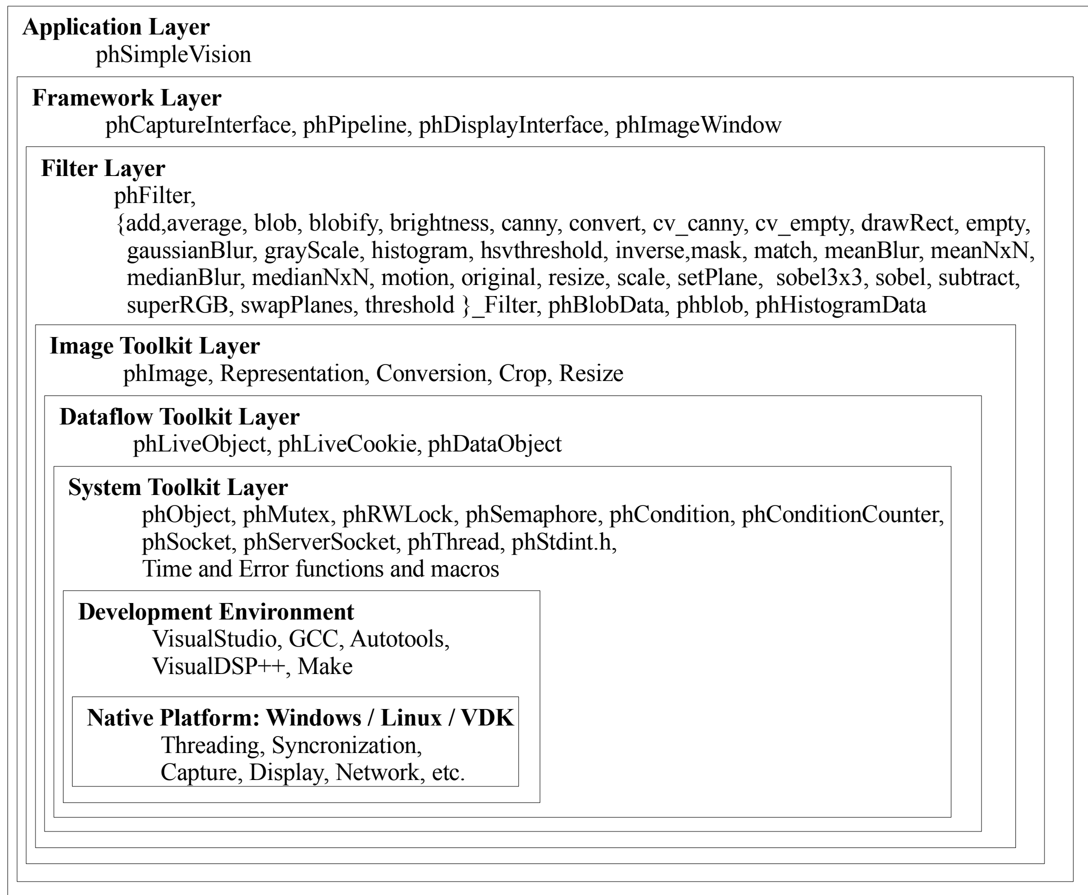


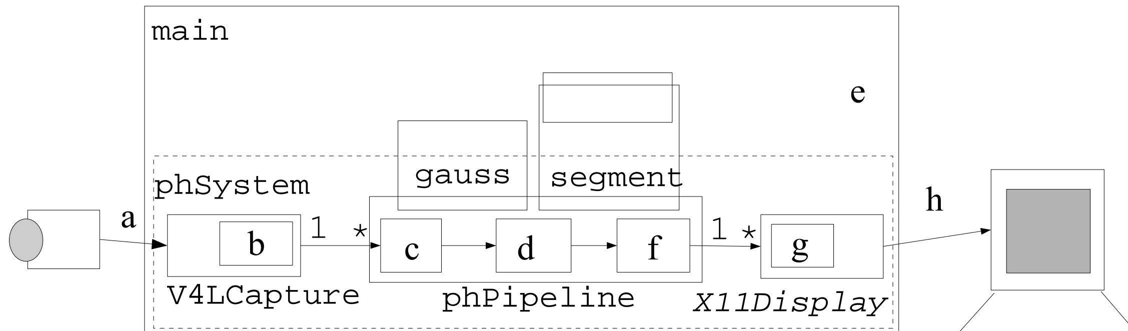
Figure 11: Phission layer diagram which shows how specific layers can only access themselves and the layers on which they are built.

5.2 Example of Use

The design of Phission can be best shown through an example. Let us assume there is a robot which must find the red cones. The red cones could be surrounded by many other objects but the assumption is there are no other object with the same exact color. An input frame can be analyzed to obtain a histogram to create a color and threshold pair that identifies only the red cones.

The practical purpose of this exercise could be navigation through an obstacle course where the robot must remain between the cones. Segmenting colored objects from a capture device is a common use for vision systems in our lab, and Phission has been used for similar applications for varying purposes (see Chapter 2). A system that segments a colored object

needs four essential components: capture, preprocessing, segmentation processing, and display.



A capture device provides an input stream of images to a pipeline object. The pipeline executes the filters (in the order they were added) with the captured image data as input. The two filters[7] usually needed are a preprocessing filter (e.g., Gaussian, mean or median blur filters) to reduce noise and a segmentation filter for segmenting the objects from the image. The noise reduction is usually necessary only if the input device (e.g., CMOS sensor) causes a lot of noise, but it can also be used for eliminating small scale texture such as a speckled rug. The segmentation filter takes a color value and threshold, looks for pixels that match, then groups those matching pixels into regions, and lastly provides a means to retrieve the list of regions. Finally, an output display allows inspection of the output from the filters that ran in the pipeline. A filter is allowed to alter the images in the pipeline for visual debugging. In this case, the segmentation filter will color in and draw a box around the matched regions.

5.3 System Design

We will assume that we are developing this application on a Linux platform. `V4LCapture` provides image acquisition for the Linux platform, `X11Display` provides a native display for Linux, `gaussianBlur_Filter` will be used for noise reduction and `blob_Filter`[8]

---

7 A **filter** is generally used to denote a component that conditions or adjusts the image. A **processor** is a component that analyzes the image data and either converts it or provides output data resulting from the analysis, such as a segmented region list. This thesis document uses the term **filter** to denote a component for conditioning or analysis. It is clarified by the name of the filter but future versions of Phission will seek to differentiate the two terms.

8 The word **blob** refers to a segmented region in an image. When color-filling the matched region, it resembles a blob of ink or goop.

will segment the image given the color and threshold values. The `phPipeline` and `phSystem` classes are containers for instances of the objects above and do not vary across platforms or applications. The `V4LCapture` class is swappable with the `VFWSource` class and `X11Display` can be swapped with `GDIDisplay` when the application is to run on a Windows platform. The `gaussianBlur_Filter` could optionally be replaced with a `meanNxN_Filter` or `medianNxN_Filter` for noise reduction.

All of the Phission vision applications and examples have the same basic program format. A code example can best show this layout and how simple it is to code with Phission (see Appendix 1 for the source and related files). The beginning of the file contains the include statements and beginning of the main function:

```
#include <SegmentationExample.h>
#include <phission.h>

int main( int argc, char *argv[] )
{
    /* Declare the function and rc variables:
     *    #define phFUNCTION(name) \
     *            const char *function = name; \
     *            int rc = 0; \
     * phFUNCTION is simply a convenience macro.
     * The "function" variable is required for other
     * Phission macros that print.
     */
    phFUNCTION("main")
```

The first step is to define all of the Phission objects for the vision system:

```
    phImageCapture      *capture        = NULL;
    phPipeline          *pipeline       = NULL;
    blob_Filter         *segment        = NULL;
    gaussian3x3_Filter  *gauss          = NULL;
    phSystem            *vision_system  = NULL;
    phDisplayInterface  *disp_pipeline  = NULL;
    phDisplayInterface  *disp_capture   = NULL;
```

There are a few other types and classes needed. The phColor type is used for setting the color value, match threshold[9], and the output color of the blob_Filter markup. The phBlobData class is used to retrieve data from the segmentation filter:

```
phColor object_color = phColorRGB24_new(230,50,55);
phColor threshold    = phColorRGBA32_new(55,25,35,0);
phColor output_color = phColorRGB24_new(0,0,0);
phBlobData blob_data;
const int32_t acceptable_min_size = 100;
```

We must now allocate the objects that we declared above:

```
capture         = new V4LCapture();
pipeline        = new phPipeline();
segment         = new blob_Filter();
gauss           = new gaussian3x3_Filter();
disp_pipeline   = new X11Display(320,240,"Pipeline");
disp_capture    = new X11Display(320,240,"Capture");
vision_system   = new phSystem();
```

Next, one must set the initial parameters of the vision system objects:

```
capture->set(320,240,"/dev/video0");
capture->setChannel(0);

segment->setColor(object_color,threshold);
segment->setOutcolor(output_color);
segment->setDrawRects(1);
segment->setColorBlobs(1);
segment->setColorMinSize(acceptable_min_size);
```

The filters must be added to a pipeline to process the image data:

```
pipeline->add(gauss);
pipeline->add(segment);
```

Each capture, pipeline and display inherits from a thread class (see section 5.9.8). Managing of these threads is performed by the system class (see section 5.5.1). Here we add them to the system object:

9 We will assume that the color we are looking for is a red color. The color and threshold values were calibrated by hand, a simple process which will not be detailed here. A more consistent and reliable means of obtaining color and threshold values is to use the histogram_Filter, align an object in a given rectangular region and retrieve the peak threshold values using the phHistogramData object.

```
vision_system->add(capture);
vision_system->add(pipeline);
vision_system->add(disp_capture);
vision_system->add(disp_pipeline);
```

The Phission objects need to be connected together to direct the flow of image data:

```
pipeline->setInput(capture->getOutput());
disp_pipeline->setInput(pipeline->getOutput());
disp_capture->setInput(capture->getOutput());
```

We must also connect a `phBlobData` object to the output from the `blob_Filter`, an object of the same class[10]:

```
blob_data.connect(segment->getLiveBlobOutput());
```

Finally, we can start the system up and proceed to our main loop:

```
vision_system->startup();
```

The main loop will wait until the display is closed before it stops. It will use the `phLiveObject::update` method that `phBlobData` inherited in order to continually copy out segmented data in a thread-safe manner. The loop will then print the data[11]:

```
while (vision_system->displaysOpen() > 0) {
    rc = blob_data.update();
    if ((rc == phLiveObjectUPDATED) &&
        (blob_data.getTotalBlobs(blob_min_size))) {
         blob_data.print_data(blob_min_size);
    }
}
```

When the loop ends the system should be shutdown to stop all of the threads:

```
vision_system->shutdown();
```

Finally, delete all of the objects to make sure memory is freed properly and return from `main`:

10 A `phBlobData` object is part of the `blob_Filter` so that we can retrieve the segmentation data in a thread-safe manner during runtime:

11 Minimum size is used to denote the smallest acceptable size for a segmented region. It is used to eliminate noisy data such as a region that is only a few pixels in size.

```
    phDelete(disp_pipeline);
    phDelete(disp_capture);
    phDelete(gauss);
    phDelete(median);
    phDelete(mean);
    phDelete(segment);
    phDelete(pipeline);
    phDelete(capture);
    phDelete(vision_system);

    return phSUCCESS;
}
```

5.3.1 Segmentation Results

Figure 12: Segmentation input frame collage of a scene with the red cones.

The program outlined in the previous section has approximately 55 lines of code. There are thousands of lines of code in lower layers that are unseen from the application level code. The capture, filter, and display classes can be swapped with other classes of the same type to construct different systems than the example shown. The segmentation code was run on a sample input movie (a collage of frames is shown in Figure 12) and the resulting output found two red cones (seen in the left two frames of Figure 13 marked up with black).

Figure 13: Segmentation output frame collage of a scene highlighting the red cones.

5.4 Application Layer

The previous section shows a program that can segment and track red cones. The code from the previous section could be encapsulated in a class of its own. Such a class would likely provide the methods to start the system, get new blob information thread-safely, and stop the system. The set up, calibration, framework construction, and allocation code would be handled internal to the class. The Application layer allows Phission to provide a ready to use vision system without requiring a developer to design the system.

The current Application layer is made of a single class called the `phSimpleVision` class. `phSimpleVision` is meant to be a quick way to implement only the most basic vision system design consisting of a single capture class, multiple pipelines, and multiple displays. An entire vision system is contained within this one class and requires significantly less code than applications that use the Framework layer components to create a custom vision system.

`phSimpleVision` allows any filter to be run and manages the connections between the capture, filter pipelines and displays. Native displays and network displays are supported and can be disabled using a single method call. The abilities to train on the color of an object and track that object are also managed by this class. Should one need to take control of the individual pieces of the simple vision system provides, a reference to each component can be retrieved and the system adjusted.

The Application layer exists as another layer to abstract the lower layers into successively simpler classes. This layer is expected to grow as more Phission applications are written and developers begin to contribute code. Application layer code focuses mainly on purpose rather than implementation. The difference between purpose and implementation can be seen from the previous section. The concern is with finding the red cone and the color that identifies the red cone. The concern is not centered on the underlying implementation of the system. The concern is more about what to do with the system. The system can be assembled quickly from Framework layer components to find the red cone.

