Report Number: WUCSE-2004-29

2004-05-01

Fusion and Perspective Correction of Multiple Networked Video Sensors

Christopher E. Neely and John W. Lookwood

A network of adaptive processing elements has been developed that transforms and fuses video captured from multiple sensors. Unlike systems that rely on end-systems to process data, this system distributes the computation throughout the network in order to reduce overall network bandwidth. The network architecture is scalable because it uses a hierarchy of processing engines to perform signal processing. Nodes within the network can be dynamically reprogrammed in order to compose video from multiple sources, digitally transform camera perspectives, and adapt the video format to meet the needs of specific applications. A prototype has been developed using reconfigurable hardware... **Read complete abstract on page 2.**

Fusion and Perspective Correction of Multiple Networked Video Sensors

Christopher E. Neely and John W. Lookwood

Complete Abstract:

A network of adaptive processing elements has been developed that transforms and fuses video captured from multiple sensors. Unlike systems that rely on end-systems to process data, this system distributes the computation throughout the network in order to reduce overall network bandwidth. The network architecture is scalable because it uses a hierarchy of processing engines to perform signal processing. Nodes within the network can be dynamically reprogrammed in order to compose video from multiple sources, digitally transform camera perspectives, and adapt the video format to meet the needs of specific applications. A prototype has been developed using reconfigurable hardware that collects and processes real-time, streaming video of an urban environment. Multiple video cameras gather data from different perspectives and fuse that data into a unified, top-down view. The hardware exploits both the spatial and temporal parallelism of the video streams and the regular processing when applying the transforms. Recon-figurable hardware allows for the functions at nodes to be reprogrammed for dynamic changes in topology. Hardware-based video processors also consume less power than high frequency software-based solutions. Performance and scalability are compared to a distributed software-based implementation. The reconfigurable hardware design is coded in VHDL and prototyped using Washington University's Field Programmable Port Extender (FPX) platform. The transform engine circuit utilizes approximately 34 percent of the resources of a Xilinx Virtex 2000E FPGA, and can be clocked at frequencies up to 48 MHz. The com-position engine circuit utilizes approximately 39 percent of the resources of a Xilinx Virtex 2000E FPGA, and can be clocked at frequencies up to 45 MHz.

2004-29

Fusion and Perspective Correction of Multiple Networked Video Sensors, Master's Project Report, May 2004

Authors: Neely, Chris E.

May 11, 2004

Abstract: A network of adaptive processing elements has been developed that transforms and fuses video captured from multiple sensors. Unlike systems that rely on endsystems to process data, this system distributes the computation throughout the network in order to reduce overall network bandwidth. The network architecture is scalable because it uses a hierarchy of processing engines to perform signal processing. Nodes within the network can be dynamically reprogrammed in order to compose video from multiple sources, digitally transform camera perspectives, and adapt the video format to meet the needs of specific applications.

A prototype has been developed using reconfigurable hardware that collects and processes real-time, streaming video of an urban environment. Multiple video cameras gather data from di erent perspectives and fuse that data into a unified, top-down view. The hardware exploits both the spatial and temporal parallelism of the video streams and the regular processing when applying the transforms. Reconfigurable hardware allows for the functions at nodes to be reprogrammed for dynamic changes in topology. Hardware-based video processors also consume less power than high frequency software-based solutions. Performance and scalability are compared to a distributed software-based implementation.

The reconfigurable hardware design is coded in VHDL and prototyped using Washington University™s Field

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FUSION AND PERSPECTIVE CORRECTION OF MULTIPLE NETWORKED VIDEO SENSORS

by

Christopher E. Neely

Prepared under the direction of Professor John W. Lockwood

A project report presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ABSTRACT

FUSION AND PERSPECTIVE CORRECTION OF MULTIPLE NETWORKED VIDEO SENSORS

by Christopher E. Neely

ADVISOR: Professor John W. Lockwood

May, 2004
Saint Louis, Missouri

A network of adaptive processing elements has been developed that transforms and fuses video captured from multiple sensors. Unlike systems that rely on end-systems to process data, this system distributes the computation throughout the network in order to reduce overall network bandwidth. The network architecture is scalable because it uses a hierarchy of processing engines to perform signal processing. Nodes within the network can be dynamically reprogrammed in order to compose video from multiple sources, digitally transform camera perspectives, and adapt the video format to meet the needs of specific applications.