5.5 Framework Layer

The Framework layer is the collection of software classes which provide a common architecture for implementing a vision processing system. These classes include capture, display, filter pipeline, and system management. Using the framework classes allow a developer to code a vision system that is capable of processing in a live environment with relative ease. A basic system acquires, processes, displays data, and outputs processed information for code that controls a robot or application (see Figure 14 for a diagram of a basic system). The capture class provides input to the system. The filter pipeline consumes data and executes filters on the input data. The display consumes data and displays it to the screen, a network or some other output. The system management class (see section 5.5.1) manages the threads that are built into the aforementioned classes.

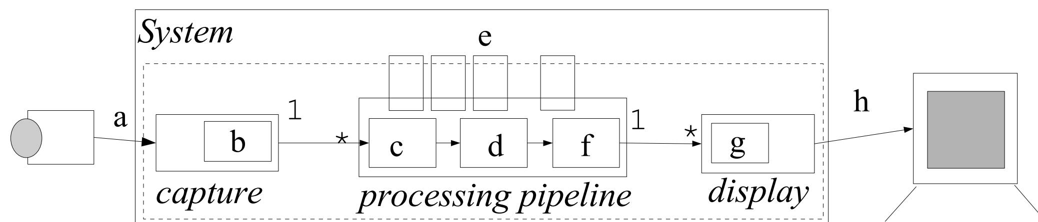


Figure 14: Outline showing the structure of a Phission built vision system. An image is captured (a), stored in a thread-safe buffer (b), copied from the capture object's buffer to a pipeline (or display) object's input buffer (c), and stored in a workspace image (d) which is processed by filters (e). Finally, the image is swapped into an output buffer (f), copied into a display object's private buffer (g) and written to the screen (h).

It is crucial for a vision system to perform in the real world and not simply a sandbox environment using recorded data. Phission processes live data using thread-safe classes and encapsulating threads within each capture, processing pipeline and display class. Each thread performs the specific task associated with the class (e.g., to capture, process, or display) and takes advantage of any possible concurrency available in the target system.

The concurrency is possible by taking advantage of DMA, a hyper-threaded CPU, or multiple cores which are managed by the operating system. The capture and display threads

can make use of DMA and free the CPU for processing. Threads rely on the scheduler to provide concurrency by giving each thread a time-slice or by queuing them on multiple CPU cores. The threads are spawned so that the vision system runs concurrently with a control thread. The current architecture allows for the control thread to poll for processed data output from the vision system such as segmented objects. When there are data available, the control thread can use the processed results. Otherwise, the control thread can continue other higher priority tasks.

The generic term given to the capture module is source. The processing pipeline executes filters and is a combination source/sink. A display is called a sink. Data are created at a source and a sink consumes that data. Through connecting various filters or pipelines to true sources and sinks allows one to create an arbitrary system.

The live data processing framework used by Phission uses an indirect protocol [Manolescu and Nahrstedt 1998, pgs. 84-91] for moving data through the framework (from a source to a sink). The indirect method puts data in a shared repository and sends a signal when there are new data available. The protocol is implemented by the `phLiveObject` class using a three method client interface (see section 5.8.1.2) and employs a reader/writer lock (see section 5.9.3) to allow multiple clients access at the same time.

Phission encapsulates source data in a class (e.g., `phImage`) that inherits from the `phLiveObject`. The source thread continually copies new data into its own internal instance. Any client threads that are listening get signaled by the source thread when the copy is complete. A copy of the new data is made to the client's private instance. Data are output from the vision system through filters running in a pipeline that contain a data class (which uses the indirect method). A control thread can poll the filter output data object and retrieve processed information in a thread-safe manner. The polling of filter data was shown by the example in sections 5.2 and 5.3. The segmentation information was retrieved using a data class that connected to another instance of the same class type contained by the segmentation filter.

The indirect method allows a sink to acquire data as fast as it can and without preventing other consumers from acquiring a copy of the same data. If the processing time of a sink is faster than the source rate, the sink will always find new data when it finishes. When new data are not available, the sink will simply wait until there is new data. Since the target platform has a limited amount of CPU time and data transfer capability, a sink may miss processing an input image. That source data may be overwritten in the shared repository before some threads get the chance to make a copy.

Overwriting old data is not a problem because the framework is meant to process data in a live environment. A video production system meant to produce movies may need to process every frame. However, a vision system processing in a live environment is really only concerned with the most recent frame or instance of data. Buffering every frame will result in an infinitely growing buffer when each frame requires more time to process than the time between frames. This buffering can quickly lead to consumption of system memory. The control thread will end up retrieving increasingly older processed data (from the filters) which in turn leads to an inaccurate and out of date world model. Since Phission was designed mostly with robotic applications in mind, the processing of data has kept to the asynchronous indirect protocol.

5.5.1 `phSystem`

The threads of the framework created using the capture, pipeline and display classes are managed using the `phSystem` system management class. It maintains a list of threads given to it and starts them in the correct order when requested. It was written when it became apparent that the startup of the threads was sufficiently complex enough to encapsulate it into a class. Prior to the existence of this class, the developer had to maintain a list of threaded objects and starting the threads manually. The `phSystem` class greatly simplified the requirements on the developer for constructing and managing the vision system as a subsystem of the control thread. The system does not need to include at least one capture, filter pipeline or display. It can simply be a filter pipeline with generated data input or a display that displays generated image data.

The thread starting order is to start displays first, pipelines second, captures third, and finally any user threads. As you may notice, this order is backwards from how one would normally think to start the threads. When a capture class is started up, it will immediately begin capturing data and making it available to any downstream thread. However, this immediate action begins consume CPU cycles and delays the start of the downstream threads. As soon as a pipeline would start, after the capture, there would now be two types of threads occupying CPU time and delaying the start of the display. The design of each capture, pipeline and display thread is to block while waiting for input data, user intervention, or program intervention.

If there is no data available from a capture or pipeline class, then the display will cease looping (except for user events) and release the CPU. When the system class goes to start the pipeline class, it does not have to share the CPU with the display class. The pipeline class will start up and block while it waits for data to be updated by the capture class. The capture class is then started, and the downstream components of the processing are ready and waiting to begin immediately. Since user threads are not predictable, they are started last to allow the vision system to start.

An application is not limited to a single `phSystem` instance. There can be any number of vision systems being managed. An example for this use is when a program requires two separable vision systems that depend on the state of an application or robot. For example, an application is training on an object which it should learn and then that object needs to be followed through a room.

In this example, these are two distinct tasks that will not be executed at the same time. The filters that train on the object can be placed in one filter pipeline and the segmentation filter which locates that object into another filter pipeline. Three `phSystem` classes would make up the entire structure of the system where the capture class would be in one, the training filter pipeline and display would be in a second, and the segmentation filter and related displays for tracking is in the third. Going from training to tracking is simply a matter of stopping one system and starting the next.

5.5.2 The Capture Class

A capture class acquires image data and supplies it to the application. The input can be a file or data from actual hardware. The capture class thread continuously acquires the images during its runtime. The images are placed into a publicly accessible output object which allows the data to be copied using the indirect method discussed in previous sections. Any number of client connections can be made to the output image.

The capture class interface is defined by the `phCaptureInterface` and `phImageCapture` classes. The design of the `phCaptureInterface` and `phImageCapture` classes is meant to allow any input interfaces to be brought into the Phission system and allow the framework to be plug and play. A developer has creative freedom within the source file to create any third party capture class that is not included with Phission. As long as control and image sourcing are implemented properly within the class, proprietary custom systems can be built and take advantage of the features that Phission offers.

The `phCaptureInterface` class provides methods to set a path that identifies the capture device, open and close a capture device, query the state of capturing, set the public output object pointer, and allow retrieving of that pointer. It is a very small set of methods that all capture devices share and allows any deriving class to be controlled without needing to know specific parameters of the capture device. Developers can create their own capture classes without requiring it to be part of the default Phission library, yet still allow Phission functions that take a `phCaptureInterface` pointer to properly control that capture device. It has the same idea as a kernel driver model.

The `phImageCapture` class inherits from the `phCaptureInterface` to which it adds the image related methods, structures and parameters for creating a class that defines a generic image capture device. The parameters include video capture parameters such as brightness, color, contrast, hue, width, height, and format. Some video capture devices do not support calibration for all of the parameters and so that list represents a joined set of all of

those parameters in order to support the majority of capture interfaces. The `phImageCapture` methods allows for thread-safe adjusting or retrieving of these parameters during runtime.

The specific implementations of capture classes include FFmpeg [2007; Wikipedia 2007m] library supported movie files, Video4Linux [V4L 2007], VideoForWindows [VFW 2007], a network capture class and the TWI/PPI control of OmniVision camera on the Blackfin Handy Board system. FireWire support is currently available on Linux by using a program that exports to a V4L interface [Coriander 2007]. Using a specific API for the target platform requires allocating the appropriately named class such as `V4LCapture` or `VFWSource` and connecting a filter pipeline or display to it. There can be any number of capture classes used within the system and it is only limited by the available resources. Each allocated capture class is passed to a `phSystem` method maintains a list of capture classes. When the `phSystem::startup` method is called, then the capture classes will be started.

5.5.3 The Display Class

A display class is meant mostly for debugging of algorithms or monitoring the system. The display class interface is defined by the `phDisplayInterface` and provides the abstract means to use any specific display without needing to know of it prior to its implementation. When a display class is allocated, it is added to the `phSystem` class. A specific display API is implemented by in a class that inherits from the `phImageWindow` class.

The display interfaces were designed to allow a developer to create a custom display and allow Phission to use that display without integrating it with the library. A proprietary display class can be written (e.g., a class that implements the Real-Time Protocol [Perkins 2003; Wikipedia 2007i; Schulzrinne et al. 2003; Schulzrinne and Casner 2003]) and it can take advantage of Phission features. A display can also be developed and thoroughly debugged before being added to the Phission library. A display must implement the necessary methods, control capabilities, and consume source data for displaying. The display only allows one input for consumption.

During the development of a vision system, the display is likely to always be active during runtime for debugging data visually at any point within the vision system. The normal display permits a user to resize, minimize, maximize, and close the display. There are displays which allow any number of instances to be created (`X11Display`, `GDIDisplay`, `NetDisplay`). Any number of locations within the vision system can be monitored at the same time. Each Phission provided display class attempts to be efficient, but will always require resources from the system that can reduce performance. In the actual deployment of the vision system, the displays can easily be removed from the system for increased processing performance.

The relationship between the `phDisplayInterface` and `phImageWindow` class is quite different from the relationship between the `phCaptureInterface` and `phImageCapture` classes. The `phCaptureInterface` is a parent class of the `phImageCapture` class. The `phDisplayInterface` is a container class for the `phImageWindow` class. The reason for making a container class is due to the nature of data signaling and transfer through the application as well as the design method to require a thread to block whenever possible. Two things that a display needs to be do are wait for new image data and wait for user events at the same time.

Writing a spin loop that polls on user input and for new image data would require one thread but would break the blocking model. A thread can only block on one resource at a time so the simplest model is to break it into two threads. The `phDisplayInterface` thread is a basic thread that uses the client method interface supplied by `phLiveObject` to wait for updates to data and then signal the `phImageWindow` class to display the new data.

The `phImageWindow` derived class makes calls to a specific display API, waits for user events, and displays the most recent image data provided by the `phDisplayInterface`. Since supported display APIs make use of event systems, each image window has a custom event implemented to support a signal from the display interface class which refreshes the

image window. The two different methods allows both the image window thread and display interface threads to block in one place and removes the spin loop.

Even though a display only permits one input, it can be any instance of a phImage class, allowing for great flexibility in design of an application and the use of a Phission display. A developer does not need to use a capture class or a filter pipeline to provide input to a display. The display data can be generated in user code using any type of algorithm. So long as the image data are of a supported image format and is placed into a phImage class using the available methods, the display will see the new data and display it. This requires the display thread to be active by giving it to the system class and starting the system.

The display can also be used to display data which is not acquired from the capture class. One example is the histogram_Filter class and the phHistogramData class. In each loop where data are updated in the phHistogramData class, an image is drawn to visually display the histogram data and it is placed into a phImage object. A display can connect to that image output from the phHistogramData class and work as if it were any other image source, which provides for the automatic redrawing of new histogram data.

There are displays that use the X11, GDI, SDL, and FLTK libraries. X11Display and phX11ImageWindow use X11 which is supported on Linux and in the Cygwin/Windows environment. GDIDisplay and phGDIImageWindow interface with GDI which is a native Win32 API on Windows platforms that is accessible from Cygwin/Windows and through Visual Studio 2005. SDLDisplay, phSDLImageWindow, FLDisplay and phFLImageWindow use the SDL and FLTK APIs, respectively, because both are portable libraries which work in Linux, Windows and Cygwin/Windows development environments. Those libraries are well tested, supported and developed which allow at least one display to work on any new platform during the porting of Phission. NetDisplay and phNetImageWindow create a network display that use Phission socket classes to create a multi-threaded image server and the client counterpart used for network capture is the phNetSource class.

5.5.4 The Pipeline Class

A filter pipeline manages the execution of filters on image data by consuming data from a source image and producing processed outputs. The filter pipeline works similar to a display with the exception that it is both a sink and a source of data. The pipeline will wait for a `phImage` buffer to be updated with data and will copy that data. The data are placed into a workspace `phImage` instance which is both the input and output of every filter. When the final filter is finished executing, the result in the workspace image is placed into the output buffer of the pipeline. A data class is also a possible source of output from the pipeline but is obtained through filter methods, such as the blob segmentation filter.

Filters are added into the pipeline in the order in which they are to be executed. While the Phission SDK was meant to remove the serial execution of the vision system, it maintained the serial execution of the filters. Executing the filters serially was done because it was the quickest means to an end. Development effort for Phission was to be focused on support for target platforms, capture and display facilities, and to provide a thoroughly debugged software package. However, the pipeline workspace image does improve efficiency by removing the need to copy an entire buffer into a filter and out of a filter. Parallel execution of filters can be accomplished by creating more than one filter pipeline. The filter pipeline provides a means for timing execution of the pipeline filters for performance measurements. One can retrieve the time of the last iteration or an averaged time value for a given number of previous iterations.

An example use of the filter pipeline is to segment an image and output image coordinates for a matching segmented region. The `blob_Filter` class takes a color and threshold variable and searches an image for matching pixels to segment out regions of an image into blobs. These blobs have information such as coordinates for the center of mass, which allow a control thread to track an object. The segmentation filter is executed by the pipeline and segmented information is output automatically in the sense that the user does not need to interact with the control of the vision system once the `phSystem` class has started the threads. The control thread can block on the blob data object and wait for new information

output from the pipeline. The control thread can also poll the blob data object and execute other tasks such as navigation and obstacle avoidance until new blob information is available.

5.6 Filter Layer

The Filter layer is built around a filter class that encapsulates a computer vision algorithm into a common interface defined by the `phFilter` parent class. Filters are placed within the processing pipeline class and executed serially within the processing pipeline thread. The Filter layer is a separate layer because a filter should not access any Framework layer components, such as a capture, display or pipeline.

The `phFilter` set of classes does not provide its own thread as it is meant to be executed within the scope of the pipeline. A filter can be executed outside a pipeline but would require implementing a protocol that is done within the run method of the filter pipeline. The filter is where most third party computer vision libraries are likely to be integrated with Phission. OpenCV and CVIPTools are two third party APIs which have had functionality integrated into Phission using the filter class. The Phission `cv_canny_Filter` includes OpenCV code and a car tracking application [Baker and Yanco 2005] includes filters that use CVIPTools.

The `phFilter` class only requires that the `phFilter::cloneFilter`, `phFilter::filter`, `phFilter::phFilter` (constructor) and `phFilter::~phFilter` (destructor) methods be implemented. The `phFilter::cloneFilter` method simply allocates a new instance of itself and sets whatever parameters the filter requires. For example, a mean filter can have a variable kernel size passed as a parameter and the `phFilter::cloneFilter` method would return a class with the same kernel size value set. No other temporal or structural values are copied using the `phFilter::cloneFilter` method. A common method added by a class derived from `phFilter` is a set method (`phFilter::set`) which takes a parameter list as input parameters to the processing algorithm. This method should be made thread-safe to permit adjustments of the algorithm parameters during runtime.

The `phFilter::filter` method is where the actual algorithm is implemented. The member variables for width, height, depth, format and the data buffer are assigned to the values contained in a `phImage` member variable. An image member variable points to the input for the filter which is the workspace image of the filter pipeline. The image data can either be copied and stored or preserved by the filter if necessary as the image member is where the output of the filter goes as well. The derived filter object can output other information, but is limited to doing so through class methods. For example, getting segmented blob data from the `blob_Filter` requires making a call to the `blob_Filter::getOutput` method and having the control loop wait for updates made to the object using the `phLiveObject` three method client interface (see section 5.8.1.2).

The `phFilter::process` method is the public interface for executing the code contained within the `phFilter::filter` method of the child class. This method provides set up and validation of input before executing the `phFilter::filter` method. The output of a filter can be monitored by a display, using a phImage class contained in the `phFilter` class. It is updated immediately after the call to `phFilter::filter` only when a client connection is made to that `phImage` class. A filter can be enabled or disabled through software calls to `phFilter::enable` and `phFilter::disable`, allowing a thread in a pipeline to be enabled or disabled without stopping the processing system. The `phFilter::process` method always checks this variable before performing any operations and returns immediately if the filter is disabled.

There are several classes of filters for processing image data that include preprocessing, temporal, analyzing, and enhancement [Phission.org 2007]. The preprocessing filters are made up of blurring filters that include mean, median, and Gaussian. These filters attempt to condition the image to prepare it for further analysis by reducing random variations within the image. The temporal filters include motion detection, frame addition, frame subtraction, frame averaging, double-difference motion detection, and motion masking. A temporal filter generally preserves old frames or processed information from old data as input for future iterations of the filter. The analysis filters are the histogram and blob which perform color

histogram analysis and color segmentation, respectively. The output from the histogram filter is a color value and threshold which can be adjusted and given as input to the blob filter. The blob filter locates the pixels that match that color and threshold pair. Segmented region information is generated as output from the blob filter which facilitates the basic recognition and tracking of colored objects. Finally, the enhancement filters include Sobel edge, Canny edge and brightness adjustment.

The Canny filter has two incarnations. The first is a custom implementation that allows Phission to always have its own Canny filter without need for a third-party library. The second implementation uses the OpenCV library [OpenCV 2007] and wraps the specifics of the OpenCV API within the filter class. Containing the OpenCV code within a Phission filter assists in wrapping the filter into other languages and shows an example where Phission is extensible to other computer vision libraries. Since the interface to the filter and image data are rather generic (by using simple types: `uint8_t, uint8_t *`), the image data can be packaged into the third party image structures and passed to the relevant processing functions.

5.7 Image Toolkit Layer

The Image Toolkit layer provides the code to represent image data and a few common utility functions. The code within the Image Toolkit layer uses features from the System Toolkit layer including print macros, error checking macros, and the standard integer types. The majority of the image code is written in C. The calls to those C utility functions in addition to the image data variables are encapsulated within an easier to use interface created by the `phImage` class.

5.7.1 `phImage`

The `phImage` inherits the `phLiveObject` interface through the `phDataObject` to provide an Image Toolkit layer data class that allows for the image dataflow. The capture, processing and display classes allocate an instance of the `phImage` and provide it as outputs in the Framework layer (see section 5.5). The `phImage` class encapsulates the image variables and calls to the utility functions to create an easy to use image class. It does not

require the developer write complex code to interact with the image functions as this has already been done by the `phImage` class. Methods allow retrieving any of the image variables and execution of a utility function is done with those variables as input.

5.7.2 Image Variables

Image data in Phission is represented with several variables to define width, height, format and the image data. Width and height are both 32-bit unsigned integers that can be any size which the underlying system can handle. Smaller dimensions are processed faster so generally the dimensions are kept to a width and height of either 160x120, 320x240 and occasionally 640x480. The image data are a contiguous buffer or array of unsigned byte values. The data are packed by rows where any pixel of (x, y) can be accessed in the buffer by using the following equation: `pixel(x, y) = buffer[ x + y * width * channels]`. Getting to a specific row (the y coordinate) within the buffer, one must multiply the desired row index by the data size in bytes of each row. The size of a row is the image width times the number of channels used to represent a specific image format. The format is currently stored as a single 32-bit integer value with the specific format being a bit field set within that integer. For example, if the second bit in the integer is set, then the format is a 24-bit RGB format.

5.7.3 Image Formats

The list of supported formats includes grayscale, RGB, HSV and YUV. Various byte orders of RGB are supported to allow for the variations in how capture APIs package image data within the buffer. One format of HSV is supported because varying the byte orders is unnecessary. The YUV format decimates the color channels by 4 and packs the data in a planar manner. Planar means that the bytes which represent the luminance (the Y component) are stored in contiguous memory locations. The U and V components are stored in a similar manner where the U bytes are stored immediately after the Y bytes and V immediately after the U bytes. Each U or V color component is the average U or V for a block of four Y pixels. This YUV format was coded mainly for reducing the amount of information required to transmit an image across the network. Special format compression was added in the form of JPEG [JPEG 2007] and Zlib [Zlib 2007]. Any format can be compressed with Zlib in a

lossless manner. Any format that is stored as JPEG is converted to RGB or grayscale before being passed to the compression functions.

5.7.4 Utility Functions

The utility functions include cropping, resizing and converting of image data. Cropping is the process of taking a rectangular region of an image and creating a new image from it. Cropping is done on regions of an image which were identified by a segmentation algorithm. Those regions can then be passed through an identification algorithm if necessary. Resizing can be done using either the nearest neighbor algorithm [Wikipedia 2007j] or the bilinear [Wikipedia 2007k] algorithm. Resizing is most often used for display purposes. When a user resizes a display window, the resizing utility functions provide the ability to take an image and fit it to the window. Resizing can also be used to further reduce image size when sending images over the network. Converting is used to change from one image format to another. Some conversions can be done in place such as RGB to BGR. Other conversions require the use of an output buffer such as YUV to RGB. Not all possible combinations of conversions are supported, but a significant effort was made to support as many of those thought to be commonly used or easier to implement.

5.8 Dataflow Toolkit Layer

The Dataflow Toolkit layer is a small layer of code made up of the `phLiveObject`, `phLiveCookie` and `phDataObject` classes. It is meant to be distinguished from the other layers because it contains the source code and algorithms for the thread-safe asynchronous method of signaling new data to be consumed and for transferring that data. Other methods of data signaling and transfer, such as a stream structure, would be implemented in this layer. Current research for this thesis has identified several methods for the movement of data through the framework using the current server/client dataflow structure (i.e., push and pull) [Manolescu and Nahrstedt 1998, pgs. 84-91].

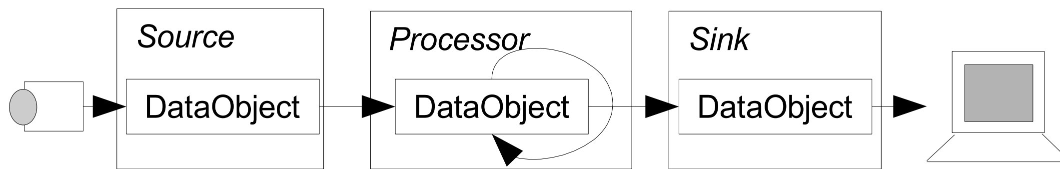


Figure 15: Data-centric signaling and transfer of data.

One method is data-centric and focuses the sourcing of data around the object, or class instance, containing the data (see Figure 15). Centering the signal and transfer on the data allows an inheriting data object to act as a semaphore for listening clients. Data that is placed in the object signals a client and when a client (e.g., a processing class) is ready then the data object will be copied into a private instance.

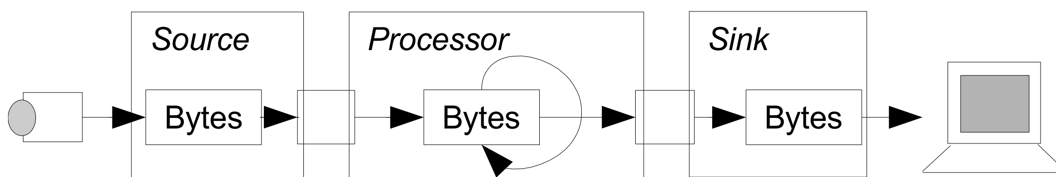


Figure 16: Byte-stream method of data transfer.

Two additional methods of transferring data use streaming structures[12]. A stream can be made to connect sources directly to sinks (as seen in Figure 16). As data are acquired (or processed) by a source, the sink receives byte packets through a byte-stream. The source node continually writes to the stream and the sink node continually reads from the stream. The other type of stream shuttles nodes that carry data objects (see Figure 17). A source is connected first, and then the processing and sink entities can connect to the conduit in any chosen order. These downstream entities wait for information to flow by. When a node is passing a connected entity, the processing components search for data to process and if they have any output they add it to the node. Node-stream processing is the main method used by [François 2007a] for transferring data between sources and sinks.

12 Phission does not use these method but there are listed here to further define the purpose of the dataflow layer. There are several additional ways to transfer data between threads, but the three mentioned above appear to be the most common.

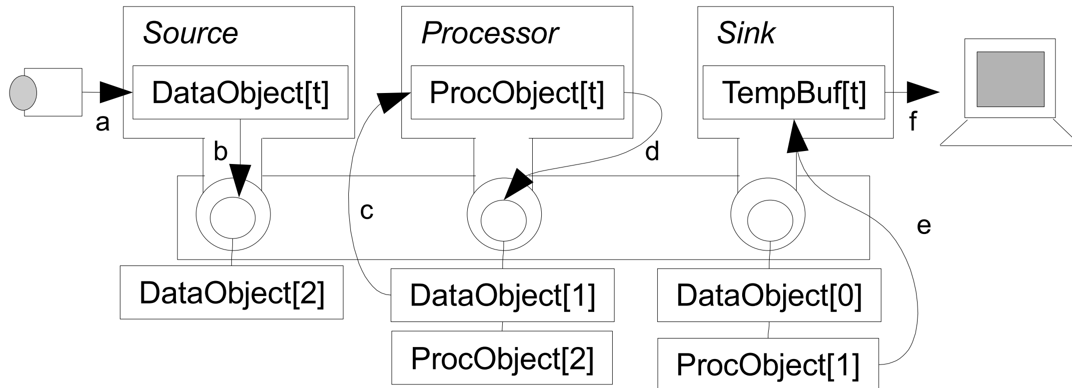


Figure 17: Node-stream method of data signaling and transfer (at t=2). Data are acquired (a) by the Source and added to the node (b). The node moves to the Processor which gets the data (c), processes it and adds the processed data object to the node (d). The node then passes to the Sink which looks for the processed data (e) and sends it to the display (f).