A prototype has been developed using reconfigurable hardware that collects and processes real-time, streaming video of an urban environment. Multiple video cameras gather data from different perspectives and fuse that data into a unified, top-down view. The hardware exploits both the spatial and temporal parallelism of the video streams and the regular processing when applying the transforms. Reconfigurable hardware allows for the functions at nodes to be reprogrammed for dynamic changes in topology. Hardware-based video processors also consume less power than high frequency software-based solutions. Performance and scalability are compared to a distributed software-based implementation.

The reconfigurable hardware design is coded in VHDL and prototyped using Washington University's Field Programmable Port Extender (FPX) platform. The transform engine circuit utilizes approximately 34 percent of the resources of a Xilinx Virtex 2000E FPGA, and can be clocked at frequencies up to 48 MHz. The composition engine circuit utilizes approximately 39 percent of the resources of a Xilinx Virtex 2000E FPGA, and can be clocked at frequencies up to 45 MHz.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Future sensor networks can take advantage of reconfigurable hardware to perform local processing in ways that save power, consume less network bandwidth, and enable advanced video applications. For applications like distributed video surveillance, data fusion can be performed to filter redundancy and reduce the number of transmissions. It is difficult however to perform this function using the microprocessors found on today's sensor networks because they lack the computational capability to process full frame rate video. Reconfigurable hardware can enable computations to be performed in real-time, at lower clock frequencies, using less power. This project report presents an architectural framework for transforming and fusing video captured from many sensors and discusses some advantages of using reconfigurable hardware.

The functions performed by sensor network nodes are evolving from simply passing sensor data to include local processing of data. Early sensor network architectures concentrated data transmission to centralized nodes that provided bulks of computational resources [4]. More emphasis is now placed on distributed architectures because centralized architectures have not scaled well for networks containing hundreds or thousands of nodes, as those centralized nodes quickly became the bottleneck links. While much sensor network research is currently exploring ways to optimize ad-hoc routing, and minimize power consumption of sensor networks, another related area of research is to determine reasonable ways to perform data fusion within the sensor network. The goal of this project is to design and implement a scalable system that fuses multiple video streams, using distributed processing in the network.

1.1.1 Video Data Fusion

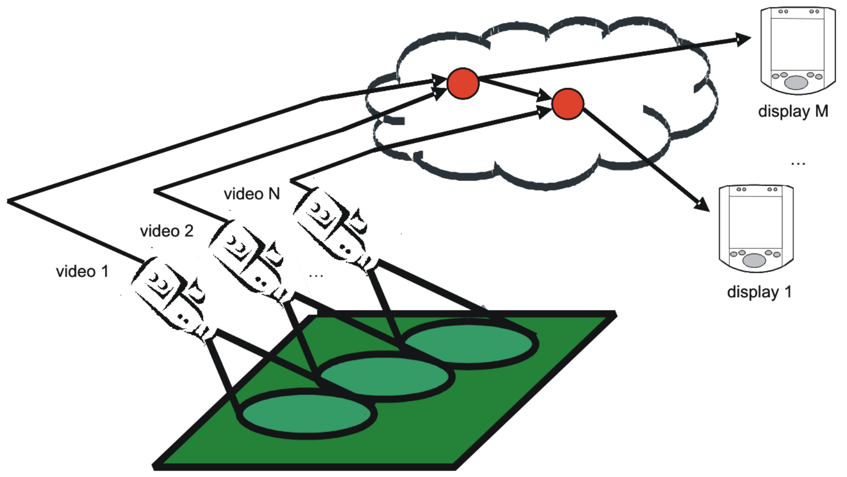


Figure 1.1: Video sensor network performs data fusion for surveillance application.

A video surveillance application might monitor road conditions for an entire city. Video sensors would be positioned above traffic intersections, roads, and highways. The sensing radius of each camera would overlap to include complete coverage– in an ideal deployment they would be evenly spaced. There could be large numbers of displays accessing the network, perhaps one display for every vehicle. Displays in close proximity might need to access the same overhead view; others might need larger views of the city. Fusing video streams creates views that are not restricted to single camera positions and also allows for the view to include larger scope from multiple cameras, as illustrated in Figure 1.1. Any magnification of any location in the environment could be accessible to any display.

The application described here is nontraditional for discussion as a sensor network because video is expensive in both communication bandwidth (1-10Mbps compressed, 100Mbs uncompressed) and power consumption, which defeats plans for implementing this application using conventional sensor network platforms that are energy-constrained. Most sensor networks that are described in literature have very explicit design constraints for low power, limited bandwidth, and limited computational resources in order to be categorized as a sensor network [4].

The advantages of performing data fusion in the network are many: reduced network bandwidth, energy savings for wireless transmission, increased scalability-because the overhead per node becomes distributed, and generation of higher-level information[4]. Some of the disadvantages include: increased computational requirements placed on every node, the configuration must either be calibrated or coordinated, and sometimes there is added latency. From the Wireless Integrated Network Sensors (WINS) project Pottie and Kaiser write, *"if the application and infrastructure permit it pays to process the data locally to reduce traffic volume and make use of multihop routing and advanced communication techniques to reduce energy costs"* [15]. As the processing capabilities at nodes improve, computational-bound latencies should become less of an issue.

1.1.2 Reconfigurable Hardware Advantages

Distributed data fusion on video requires that each node have a reasonably fast processor that is capable of processing the high bit rate video streams. Application-specific integrated circuits (ASICs) can be designed to run at slower clock speeds, use less precision, and therefore can consume several orders of magnitude less energy than digital signal processors (DSPs) [15]. However ASICs are somewhat inflexible in terms of programmability and have much higher costs compared to DSPs. Field-programmable gate arrays (FPGAs) offer a viable compromise in terms of performance and cost. FPGAs offer a reprogrammable fabric containing millions of gate equivalents and in some cases integrated microprocessors. Like ASICs, FPGAs can be used to implement circuits that run at lower clock speeds using custom precision, and the functions can be later reprogrammed like a processor.

1.1.3 FPX Platform

Washington University's Field-programmable Port Extender (FPX) platform is an advanced platform for prototyping network applications using FPGA reconfigurable hardware [8]. The FPX, shown in Figure 1.2, contains a large reprogrammable application device (RAD), implemented using a Xilinx VirtexE 2000E, and fast gigabit network interfaces (2.4Gbps). The RAD supports applications to perform processing on both the ingress and egress data paths[9]. There is a set of Layered Protocol Wrappers that has been developed to provide abstract interfaces for Internet protocol (IP) packets [1][16]. Although present day Internet protocols were not designed with the need to conserve energy [15], IP is a very commonly used protocol and IP hardware is both low cost and widely available. More details about the FPX device and configuration are presented in Appendix A.

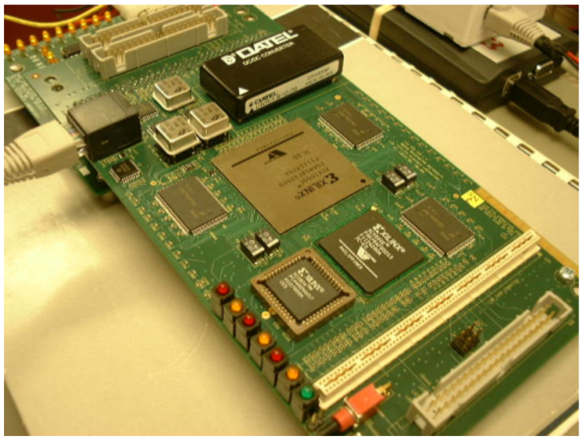


Figure 1.2: Field-programmable port extender device.

1.1.4 Java Media Framework

Since it can take a long time to specify a full-featured design at the register transfer level, a Java prototype was first implemented using the Java Media Framework. The Java software language provides libraries that can enable rapid application prototyping and development. The Java Media Framework API provides interfaces for capturing, processing, transporting, and presenting time-sensitive multimedia streams, see Figure 1.3 [6]. This includes applying special effects, transcoding media formats, and transmitting multimedia streams using the real-time protocol (RTP) over universal datagram protocol (UDP) packetizers.