5.8.1 `phLiveObject`

The `phLiveObject` encapsulates the data-centric method of signaling and data transfer. Data classes can derive from `phLiveObject` and make use of the thread-safe signaling and transfer with little coding requirements. Data-centric transfer is the main method of moving data between threads in Phission. Data are transferred from a capture class to the processing pipeline and eventually to a display class or some other endpoint through a series of semaphore postings and memory copies.

5.8.1.1 Data Centric Signaling and Transfer

The design of a data-centric signal and transfer system was done because it is a simple logical first step[13]. An image object contains member variables for image data (stored as a buffer), width, height and format attributes (see section 5.7.2). The simplest way to get data from one thread to another is to copy the buffer and attributes. The alternative is to pass a buffer from one thread to another thread using pointers. Pointer passing would only serve a

13 The Video4Linux API also influenced this decision because frames are grabbed as a whole rather than in sections. When synchronizing on a capture request, V4L1 only returns once the entire frame is completed.

vision system that employed a very basic design which allows one sink per source. In other words, a capture class could only connect to either a display or a pipeline but not both. Instead, the common example situation (taught in Operating System courses), where one producer creates data for any number of consumers, fits the profile.

Having had experience working with the producer and consumer scenario, coding the initial algorithms was quick and easy. Further modifications were made to use reader/writer locks where basic mutex structures were initially used. The signal and synchronization code was abstracted and encapsulated into the `phLiveObject` to allow for easy reuse in any data class. The memory copy method of information transfer is extremely useful given its logical simplicity, however, its scalability is hindered by the linear growth of memory bandwidth requirements as the number of clients increases.

5.8.1.2 Three Method Client Interface

The application client interface includes connect, update and disconnect. The `phLiveObject::connect` method allows the destination or sink instance to associate itself with the source instance. The disconnect method removes this sink to source association. The `phLiveObject::update` method uses this association to wait for the source instance to signal new data has been assigned to the data object (by calling the `phLiveObject::notify` method internal to the data class). The Framework layer classes for capture, processing pipelines, and displays make use of this simple three method interface to create an automatic movement of data from thread to thread as new data are available. Automatic data movement is a key to reducing the load on a developer who would otherwise need to construct and design dataflow through the processing framework.

Some programming toolkits provide a means to get data and to display data, yet require the developer to explicitly call the functions which do so. In addition, the developer needs to pass the data around to the different pieces of the vision system. Phission provides the ready-made components which already perform these common tasks to allow for basic vision system processing frameworks to be created. However, Phission also allows the aforementioned manual (i.e., not automatic) method of data updating and transfer in user

code by calling the `phLiveObject::update` method explicitly (where the object method being invoked is connected to a source).

5.8.1.3 The Cookie

An association between a sink data object and a source data object is made through the `phLiveObject::connect` method. `phLiveObject::connect` allocates a cookie (`phLiveCookie`) and places it in a list structure. The source data object iterates through the cookies to signal sink data objects that new data are ready to be copied. The cookie is a holder of a key and inherits from the `phSemaphore` class. The key is used as a redundant error checking mechanism to protect against a situation that could arise from internal software errors. The key is checked against the source pointer to make sure they are the same and that the destination object is correctly associated.

Inheriting from `phSemaphore` allows the destination object to check whether new data are available. The use of non-blocking or blocking semaphore takes are permitted which can allow a thread (that calls the update method of the destination instance) the ability to give up its time-slice or perform other actions until new data are placed into the source instance. A reader/writer lock is used to allow virtually parallel copying of data from the source to the sink object after the sink instance receives the signal that new data are available. A reader/writer lock implementation permits distribution of data from a single source to any number of sinks. The efficiency of the current implementation can be hindered if the data being copied from source to sink requires a significant amount of memory bandwidth and CPU.

5.8.1.4 Swap and Copy

The `phLiveObject::swap` and `phLiveObject::copy` methods are simple methods that take a `phObject` pointer (from which the Phission classes are derived). The name stored in the passed `phObject` should match the name of the class whose swap or copy method is called. Checking of the object name occurs before a recasting of the pointer to `phDataObject` or any other data class type that inherits from `phLiveObject`. The

actual call to code the `swap` or `copy` method is called from within the `phLiveObject` class when new data are available.

The `phLiveObject::connect` method allows for a choice of either swapping data or copying data to an object that is associated as a client with another object. For implementation purposes it is usually the case that there are methods such as `swapData` and `copyData` in the inheriting data class. Having similarly named methods are to allow easy hooking into the `phLiveObject` interface by using preexisting copy and swap methods that are used in application code. These methods are where the actual copy and swap take place in a write-protected region. Within those methods all of the member private variables that are necessary to copy are copied from the source to the destination object.

5.8.2 `phDataObject`

The `phDataObject` is an example of a class that inherits from the `phLiveObject` class that implements the internal interface set of methods. It encapsulates a single buffer of data and the size of that buffer. The three methods for internal implementation are notify, swap and copy. The notify (implemented in the `phLiveObject` class) method is called immediately prior to releasing the write lock on a critical section of code where new data has been copied or swapped in the respective methods of `phDataObject`. The notify method also traverses the list of cookies that represent the other connected `phDataObject` objects. When the method completes and the write lock is released, the connected objects get read access to the object and copy or swap the data. In the case of a swap, only one object should be connected.

5.9 System Toolkit Layer

The System Toolkit layer is a common layer for other Phission software layers. It includes macro groups, reusable classes, and function sets. It provides interfaces for C++ threads, mutexes, reader/writer locks, condition variables, semaphores, network sockets, time and error functions, and the standard integer types [ISO/IEC 2005, sec. 7.18]. The mutex, reader/writer lock, semaphore, and condition variable classes are simple implementations which provide little extra functionality than other implementations. This collection of classes,

macros and functions is called a toolkit because it does not influence the architecture of upper layers [Gamma et al. 1994]. The major importance given to this layer is to provide all of the portability necessary to support the rest of Phission.

Phission is focused on providing the pieces for the system in which vision code runs and for the ability of that system to process live data. Full blown portable interfaces for every feature of an operating system were not necessary. One such case is file support. Only a portable method of file locking is provided without providing for rest of the file interface. Besides, there is a defined file interface in standard C which both Windows and Linux provide. However, there are minor details which required the need for that specific file locking interface.

There are a few basic groups of code that are meant for utility and support. The first group is the variadic utility macros used for printing and error checking. There is a Quicksort [Wikipedia 2007f] group of functions that are used in the segmentation code contained in `blob_Filter`. Those Quicksort routines are reusable and not specific to that blob code, so exist in the System Toolkit layer for use by any Phission or user code. There are the thread-safe list classes which can be the basis for queue structures that move data between threads. There are time related functions and classes that support some profiling code for testing and debugging. These include a class that holds a single time stamp value, one that holds two stamps which make up a time interval, and another class which uses the list class to create a record of timing information for profiling processing filters and pipelines.

While using macros increases the footprint of the binary, it is quite useful for several reasons. The first reason is that no function overhead is associated with invoking the code that is being used. The macros can reference common local variables without function overhead (i.e., needing to pass arguments on the stack). The second reason is that common code fragments, such as those used in error checking, can be altered globally because the macro is defined in one place. The next feature of variadic macros is that they take any number of arguments. For the error checking macros, variadic macro support is very useful. The macro can be used to print an error message with any number of addition informative arguments

just as printf would allow. In the case of the print variadic macros, extra identifying information can be printed such as the function and thread id from where the message came.

Almost every function and method within the Phission code base makes use of a phFUNCTION macro. This macro provides the function variable for the print and error checking macros. It also declares the common return code integer variable, rc, which is used quite often. The combination of error checking macros and print macros is a method of instrumenting the code in such a way that the instrumentation can be removed at compile time. While there are options for instrumenting code with the compiler, this is a standard and predictable way to instrument across different development environments. Since the macros lie in the global scope of the system, any printing and error checking code provided by those macros which is not necessary can be compiled out, allowing for the creation of a smaller and faster binary program.

5.9.1 phObject

Most classes provided by Phission inherit the interface of the phObject class which serves two purposes. The first is to allow any object to be cast as a phObject pointer and passed to any function accepting the phObject pointer type. The second is to allow runtime object inheritance tree or name checking to facilitate recasting of the pointer to any class within its inheritance tree as needed. One instance of the runtime name checking is within the Framework layer system management class (phSystem) to allow it do determine a capture, pipeline, display or user class from a generic phThread pointer and calling the appropriate startup method. Implementing a class derived from the phObject is as simple as making a call to the phObject::setName method which sets the name for the object. Every object constructor call will add a name entry into an inheritance list stored by phObject.

5.9.2 phMutex

The phMutex class provides the interface to a portable C++ mutex and the classes that are thread-safe usually inherits from it. It provides lock, unlock and trylock methods that allow mutual exclusion to critical sections of code within any derived object. There are a

group of locking macros to reduce typed code and to provide error checking and handling when something goes wrong.

5.9.3 phRWLock

The phRWLock class provides a C++ interface for a reader/writer lock. A reader/writer lock allows for two types of critical sections. The first is where only one thread is allowed to access a region of code for the purpose of modification: a write lock. The second is where any number of threads are allowed to access a region of code but are not permitted to change the state of the protected entity: a reader lock. The phRWLock is used mainly by the phLiveObject (see section 5.8.1). It is meant for use with data classes that permit a source to place data into the object and any number of thread to read the data out with as much concurrency as the system will allow. Reader/writer locks are a more efficient means of protection and dissemination of data than using a simple mutex.

5.9.4 phSemaphore

The phSemaphore class provides a portable interface for a C++ semaphore object. A semaphore provides for posting and taking of tokens that signal the availability of a resource. Sometimes it is used simply as a means of blocking a thread from continuing until a specific action has taken place. Only one thread or consumer can receive each token that is posted. The most important use of the phSemaphore is within the phLiveObject. It is used as a means to signal an "attached" or "connected" object that new data are available.

5.9.5 phCondition

The phCondition class provides a portable implementation of a condition variable. A condition variable is used to signal a group of waiting threads that a certain condition has changed such as the value of a variable. It is different from a semaphore in that one thread can signal many waiting threads instead of just one thread by using a broadcast. However, the condition variable also provides the ability to signal one thread at a time using the signal method.

5.9.6 `phConditionCounter`

An example of a class with extended features is the `phConditionCounter` class. This class was coded to keep track of how many clients are connected in the `NetDisplay` class. Since the `NetDisplay` is a multi-threaded server, there can be any number of connections which the system can handle. However, in the case where there are zero clients connected, it is not useful for the server to continually perform operations that go unused. `phConditionCounter` is derived from the `phCondition` class and adds an integer variable with methods to increment, decrement or assign a value to the integer. The `phConditionCounter::waitForValue` method waits for the counter variable to reach a desired state (equal to, greater than, not equal, etc.) with respect to a desired value. The `phConditionCounter::update` and `phConditionCounter::wait` methods contain the code needed to deal with the condition variable.

5.9.7 `phSocket` and `phServerSocket`

There are two classes for portable network sockets for both client and server operations. The `phSocket` class implements the necessary client operations such as `phSocket::connect`, `phSocket::disconnect`, `phSocket::receive` and `phSocket::send`. The `phServerSocket` class inherits from `phSocket` and adds the server related features for setting up a server that include the listen and accept methods. One reason for these classes is to provide a C++ interface to sockets and two separate classes for easily distinguishing a server class of code and client code. It also provides for the minor differences in the way the Berkeley sockets API [Wikipedia 2007g] is implemented differently on Linux and Windows as well as the use of native Win32 sockets. There are some utility methods that allow the sending of different integer types such as short or long integers and convert to network byte order.

5.9.8 `phThread`: The Phission thread class

The `phThread` class is a C++ threading class that extends the normal set of threading features. Careful design for the Phission threads makes sure they execute necessary cleanup. Threads can be unblocked, do not have to be canceled or killed, and handle errors properly. It

allows the code provided by Phission to be more stable. User implemented classes can use Phission thread classes code as an example of how to write similar code. The phThread is a very important part of the Phission system because concurrency is a key design method (see section 5.10.4). Concurrency is one of the necessary system requirements of a vision system SDK and a significant amount of effort went into developing and debugging this Phission class.

The user/application level code can easily start and stop a thread using the appropriately named phThread::start and phThread::stop methods. The return codes for each should be checked to make sure they executed properly. As opposed to traditional implementations, careful coding has been implemented to allow the phThread::start method to be synchronous. It waits for the spawned thread to signal that it has started successfully or an error has occurred. Checking of the return code allows the application to stop and print errors so they are detected early during runtime. The phThread::stop method will wait until a thread completes before returning. It does not cancel or kill the thread as it is expected that any thread implementation would have been carefully constructed according to the design methods. The thread must be capable of being signaled to clean up in a sane manner. There is the option of making a thread run in a "detached" state which makes phThread::stop execute asynchronously. The thread is signaled to cleanup, and phThread::stop will attempt to unblock the thread, but will not wait around to find out whether the thread finished. An example of where detaching is useful is in the NetDisplay multi-threaded network display class. The best method of cleanup the client threads is to let the clients clean themselves up which happens when a thread is set as "detached."

Status about the thread can be retrieved with the phThread::isRunning, phThread::getReturnCode and phThread::detached methods. These respectively return the state of the thread, the phThread::run return value when the thread was stopped, and whether the thread was started in a detached context. The phThread::equal method takes a thread argument and compares it to the current thread

to check whether they are the same. A similar static method takes two thread class arguments and compares them against each other. Priority adjustment and query methods have not been implemented as part of the thread features in the current release of Phission.

The `phThread::run` method is for the code to be executed by a thread. The only necessary protocol in using the Phission thread is to call either the `phThread::signal_running` or `phThread::signal_error` methods from within the `phThread::run` method. Those signal methods allow the spawning thread to drop through `phThread::start` and continue in its context. Up until the point at which the two signal methods are called, the spawned thread has a lock on its object to allow any setup, checking or adjustment of its class private member variables. The `phThread::signal_running` signals to `phThread::start` that all is well in the new thread and that it will begin executing its main loop. The `phThread::signal_error` method notifies the spawner of some major error during a threads startup such as an unavailable resource such as an acquisition or capture device.

The thread class provides several extension methods for executing code at different stages during the lifetime of a thread. The `phThread::setup` method is called from within the context of the new thread prior to calling the `phThread::run` method. The `phThread::cleanup` method is an extra place to put cleanup code that gets executed within the context of the thread that called the `phThread::stop` method. The `phThread::error` method is called from the context of the thread that called `phThread::start` if there is an error starting up a thread and allows any special reinitialization of variables that `phThread::setup` does not handle. Finally, the `phThread::wakeup` method allows a `phThread` derived class to unblock the thread associated with it. It can use any of the Phission classes that support `phThread::wakeup` methods to unblock the thread. Waking a thread up and unblocking it in a safe and clean manner is one of the essential design choices in the Phission vision processing system.

Current thread interfaces that are supported by Phission include native POSIX threads (pthreads) [Nichols et al. 1996; Wikipedia 2007h], Win32 threading [MSDN 2007] and VDK threads [VDK 2006, Chapter 3]. Pthreads is supported for use mainly on Linux but should allow quick porting to other platforms that support pthreads. Originally, pthreads were used by Phission as the means of threading in Cygwin/Windows but native Win32 threads are now supported for both Cygwin and Visual Studio 2005 applications. The VisualDSP++ Kernel threads provide the threading interface for VisualDSP++ supported targets, but the Analog Devices' Blackfin EZ-Kit is currently the only tested target. The addition of other thread interfaces is done internally to the `phThread` class for reasons of encapsulation that allows the Phission framework to be easily ported without disruptive changes.

5.10 Phission's Design Methods

Phission makes use of a number of design methods, such as taking advantage of object-oriented design to simplify interfaces and provide for reuse. As such, most of Phission is object-oriented but not all of it because standard C can still have its benefits, so functional programming is used for the implementation of certain source code. Checking every return code in a C/C++ application is a very important requirement. C++ adds exceptions as a means of returning and signaling errors but they are not implemented in Phission because their runtime performance is questionable. The template feature of C++ was thought to be confusing for an inexperienced developer, which Phission targets, so they are not used in the current version.

Using Phission in other programming languages is a very important design choice because it permits extension and application of a standardized vision system structure in languages that might be better for some applications (such as GUIs). Phission classes and methods maintain a standard naming scheme to reduce conflicts with language reserved words or standard classes provide by the languages. Several issues during development lead to the realization that the SDK should never have any library wide global data. Finally, the decision to make the system concurrent requires blocking often, providing for thread-safety and the ability to cleanly stop an executing thread.

5.10.1 Object-Oriented and Functional Programming

Building the entire software development kit using layers which get successively easier to use is an important feature. The C++ language provides for the necessary object-oriented features such as encapsulation, abstraction and inheritance. Specific knowledge of capture, display and system APIs were abstracted into a common portable and modular interface. A specific capture or display interface can be chosen through simple choice of class and the control of the different capture or display devices use the same methods.

The system APIs are abstracted into Phission classes which supply an unchanging interface that works the same across platforms. The ability to inherit from an abstract capture or display interface class allows a third party to extend the existing support for those devices and allow it to work the same as the Phission provided classes. The inheritance feature of object-oriented design allows a developer to add the dataflow features of the Dataflow Toolkit layer to their object. Finally, encapsulation allows the high level framework classes to remain as they are while Phission changes internally. System performance can be improved without needing to change the application.

These features have been demonstrated to be crucially important during the development of Phission. Newer capture and display APIs were added to the feature set of the SDK. Each new capture or display class could be swapped out and replaced with the newer class with very minor changes to an application. Inheriting from the `phLiveObject` class (from the Dataflow Toolkit layer) allows a derived object to take advantage of the algorithms for synchronization and data transfer. The synchronization and data transfer algorithms are a key piece to the efficient functioning of the Framework layer in Phission. In addition, the encapsulation of what occurs when a capture, pipeline and display are connected to each other will allow for future data transfer structures to be implemented without needing to change the application level code. The use of encapsulation requires that no system specific header files be present in the public Phission header files that define the API. System specific details are left to the source files which are thought of as private entities where the header files are public. Besides being a good practice, keeping any system dependent code out of header files eases the wrapping of the Phission API into other languages.

Object-oriented design has advantages such as those described above but it is not always necessary to implement a feature or algorithm. Most of the Image Toolkit layer is implemented using standard C. The image utility code is simply better represented using C because it is a collection of functions that do a specific thing. Putting it into C++ would bloat the `phImage` class which wraps the C interface into C++ methods. Similar choices were made for mostly the vision algorithms, such as segmentation or blurring, where OO design does not afford any advantages over C.

5.10.2 Check Error Codes / Do not Use Exceptions

A critical design choice of any application is the checking of return status from functions or the execution of regions of code. Phission refrains from using exceptions internally to get return status because exceptions introduce questionable runtime overhead. Instead, Phission maintains the use of minimally standardized return code values. The definitions of `phSUCCESS` and `phFAIL` are zero and negative one, respectively. Every function and method that should have a return code will return one of these values with the exception of more informative return codes.

Failure return codes that provide more information are less than the value of `phFAIL`. Return codes that designate some status other than complete success are greater than `phSUCCESS`. An example of a return value larger than `phSUCCESS` is the returned value from a non-blocking take on a semaphore. If the semaphore was available, then `phSUCCESS` will be returned. However, if no semaphore was available, then a positive value is returned. Returning a positive value denotes the method succeeded without error but did not succeed in getting the semaphore. Phission provides a set of macros that conform to these specifications by checking the return codes for a value less than zero and responding appropriately by printing and possibly returning from the function with a new error value.

The mandating of using return codes and not exceptions by using macros, which print informative error information, allow for problems to be quickly tracked down and addressed. It is even possible to completely debug the error that occurred from the error messages

generated. Occasionally, the Phission code is capable of handling the error or issue cleanly without requiring the application to exit. Finally, the overhead of checking return values is predictable whereas exception handling is provided by the compiler and overhead can not be directly known.

5.10.3 Multiple Programming Languages

Multiple language support has always been a design requirement because Phission was originally meant for use with the Pyrobot [2007] software, which is written in Python. While it may not seem necessary for a vision processing SDK to support multiple languages, it has proven to be very necessary given that each language has its own advantages over other languages.

Python is not very efficient as a language in terms of performance because it is an interpreted language. Most the algorithms meant for vision applications are currently implemented in C or C++ toolkits and can be custom written in C/C++ to be faster. C/C++ is also a very common language for embedded applications and a common denominator for many applications. However, Python is very useful as a language for quickly and easily creating software. While it is inefficient when it comes to coding algorithms and working with memory, one does not have to worry about pointers or have to deal with recompiling all of the time. The Java language is widely supported and allows one to create a powerful and portable user interface. Phission is written in C/C++ and loadable modules are provided for both the Python and Java languages.

The fact that not everyone programs in the same programming language is reason enough to provide a tool which can be used in any language. The facilities that the vision processing system can be integrated with, such as The Java Media Framework in Java, are a huge advantage. The SWIG [2007] tool is used to create the loadable modules for Python and Java. Other languages can be included in the list with minimal effort and changes required to Phission. The Python and Java languages did require some changes to the Phission interface. It actually enforced the need to encapsulate the system dependent knowledge into the source files. The SWIG tool uses the public header files of Phission to create the loadable modules

and interface for the other languages. Keeping the public header files as clean and portable as possible make sure that future languages can be added easily.

5.10.4 Concurrency

A principle design feature of Phission is that it makes use of many small single purpose threads. One could argue that threads are not necessary and processes would be better. Threads are not absolutely necessary and processes could easily be built into the Phission components because encapsulation would facilitate it. The `phThread` class could make use of process forking instead of thread spawning and the `phLiveObject` could use Inter-Process Communication to transfer the data from one process to another. The implementation of the concurrency is not a concern because it can be achieved with threads, processes or a custom context switching system. The importance is that the concurrency exists within the system.

With the advent of processors offering threading optimization (e.g., Pentium Hyper-Threading, SPARC threading) and multiple core CPUS (e.g., Analog Devices' Blackfin, AMD 939/AM2, Intel Core Duo) that are available to the common consumer, concurrency has become even more important. Some resources, like the capture and display, can be run with some concurrency even on a single processor computer by using DMA to transfer image data. The use of DMA frees up CPU time that can be used to process the images being captured.

Achieving the best results means the vision system must take full advantage of these concurrent features. Creating a concurrent system also simplifies the design of the system. Each thread that runs in the system is tasked with performing its own role of the vision system. For example, the capture class can capture an image and then wait for the next image. Capture and wait is a logically simple task that can be given to a capture class. The same can be done for a pipeline class, display class or some other class. There is even a thread in the `NetDisplay` class which is meant to wait for a client to connect and then create a new client thread.

5.10.5 Thread-Safety

Due to the design of Phission and the structures used, thread-safety is very important to the stability of the SDK. The thread-safety is only present in the C++ classes as they are the structures expected to be accessed by more than one thread at a time. The thread-safety within the C++ classes is taken to an extreme. Most methods lock the object before touching private member variables which alter the state of the object. Locking the object is done through the inherited `phMutex` interface or an instance of a `phMutex` object meant for a more specific critical section. However, there are optimizations that can be made while still providing thread-safety such as the introduction of the reader/writer lock to the `phLiveObject` class. The reader/writer locking allows many threads to copy data from an object at logically the same time instead of restricting it to only one thread if a mutex were used.

5.10.6 Threads Block As Often As Possible

With the use of concurrency there is the possibility of a thread "spinning" or eating up CPU time without doing anything. A temporary solution to spinning is to yield the processor as soon as a task is finished so another thread can execute. Spinning and then yielding is not very practical for the real world but it served to help in debugging deadlock problems in the initial stages of Phission development. Solving the spinning issue is done by "blocking" a thread or removing it from the run queue until there is a task for it to perform. In addition to the design principle that follows the rule of using numerous small threads, Phission also maintains that every thread should block as often as possible and should never spin. When a thread is unblocked it allows for a low latency response in that thread performing a task.

5.10.7 Thread "Wakeup" Feature

Blocking introduces an issue of how to cleanly stop a thread if it is blocked on some resource. The implementation of a "wakeup" feature, which is not standard in thread and synchronization classes, was necessary to solve the problem of cleanly unblocking a thread. Each thread has a wakeup method which can be overloaded to signal any objects or resources which might be blocking the thread. Wakeup methods in the synchronization objects return specific values to anyone who is blocked on that object to denote that its wakeup method was

called. Thread-safety is important here because it requires exclusive access to the synchronization objects to set the wakeup state.

Any threads that try to block after the fact might miss the wakeup state and stay blocked so the wakeup state is usually persistent and must be reset in the `phThread::setup` method. Locking the thread object is prohibited while in this method because it introduces the possibility of deadlock so only one thread should be calling this method at a time. Any method call which returns an error code designating wakeup should cause the thread to check the `phThread::isRunning` return value when a wakeup is received to see if the thread is being stopped. Implementing the wakeup method requires stepping through the possible execution paths of a thread to find any blocking points and introduce a method of waking up for each.

5.10.8 Cleanly Stop Threads: Never Cancel or Kill

This is the next design principle that falls into the concurrency department. Every thread must be cleaned up properly and safely. Canceling threads or killing them is deemed improper because memory may be left allocated with no way to free it. Even worse is the case where a capture or display resource is still in use (within the single process scope of a threaded application) even after the thread that was using it was stopped. Also, there is the possibility that a lock was acquired by the thread that cannot be released by any thread except the one that was just killed. In any case, canceling and killing is not a portable feature, so the "wakeup" methods were devised to permit the carefully stopping of the threaded system and allow the threads to properly clean up resources by freeing memory, releasing or closing hardware resources and releasing all of the acquired locks.

5.10.9 Preallocated and Resident Memory

Since Phission is designed to be used on live continuous data, there are several places internally where preallocated and memory resident buffers or structures are used. These buffers and structures can be reused for further calls to the same functions. Allocating large segments of memory during runtime and then freeing them shortly after may introduce unnecessary overhead. While it has not been tested and proven, improvements were noticed

with the addition of code that uses preallocated buffers. Within Phission, the image utility functions and several video filters make use of resident buffers that are only freed when the object is destroyed or explicitly cleaned up by a reset method. There are a set of dynamic memory macros dedicated to readjusting memory buffers to fit possibly runtime variable dimensions of data used by the video filters. Examples of filter that use them include the motion, add, subtract, average and double-difference filters.

The use of memory resident buffers ultimately limits the size and scalability of a vision processing system built with Phission. Future work will seek to solve this memory scalability issue, test various solutions, and provide a means to improve the vision system performance. One such solution that was seen to be essential is the use of the workspace image within the pipeline. Instead of having a buffer be copied from the output of one filter to the next filter, the input and output of an image filter is almost always the workspace buffer. It reduces memory copies but requires that the filters be run serially within the context of the pipeline thread.

5.10.10 No Global Variables

Phission mandates that there are no global data variables within the library. However, there may be global variables resulting from the inclusion of a system resource such as a header file or third party library. If there are no global variables, then each component of Phission can be used without the fear of colliding with another component. All of the objects have internal private member variables to maintain the state for each instance of an allocated object. One situation where global variables became a problem was with the use of the SDL [2007] library.

SDL provides a portable interface for creating games that include the use of audio, video and input devices. The problem is that SDL uses global variables for the display window which is used to display video and the `SDLDisplay` class can only have one instance running at a time. The same issue also occurred with the use of the portable FLTK GUI library. Running multiple instances caused unpredictable behavior and confusing errors. The `X11Display` and `GDIDisplay` classes were written after these issues were discovered and provide for

native displays on the Linux and Windows systems for multiple window video debugging. SDL does have the advantage of many years of development and fine tuning beyond what the `X11Display` and `GDIDisplay` display classes offer. To prevent Phission from introducing the same issue to other software which uses it, the mandate of no global data variables as a design method was introduced.

5.11 Conclusions

Phission satisfies the system requirements by providing for concurrency, modularity, extensibility, portability, dataflow and coherent packaging. The concurrency is provided by the `phThread` class and related synchronization structures. The framework classes for capture, processing and display inherit from the `phThread` class and provide for concurrent execution of capturing, image processing and display of output. Users can easily add their own concurrent structures, such as a two input processing pipeline, by using the existing software as a model.

Removing **concurrency** would introduce extensive latency in the processing of images and increase the complexity of the system design. The latency would result from the need to wait for the image to be processed and displayed before acquiring another image which also takes time. Some pipelines may be able to process faster than others. Displays may also be able to display raw input before the pipelines. Concurrency allows the tasks which do not take extensive time to continue running.

Phission is **modular** because it allows integration with existing software and is broken into manageable components and layers. Being modular allows a program to use Phission without needing to change the existing application significantly. It resembles a component which can be plugged into the existing software and provide a vision system capable of live vision processing. Phission is broken into six layers that build on each other and do not require the use of global variables, which is an important aspect of modularity. In addition the capture, processing, and display components are completely separate and understandable modules of code.

Unsupported code can be compiled out and provide for customizations on the target platform. The usual method for compiling a software library is through either a project or a makefile type system. Phission supports both types of development systems and also produces a single target that make it rather easy to include in applications and create new project setups for new development environments. Extensions to new capture, display and processing algorithm APIs are made through the creation of software modules that inherit from their respective classes.

Extensibility has several sub-requirements which include that the SDK allow use with other programming languages, support multiple development environments, and permit the addition of new capture, display and processing code. Phission can be integrated with any object-oriented language and currently supports Java and Python. This language integration is accomplished by removing system dependencies from header files and creating loadable extension modules for Java and Python to use the Phission software components. Phission also supports development environments for Linux, Windows, Cygwin/Windows and VisualDSP++ for the Blackfin processor. Newer development environments can be supported because of a few design methods used by Phission.

Phission is **portable**, which has been shown through support for Linux, Windows and VDK as target platforms. Application layer code requires no changes because the System Toolkit layer allows porting to new target platforms by encapsulating the changes internally. Phission can thus be said to be a portable vision system software development kit. Porting usually involves the addition of a specific capture or display interface which is provided through the modular design of Phission. Another aspect of porting can require the extension of supported development environments, such as Visual Studio 2005, for which there is validated precedence.

Dataflow handled automatically in Phission when the vision system components have been connected to each other using the appropriate methods. Instead of requiring the user to actually write code that transfers the data, it is implicitly done during the vision system runtime and flow is directed through the component connections. The Dataflow Toolkit layer

provides a class (`phLiveObject`) which defines a method of signaling and transferring new data. The explicit calls to `phLiveObject::update` (which transfers the data) are encapsulated within the Framework layer classes.

Phission is provided as a single library which can be compiled using a few commands and uses consistent naming of classes, functions, and files. It does not define any methods in header files (a requisite of creating language modules), has no global variables (which effect modularity), and includes documentation for the whole API. Phission can be obtained without requiring the need of other packages (other than the build system software or SWIG if C/C++ is not the target language) and still be used because of piece of Phission can be compiled out.

The removal of any single requirement hinders its ability to function effectively as a vision system software development kit. Proof that each requirement is necessary became apparent during the development of Phission as each was crucial to its effectiveness in vision processing applications and lab research. Other vision system software may be lacking one or more of these minimal requirements and are not effective because they may require more design, porting or dataflow work on the part of the developer to actually use it. Phission is effective because it aims to do everything for the user from automatically capturing image data, transferring data, displaying data, executing developer chosen filters on data and allowing data to be molded into a thread-safe class that can signal new data without much complexity.

6 Future Work

There are many things that have yet to be done with Phission. This chapter outlines future works that centers on Phission enhancements and possible research that can be done. There are structures, data movement approaches and areas of study that have yet to be explored including different data types other than video. Phission does not provide for an optimal runtime system and creating structures that can provide a more optimal runtime is important. Performance analysis is going to be a critical step in validating some of this future work as Phission aims at being more of an optimal solution.

6.1 Design Patterns

There are a great number of books on Design Patterns for software engineering. Wikipedia.org is one place to start for finding a list of books on the subject [Wikipedia 2007l] or a search of any of the online book retailers. Employing design patterns in Phission would allow for some quick solutions that have been verified and possibly allow some Phission classes or structures to be redone. Whatever the case may be, design patterns are a reasonable next step towards improving the Phission API and strengthening its already unique offerings for constructing vision processing systems.

6.2 Data Event System

It is not expected that everyone will implement a robot control loop as most Phission applications researched in Chapter 2 used. An idea came up that perhaps an event system with processed data outputs or events from Phission would be useful. An event system that provides a similar interface as the X11 event queues. Everything would still be threaded on the inside where the processing and data movement occur. However, the developer application code could include an event loop instead of a control loop. They could insert their own processed data into the event queue from another thread such as sonar or laser and possibly processed sonar and laser. The design of the application would be very similar as it is now except the event processing loop would control the robot and there would be threads feeding it processed information.

The processed information would carry a certain priority value that would place important data on the front of the queue most the time and superseding lower priority information. An example would be high priority processed sonar or laser versus low priority segmented visual information, or vice versa. This application would require research into existing event systems. Robot control system research is also a good place to start. How efficiently it would work and whether it would be practical is likely to have been answered partly in some research papers. For example, an idea of data buffers instead of event systems may be more useful. Data can be fed into buffers where sonar data packets are retrieved for analysis. Any of this research is likely to be useful for components that can be added to the Framework layer.

6.3 Generic Data Processing

The Modular Flow Scheduling Middleware software [François and Medioni 2000, pgs. 371-374; François 2007a] and the Software Architecture for Immersipresence [François 2003] have a strong architectural design with the capability for generic data processing. Phission was always written with the expectation of moving into fields other than vision processing. Generic data handling has even been explored with by wrapping the Player/Stage client API into a series of Phission classes that inherit from the `phLiveObject` interface. It allows a client thread to act as a source thread by placing the available robot state information it receives into output data classes for the application to consume. The issue is that the manner in which the application consumes the data in an ad hoc design is a result of the robot control loop paradigm.

Providing structures that could carry any information to any processing entity would improve the Framework layer offerings and allow for better application design. MFSM/SAI already has an architectural solution that uses data “pulses” to carry information and propagate it to processing “cells.” A pulse can carry any type of information. Introducing these features into Phission would also permit better sensor fusion. While MFSM has a strong architectural answer to optimally processing data, the current modular software packaging (not to be mistaken with its modular software design) is divergent from the single software package of

Phission. SAI structures and ideas can be implemented on the Phission software layers instead of implementing MFSM interface features exactly.

Phission was designed to always progress towards an optimal and efficient solution. The first priority, however, has always been to provide a very reasonable and useful runtime with most of the focus on portability. The Framework layer's capture and display components in addition to the System Toolkit layer received most of the effort during development and Phission is useful on more than one platform because it provides support for those numerous capture and display APIs. There are several things which still have yet to be done in each layer that can improve Phission as a whole. Some of these cross cut layers and some are solely a per layer enhancement that can be done opaquely using encapsulation.

6.4 System Toolkit Layer

Phission runs on three different operating platforms that include Windows, Linux and the VisualDSP++ kernel. As portability is on of the system requirements for an effective vision system software development kit, Phission should provide the ability to run on more than those three systems. Apple systems running OS X, or other versions of their operating system, are a possible target for implementation. In addition, there is the FreeBSD operating system and countless embedded systems to which Phission could be ported. Providing this common platform for vision system development can make one developer's efforts incredibly effective.

The error facilities are currently simple, as most the effort was directly toward refining Phission and making it portable. Return code values are always checked. If an error is determined to have occurred, then a message is printed and error handling code is executed. These facilities could be improved and since they are macros, it may not require much work in terms of instrumenting Phission. Exceptions could be introduced so that performance statistics could be taken on the differences between return code checking and exception handling.

A significant addition to the System Toolkit layer would be the improvement of the thread class. It currently offers synchronous thread spawning and stopping that may be impacting startup and shutdown performance or response. Applying asynchronous starting and stopping for certain groups of threads may increase performance of the system. For instance, spawning the threads that deal with displaying data could be started at relatively the same time and then synchronization could occur after they have been spawned. The same could be done for filter pipeline threads, capture threads and user threads.

6.5 Dataflow Layer

The dataflow features of Phission are limited to the indirect protocol of the `phLiveObject` class. The first objective of this thesis, to provide a computer vision system SDK, has been accomplished and there is need for more work to be done on different dataflow methods. There is better support for capture, display and differing operating systems than there is for the middle components that process and move data through the system. Structures such as buffers, channels, streams, and pulses [François 2007a] should be added. The data movement protocols should also be completed to include push and pull data movement [Manolescu and Nahrstedt 1998, pgs. 84-91].

The indirect method is an asynchronous type of data movement and synchronous data movement should be explored. Synchronous features could allow for normal synchronous and delayed synchronous movement of data. Normal synchronous movement would wait until all of the clients received the posted data before continuing. Delayed synchronous movement would continue to acquire new data and then wait until all of the clients received the last data before posting new data.

Time stamping each piece of data is a necessary feature for the future of Phission. It will allow movies to be created, Real-Time Protocol streaming and post analysis of processing and system performance. A data container class should contain the data, data descriptors, profiling information such as time and where the data has been. Such features are assumed to have been implemented in other software but will serve to improve offerings and usefulness of Phission.

6.5.1 Efficiency and Optimality

The improvement to the dataflow features is linked to the generic data processing future work in that these structures should be written to handle any type of data as the MFSM/SAI software does. An update is expected to be a major improvement in the operating efficiency of Phission. Phission is not a completely efficient or an optimal solution, yet it provides good results in real world live processing applications. These real world results can be seen in the list of applications to which Phission has been applied. Memory usage and transfer improvements can be made that would likely improve the runtime performance of Phission and potentially reduce its memory footprint. Currently, Phission transfers memory between modules using a whole memory copy. When this is done with small variables it has little impact, but when applied to a whole image buffer this can severely impact performance.

A solution would be reference counting and implementing a solution such as a "pulse" as done in the MFSM/SAI research. This pulse can contain all of the processed data resulting from an initial piece of data. A pulse travels from the source to the sink and can be freed only when the data are no longer needed. Pulses could completely remove the need to replicate the data with memory copies except for once at the source.

Each of Phission's temporal filters could replicate the data to save it for use in future iterations. The data may be saved unnecessarily and could be stored in a "passive stream" as done in the MFSM/SAI software. Reference counting of the data would keep it around only as long as it is needed, and it would be deleted as soon as the last reference to it was removed.

Removing these memory copies and holding the data resident in memory may actually reduce the amount of memory required because multiple copies of an individual instance of data are not made. Most memory copies are done using the CPU and this also reduces the CPU load. Since implementing these structures could significantly improve performance, performance statistics should be taken and analyzed. Comparing the performance of Phission prior to using these structures and afterwards could provide some useful information. Programs such as the Valgrind toolset can provide very accurate measurements. The

Valgrind programs simulate the processor and run the software. Instrumenting the code directly could provide an additional data set that would provide for real world results as well.

6.6 Image Toolkit Layer

The Image Toolkit layer has support for only a few formats. Adding an image type such as those found in computer vision toolkits (e.g., OpenCVand Gandalf), should be done to improve Phission's ability to represent more formats in a standard way. An overhaul of Phission code that uses image variables will result in simplified functions that take an image type containing all of those variables. The width, height and data pointer (which will change from `uint8_t *` to `void *`) are will be the basic variables in the image type. The use of a normal integer value will represent formats types and an extra compression variable to specify the compression method used on the data buffer, if any.

The current formats are specified as a bit field in an integer. At the time it was implemented, the bit field representation of an image format was satisfactory. However, its limits were very noticeable when attempts to add the Spherical Coordinate Transform format, or LAB format. Since all of the image formats are limited to byte wide channels, the SCT/LAB format could not be accurately represented because the spherical coordinates for values between 256 and 360 do not fit in a byte value. Similar issues were encountered with HSV formats, but are not as severe an issue because HSV is still useful.

Another image variable should be added to represent the data type (as other computer vision toolkits do) which can be signed and unsigned bytes, shorts, integers, doubles, floats or a more complex type. Code that uses large data types may be slower than smaller data types and are clearly going to require more memory. Specialized floating point processor features may actually process float data types faster than byte data types.

Finally, Phission is meant to be integrated with all of the other reviewed computer vision toolkits. Integrating with the large number of computer vision toolkits would require a set of conversion utility functions to convert from Phission's image type to the other toolkit image types and vice versa. A standard set of required utility conversions is likely to appear and any

developer integrating a new toolkit into Phission can provide for each new set of functions. All of those enhancements, as well as the revamped Image Toolkit layer, should make Phission very competitive in the computer vision system software development kit field.

6.7 Framework Layer

Phission is meant to support as many image capture interfaces as it can. The current interfaces supported provide only for the basics of each system and there are more advanced interfaces which are going to have to be added in the near future. The DirectX/DirectShow API will provide a more up to date capture facility for the Windows platform. Adding a V4L2 capture class will do the same for Linux. The FireWire interface provides a very fast image acquisition capability from very expensive high performance cameras. FireWire technology seems to have been used in many applications and FireWire can only be used with Phission through Coriander. Coriander redirects FireWire data into a V4L interface, for which Phission has support. Gazebo and Player/Stage offer the ability to have Phission integrate with completely simulated interfaced which reduces the cost of entry in real world data research to almost zero. Capturing data from a network client that uses RTP would be useful for remote applications. Finally, full support of file reading and writing formats should be added to allow for a variety of prerecorded data inputs.

There are almost as many display interfaces as there are capture interfaces which should be added. Again, DirectX will provide a better display on the Windows platform. Using the Xv extension of the Xlib library will do the same on Linux. A network display for the Real-Time Protocol would be very useful in remote applications where teleoperation is necessary or for simply monitoring the system. The current display on the Blackfin Handy Board is a network image server. There is no native display support. Adding hardware and then providing a software class to interact with it would be a very useful.

The Pipeline is likely going to be removed from the Framework layer in its current implementation. Research into design patterns may provide for a better implementation. Planned future work to introduce a pulse and stream structure to the Dataflow Toolkit layer may make the pipeline completely unnecessary. Each filter would connect to the stream in

order of desired execution instead of being inserted into the pipeline. Making each filter run its own thread could improve the concurrent performance of system as a whole and improve scalability. It will not be as simple threading the filters because one filter may be able to process data really quickly. Creating filter groups may be necessary where they all converge to a point where each filter processes input as fast as the slowest filter can consume.

6.8 Application Layer

The Application layer is going to be the target of advanced systems implemented within a single class. This could be a very good way to distribute research that uses Phission. Encapsulating a single purpose or multiple purpose vision system within a Phission thread would allow for a black box implementation of varying complex vision systems. A stereo vision processing class could output the location of objects in its near field of view. For a beginning developer, adding the stereo vision would then only require including the single class in a robotic application and start it running. Perhaps an advanced developer could develop a class that encapsulates a vision system which learns objects and then can identify them using a simple two method interface. The ability to contain the entire system in a simple C++ class interface could provide for many research opportunities.

6.9 GUI Layer

A new GUI layer should be added which would allow the construction of advanced vision systems that use Phission similar to a Rapid Application Development type system as that offered by LabView. The vision system could be saved to an XML format. The XML output would have to use already coded components because it would not be compiled. A Phission class could parse that vision system XML file and then construct the vision system automatically. The GUI could use Java, Perl, Python or Qt GUI toolkits to provide for portability. The GUI toolkit chosen should allow for fast displaying of video so that the vision system could be constructed and the results seen in real time.

6.10 Conclusions

There is too much future work to mention in this chapter mainly because most are minor updates and enhancements which are of lesser importance compared to the topics discussed above. The future work mentioned in this chapter is meant to give a view of Phission's larger

goals in providing for the best computer vision system software development kit available. The future work is going to require many man hours to complete. Phission development spanned several years where most of the above enhancements and ideas emerged from research that was conducted. The possibilities for future work in such an immense field as computer vision are practically unlimited. Even though Phission meant to target computer vision system construction, the Application layer and added algorithms are likely to expand indefinitely.

7 Conclusions

This document has detailed a list of system requirements for providing a computer vision system software development kit. A computer vision SDK must include facilities for capture, processing, and display, and have the ability to represent images. The SDK must be extensible, modular, portable, concurrent, and provide dataflow mechanisms. These requirements were found to exist in many other computer vision systems as well as related software and research. The motivation for the research in this document was to bring all of the requirements together into a single integrated package.

Phission is a product of the research done in this document and the integration of all of the aforementioned system requirements into a single coherent package. Phission uses a layered approach to build a platform independent, or portable, system with the ability to swap capture, display and filter components to build arbitrary vision systems on the supported platforms. The asynchronous signaling and automatic transfer of data allows the system to be very useful for robotic applications because it does not buffer data and allows the most recent information to always be accessible. The filter classes allow it to integrate any third party API by containing the specifics within the source code. The support of Phission has allowed a number of successful applications to be implemented.

Several applications were presented to show what type of scenarios drove the development of Phission as well as how successful they were at their task. These applications were rather simple to implement given that the Phission SDK provides for most the design and implementation. The design and implementation is made simple by using the high level objects that can be linked together. Using Phission, applications such as identifying a soccer ball and following a green trail with an autonomous robot can be assembled in the course of a day or two. Those applications, once coded, can easily be ported to any one of the systems that Phission supports.

There are many possible future paths to pursue to further the usefulness of Phission and computer vision software development kits. Each layer of Phission has room for improvement, as discussed in the Chapter 6. Phission offers a stable, well debugged and well

developed platform on which to develop. Expanding its support to more platforms, more capture devices, more display capabilities (e.g., RTP) and integrating many of the third party algorithm libraries will allow Phission to become a greater resource for computer vision system development.

8 References

[Baker and Yanco 2005] Michael Baker and Holly A. Yanco. “Automated Street Crossing for Assistive Robots,” *Proceedings of the International Conference on Rehabilitation Robotics*, Chicago, June 2005.

[BCMT 2007] Berkeley Continuous Media Toolkit, http://bmrc.berkeley.edu/research/cmt/, accessed on 21 April 2007.

[BFHB 2007] Blackfin Handy Board, http://www.cs.uml.edu/blackfin/, accessed on 23 April 2007.

[Blackfin 2007a] *ADSP-BF537 Blackfin Processor Hardware Reference*, Analog Devices, 2005.

[Blackfin 2007b] *ADSP-BF537 Blackfin Processor Hardware Reference*, Analog Devices, 2005.

[Blank 2005] Doug Blank. “AAAI 2005 Scavenger Hunt Objects,” http://dangermouse.brynmawr.edu/~dblank/scavenger/, 1 March 2005, accessed on 23 April 2007.

[Casey et al. 2005] Robert Casey, Andrew Chanler, Munjal Desai, Brenden Keyes, Philip Thoren, Michael Baker, and Holly A. Yanco. “Good Wheel Hunting: UMass Lowell's Scavenger Hunt Robot System,” *Proceedings of the AAAI-05 Robot Competition and Exhibition Workshop*, Pittsburgh, PA, July 2005.

[CImg 2007] The CImg Library: C++ Template Image Processing Library, http://cimg.sourceforge.net/, accessed on 21 April 2007.

[CMVision 2007] CMVision: Realtime Color Vision, http://www.cs.cmu.edu/~jbruce/cmvision/, accessed on 21 April 2007.

[Coriander 2007] Coriander, http://damien.douxchamps.net/ieee1394/coriander/index.php, accessed on 25 April 2007.

[CVIPTools 2007] CVIPTools: A Software Package for the Exploration of Computer Vision and Image Processing, http://www.ee.siue.edu/CVIPtools/, accessed on 21 April 2007.

[CVTK 2007] Computer Vision Tool Kit, http://cvtk.sourceforge.net/index.html, accessed on 21 April 2007.

[CVTool 2007] Computer Vision Tool, http://cvtool.sourceforge.net/, accessed on 21 April 2007.

[Desai and Yanco 2005] Munjal Desai and Holly A. Yanco. “Blending Human and Robot Inputs for Sliding Scale Autonomy,” *Proceedings of the 14th IEEE International Workshop on Robot and Human Interactive Communication (Ro-MAN)*, Nashville, TN, August 2005.

[DevIL 2007] “A full featured cross-platform Image Library,” http://openil.sourceforge.net/, accessed on 20 April 2007.

[DirectX 2007] DirectX, msdn.microsoft.com/directx/, accessed on 21 April 2007.

[FANN 2007] Fast Artificial Neural Network Library (FANN), http://leenissen.dk/fann/, accessed on 23 April 2007.

[Felzenszwalb and Huttenlocher 2004] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. “Efficient Graph-Based Image Segmentation.” *International Journal of Computer Vision*, Volume 59, Number 2, September 2004.

[FFmpeg 2007] FFmpeg, http://ffmpeg.mplayerhq.hu/, accessed on 15 June 2007.

[François and Medioni 2000] Alexandre R.J. François and Gérard G. Medioni. “A Modular Middleware Flow Scheduling Framework,” *Proceedings of ACM Multimedia 2000*, Los Angeles, CA, November 2000.

[François and Medioni 2001] Alexandre R.J. François and Gérard G. Medioni. “A Modular Software Architecture for Real-Time Video Processing,” *Proceedings of the International Workshop on Computer Vision Systems*, Vancouver, B.C., Canada, July 2001.

[François 2003] Alexandre R.J. François. “Software Architecture for Immersipresence,” *IMSC Technical Report IMSC-03-001*, University of Southern California, Los Angeles, December 2003.

[François 2007a] Alexandre R.J. François. “MFSM: User Guide,” http://mfsm.sourceforge.net/, accessed on 21 April 2007.

[François 2007b] Alexandre R.J. François. “Semantic, Interactive Manipulation of Visual Data,” December 2000.

[Gamma et al. 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gandalf 2007] Gandalf, http://gandalf-library.sourceforge.net/, accessed on 21 April 2007.

[GStreamer 2007a] GStreamer, http://www.gstreamer.net/, accessed on 21 April 2007.

[GStreamer 2007b] Object Oriented, http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/section-goals-design.html, accessed on 4 May 2007.

[GStreamer 2007c] Integration, http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/chapter-intgration.html, accessed on 4 May, 2007.

[GStreamer 2007d] GStreamer: Applications, http://gstreamer.freedesktop.org/apps/, accessed on 4 May 2007.

[GStreamer 2007e] GStreamer Application Development Manual, http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/chapter-intro.html#section-intro-what, accessed 12 June 2007.

[Hestand and Yanco 2004] Dan Hestand and Holly A. Yanco. "Layered Sensor Modalities for Improved Feature Detection, *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, October 2004.

[IM 2007] "IM: Image Representation, Storage, Capture and Processing," http://www.tecgraf.puc-rio.br/im/, accessed on 21 April 2007.

[ImageLib 2007] "ImageLib: An Image Processing C++ Class Library," http://www.dip.ee.uct.ac.za/~brendt/srcdist/, accessed on 21 April 2007.

[ImageMagick 2007] ImageMagick, http://www.imagemagick.org/script/index.php, accessed on 20 April 2007.

[ISO/IEC 2005] ISO/IEC 9899:TC2, WG14/N1124, 6 May 2005. http://www.openstd.org/JTC1/SC22/WG14/www/docs/n1124.pdf, accessed on 25 April 2007.

[IVT 2007] Integrating Vision Toolkit, http://ivt.sourceforge.net/, accessed on 18 April 2007

[JMF 2007] Java Media Framework, http://java.sun.com/products/java-media/jmf/, accessed on 18 April 2007.

[JPEG 2007] International JPEG Group, www.ijg.org, accessed on 25 April, 2007.

[Kameda 1997] Yoshinari Kameda. "Double-Difference image," http://www.kameda-lab.org/research/publication/1996/vsmm/html/node2.html, April 1997, accessed on 23 April 2007.

[libvideogfx 2007] LibVideoGfx, http://vca.ele.tue.nl/farin/software/libvideogfx/index.html, accessed on 21 April 2007.

[Lindblad and Tennenhouse 1996] C. J. Lindblad, and D. L. Tennenhouse. "The VuSystem: A Programming System for Compute-Intensive Multimedia," *IEEE Journal of Selected Areas in Communications,* September 1996.

[Lindblad et al. 1994] C. J. Lindblad, D. J. Wetherall, and D. L. Tennenhouse. "The VuSystem: A Programming System for Visual Processing of Digital Video," *Proceedings of ACM Multimedia 94*, San Francisco, CA, October 1994.

[LiVES 2007] Linux is a Video Editing System, http://lives.sourceforge.net/, accessed on 22 April 2007.

[Lovell 2004] Nathan Lovell. "Real-Time Embedded Vision System Development using AIBO Vision Workshop 2," *Proceedings of the Fifth Mexican International Conference in Computer Science (ENC'04)*, 20-24 Sept. 2004.

[LTI-lib 2007] LTI-Lib, http://ltilib.sourceforge.net/doc/homepage/index.shtml, accessed on 21 April 2007.

[Manolescu and Nahrstedt 1998] Dragos-Anton Manolescu and Klara Nahrstedt. "A Scalable Approach to Continuous-Media Processing," *Proc.~8th International Workshop on Research Issues in Data Engineering*, Orlando, Fl, 23-24 February 1998.

[Martin and Chanler 2007] Fred G. Martin and Andrew Chanler. "Introducing the Blackfin Handy Board," Submitted for publication in the 2007 AAAI Spring Symposium, Standford, CA.

[MATLAB 2007] MATLAB, http://www.mathworks.com/products/matlab/, accessed on 21 April 2007.

[Mimas 2007] Mimas, http://vision.eng.shu.ac.uk/mediawiki/index.php/Mimas, accessed on 21 April 2007.

[Mines et al. 1994] R. Mines, J. Friesen, and C. Yang. “DAVE: A Plug and Play Model for Distributed Multimedia Application Development,” *Proceedings of the second ACM International Conference on Multimedia*, San Francisco, Ca, October.

[Mori et al. 1994] H. Mori, N. M. Charkari, T. Matsushita. “On-line vehicle and pedestrian detection based on sign pattern,” *IEEE Trans. on Industrial Electronics, Vol. 41, No. 4*, Aug. 1994.

[MSDN 2007] Processes and Threads, Microsoft Developer's Network, http://msdn2.microsoft.com/en-us/library/ms684841.aspx, accessed on 25 April 2007.

[Myron 2007] Myron (WebcamXtra), http://webcamxtra.sourceforge.net/, accessed on 21 April 2007.

[Nichols et al. 1996] Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farell. *Pthreads Programming*, O'Reilly, 1996.

[Octave 2007] Octave, http://www.gnu.org/software/octave/, accessed on 21 April 2007.

[OpenCV 2007a] Open Computer Vision Library Wiki, http://opencvlibrary.sourceforge.net/, accessed on 18 April 2007.

[OpenCV 2007b] HighGUI - Open Computer Vision Library Wiki, http://opencvlibrary.sourceforge.net/HighGui, accessed on 21 April 2007.

[OpenGL 2007] OpenGL Documentation, http://www.opengl.org/documentation/, accessed on 22 April 2007.

[Perkins 2003] Colin Perkins. *RTP - Audio and Video for the Internet*, Addison-Wesley, 2003.

[Phission.org 2007] Phission: Image Processing Filters, http://www.phission.org/d6/df0/group__Image__Filters.html, accessed on 15 June 2007.

[Pyrobot 2007] Python Robotics, http://pyrorobotics.org/?page=Pyro, accessed on 22 April 2007.

[Qt 2007] Qt Toolkit from Trolltech, http://www.trolltech.com/products/qt, accessed on 22 April, 2007.

[Reference 2007a] Definition of dataflow, http://www.reference.com/search?r=13&q=Dataflow, accessed on 16 April 2007.

[Roseman and Greenburd 1992] Mark Roseman and Saul Greenburg. “GROUPKIT: A Groupware Toolkit for Building Real-Time Conferencing Applications,” *CSCW 92 Proceedings*, November.

[Russell and Norvig 2003] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, Second Edition, Prentice-Hall, 2003.

[S2iLib 2007] S2iLib, http://s2i.das.ufsc.br/tikiwiki/tiki-index.php?page=S2iLibSourceForge, accessed on 21 April 2007.

[SciLab 2007] SciLab, http://www.scilab.org/, accessed on 21 April 2007.

[Schulzrinne and Casner 2003] H. Schulzrinne and S. Casner. “RTP Profile for Audio and Video Conferences with Minimal Control (RFC 3551),” http://tools.ietf.org/html/rfc3551, accessed on 25 April 2007.

[Schulzrinne et al. 2003] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson. “RTP: A Transport Protocol for Real-Time Applications (RFC 3550),” http://tools.ietf.org/html/rfc3550, accessed on 25 April 2007.

[Singhal and Shivaratri 1994] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems,* McGraw-Hill, 1994.

[SWIG 2007] Software Interface Generator (SWIG), http://www.swig.org/, accessed on 18 April 2007.

[Unicap 2007] unicap, http://unicap-imaging.org/, accessed on 22 April 2007.

[Veejay 2007] Veejay, http://veejay.dyne.org/, accessed on 21 April 2007.

[VIAGRA 2007] “Vision with Generic Algorithms,” http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/, accessed on 20 April 2007.

[VIPS 2007] Vips Wiki, http://www.vips.ecs.soton.ac.uk/index.php?title=VIPS, accessed on 21 April 2007.

[VisualDSP++ 2007] VisualDSP++ for the Blackfin Processor, http://www.analog.com/en/epHSProd/0,2542,VISUALDSPBF,00.html, accessed on 23 April 2007.

[VuSystem 2007] VuSystem distribution directory, ftp://ftp.tns.lcs.mit.edu/pub/vs/, accessed on 21 April 2007.

[Wikipedia 2007a] Definition of dataflow, http://en.wikipedia.org/wiki/Dataflow, accessed on 16 April 2007.

[Wikipedia 2007b] Definition of extensibility, http://en.wikipedia.org/wiki/Extensible, accessed on 16 April 2007.

[Wikipedia 2007c] Definition of modularity, http://en.wikipedia.org/wiki/Modularity_%28programming%29, access on 17 April 2007.

[Wikipedia 2007d] Definition of "UNIX-like," http://en.wikipedia.org/wiki/Unix-like, accessed on 22 April 2007.

[Wikipedia 2007e] HSV color space, http://en.wikipedia.org/wiki/HSV_color_space, accessed on 22 April 2007.

[Wikipedia 2007f] Quicksort algorithm, http://en.wikipedia.org/wiki/Quicksort, accessed 25 April 2007.

[Wikipedia 2007g] Berkeley sockets, http://en.wikipedia.org/wiki/Berkeley_sockets, accessed on 25 April 2007.

[Wikipedia 2007h] POSIX Threads, http://en.wikipedia.org/wiki/POSIX_Threads, accessed on 25 April 2007.

[Wikipedia 2007i] Real-time Transport Protocol, http://en.wikipedia.org/wiki/Real-time_Transport_Protocol, accessed on 25 April 2007.

[Wikipedia 2007j] Definition of Nearest Neighbor Interpolation, http://en.wikipedia.org/wiki/Nearest_neighbor_interpolation, accessed on 3 May 2007.

[Wikipedia 2007k] Definition of Bilinear Filtering, http://en.wikipedia.org/wiki/Bilinear_filtering, accessed on 3 May 2007.

[Wikipedia 2007l] Design Pattern (computer science), http://en.wikipedia.org/wiki/Design_pattern_(computer_science), accessed on 13 June 2007.

[Wikipedia 2007m] FFmpeg, http://en.wikipedia.org/wiki/FFmpeg, accessed on 15 June 2007.

[UML 2007] UMass Lowell Robotics Lab, http://www.cs.uml.edu/robots/, accessed on 4 May 2007.

[Vaughn et al. 2007] Gary V. Vaughan, Ben Elliston, Tom Tromey and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool,* http://sourceware.org/autobook/

[V4L 2007] Video4Linux, http://linux.bytesex.org/v4l2/, accessed on 15 June 2007.

[VDK 2006] VisualDSP++ 2.5 Kernel (VDK) User's Guide, Analog Devices, April 2006.

[VFW 2007] VideoForWindows, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_video_for_windows.asp, accessed on 15 June 2007.

[Yanco et al. 2005] Holly A. Yanco, Michael Baker, Robert Casey, Andrew Chanler, Munjal Desai, Dan Hestand, Brenden Keyes, and Philip Thoren. “Improving Human-Robot Interaction for Remote Robot Operation,” *Robot Competition and Exhibition Abstract, Proceedings of the National Conference on Artificial Intelligence (AAAI-05)*, July 2005.

[Zlib 2007] Zlib compression library, www.zlib.org, accessed on 25 April 2007.

A.1 SegmentationExample

A.1.2 Directory Listing

```
.:
include  Makefile  Makefile.example  src

./include:
SegmentationExample.h

./src:
SegmentationExample.cpp
```

A.1.3 src/SegmentationExample.cpp

```
/*
 *  Phission :
 *      Realtime Vision Processing System
 *
 *  Copyright (C) 2007 Philip D.S. Thoren
 *      (pthoren@cs.uml.edu)
 *  University of Massachusetts at Lowell,
 *  Robotics Lab
 *
 */
#include <SegmentationExample.h>
#include <phission.h>

int main( int argc, char *argv[] )
{
    /* Declare the function and rc variables:
     *    #define phFUNCTION(name) \
     *            const char *function = name; \
     *            int rc = 0; \
     * phFUNCTION is simply a convenience macro.
     * The “function” variable is required for other
     * Phission macros that print.
     */
    phFUNCTION("main")

    /* Declare the objects */
    phImageCapture      *capture        = NULL;
    phPipeline          *pipeline       = NULL;
    blob_Filter         *segment        = NULL;
```

```
gaussian3x3_Filter  *gauss          = NULL;
medianNxN_Filter    *median         = NULL;
meanNxN_Filter      *mean           = NULL;
phSystem            *vision_system  = NULL;
phDisplayInterface  *disp_pipeline  = NULL;
phDisplayInterface  *disp_capture   = NULL;

phColor object_color = phColorRGB24_new(230,50,55);
phColor threshold    = phColorRGBA32_new(55,25,35,0);
phColor output_color = phColorRGB24_new(0,0,0);
phBlobData blob_data;
const int32_t blob_min_size = 100;

/* Allocate: all of the vision system objects. */
capture         = new V4LCapture();
pipeline        = new phPipeline();
segment         = new blob_Filter();
gauss           = new gaussian3x3_Filter();
median          = new medianNxN_Filter(3);
mean            = new meanNxN_Filter(3);
disp_pipeline   = new X11Display(320,240,"Pipeline");
disp_capture    = new X11Display(320,240,"Capture");
vision_system   = new phSystem();

/* Setup: adjust parameters of capture devices,
 *        filters, etc.
 */
capture->set(320,240,"/dev/video0");
capture->setChannel(0);
capture->setColour(35000);
capture->setHue(22700);
capture->setContrast(32000);
capture->setBrightness(28000);

segment->setColor(object_color,threshold);
segment->setOutcolor(output_color);
segment->setDrawRects(1);
segment->setColorBlobs(1);
segment->setColorMinSize(blob_min_size);

/* Processing: Add all of the filters to the
 *             pipeline.
 */
pipeline->add(gauss);
pipeline->add(segment);

/* Management: Add all of the threaded objects
```

```
 *             to the system.
 */
vision_system->add(capture);
vision_system->add(pipeline);
vision_system->add(disp_capture);
vision_system->add(disp_pipeline);

/* Connect: Connect all of the objects to create the
 *          vision system.
 */
pipeline->setInput(capture->getOutput());
disp_pipeline->setInput(pipeline->getOutput());
disp_capture->setInput(capture->getOutput());
blob_data.connect(segment->getLiveBlobOutput());

/* Start the vision processing system. */
vision_system->startup();

/* Loop until all of the displays are closed.*/
while (vision_system->displaysOpen() > 0)
{
    /* Get the most recent blob data */
    rc = blob_data.update();
    if ((rc == phLiveObjectUPDATED) &&
        (blob_data.getTotalBlobs(blob_min_size)))
    {
        blob_data.print_data(blob_min_size);
    }

    /* Yielding is optional. This gives up the
     * thread's timeslice to prevent slow response
     * in other threads. It consumes more CPU
     * cycles than sleeping. Use it instead of
     * sleeping if this loop is processing anything.
     */
    phYield();
}

/* Shutdown the vision processing system. */
vision_system->shutdown();

/* This is not necessary, but shows that the object
 * can be disconnected manually. It could then be
 * connected to a different segmentation/blob filter
 * if necessary.
 */
blob_data.disconnect();
```

```
    phDelete(disp_pipeline);
    phDelete(disp_capture);
    phDelete(gauss);
    phDelete(median);
    phDelete(mean);
    phDelete(segment);
    phDelete(pipeline);
    phDelete(capture);
    phDelete(vision_system);

    return phSUCCESS;
}
```

A.1.4 Makefile

The contents of Makefile.example can be found in the Phission package directory examples/cpp/ which is also visible online at: http://phission.cvs.sourceforge.net/phission/phission/examples/cpp/Makefile.example?view=markup

```
#
# Makefile : Modified by Philip D.S. Thoren
(pthoren@cs.uml.edu)
#
topdir := .

BINARY=SegmentationExample

include ${topdir}/Makefile.example
```

A.1.5 include/SegmentationExample.h

```
#ifndef SEGMENTATIONEXAMPLE_H
#define SEGMENTATIONEXAMPLE_H
#endif
```