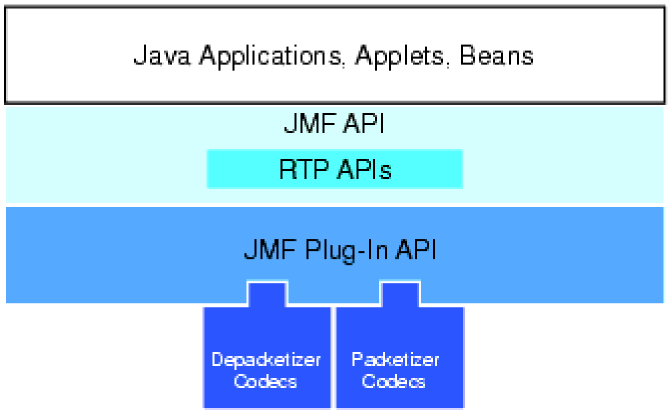


Figure 1.3: JMF API layers

1.2 Contributions

An architecture for performing data fusion on surveillance video is presented. Two versions of a video fusion project have been and implemented and are compared—hardware modules that run on the FPX platform and software applications that execute on top of the Java Media Framework. This video surveillance application demonstrates distributed, cooperative processing of sensor data by nodes throughout the network, which can be efficient for large numbers of displays. The processing that is performed in the network allows for computational resources at end systems to be conserved to support higher-level tasks. Configuration, tools, and methodology, are also described in this report.

1.3 Related work

Related research has explored: frameworks for data fusion applications[7], topologies and hierarchies for sensor networks[21], and sensor networks used for motion tracking using background subtraction from video[13]. Nguyen [13] implemented the motion tracking algorithms in software at camera nodes but anticipated better performance using FPGAs. Pless et. al [14] solved for homography transforms to track aerial camera movement.

Kumar [7] presented a similar architecture for general, distributed data fusion called DFuse. DFuse is a multi-threaded runtime system targeted for deployment in heterogeneous ad hoc sensor network environments. This video project is similar to the video *collage* application they use to motivate their system architecture. Kumar measures transmission costs of module placement between iPAQ processing nodes.

1.4 Organization of Rest of Project Report

Chapter 2 describes an architecture for video fusion that could support large numbers of sensors and displays. Specifically, this chapter discusses: components of this architecture and a hierarchy for directed data fusion. Chapter 3 details the full prototype software implementation that uses the Java Media Framework, which includes the following applications: a simple video capture, `TransformEngine`, and `CompositionEngine`. Chapter 4 details the designs of the hardware modules for the FPX that implement the transform engine and the composition engine of this architecture. Chapter 5 describes performance evaluations and compares latency and throughput between the implementations. Chapter 6 concludes and suggests future work for extending and improving the design.

Chapter 2

Video Sensor Network Architecture

This chapter describes the architecture for performing data fusion between multiple video streams that is designed to support large numbers of sensors and displays[1]. In a realistic environment, video from multiple cameras requires some processing before it can be combined. It is unlikely to assume that large numbers of video sensors could be deployed without the need for alignment. So, image transforms can be used to passively align video sensors. After transformation, multiple videos will fit roughly in the same coordinate system and the best view of overlapping regions can be selected for display.

The approach described above was used to partition the application requirements into subtasks so the solution could be implemented as separate modular components. The first subtask is to transform the views from each cameras, so that views are roughly the same scale and of the same viewing plane. The second subtask is to align and stitch multiple transformed views together as one ”seamless” view. This

[1]This architecture was co-developed with Christopher K. Zuver in the summer of 2003 as a project for Rockwell Collins

second subtask, the composition step, can be recursively repeated to build a hierarchy of views at various magnifications. Other requirements that were considered during the design process: the need for a flexible deployment and coordination between components; and also the desire to push processing into the network as a way to reduce data transmission.

2.1 Motivation for Performing Video Processing in the Network

After defining the goals for each subtask, the challenge becomes to determine where the processing for each subtask should take place. The task assignment throughout the network and at end systems has the potential to limit the performance of the system. For example, it might seem obvious to correct for problems like camera lens distortion at the camera, itself, because otherwise every display that accesses that camera would need to perform the same type of correction.

This form of reasoning can be extended to construct a larger argument. Assume that each display is given the flexibility to select from a set of cameras, and assume that the sensing task at-hand requires displays to cooperate on some task. Then, the selected subsets might have popular subsets of intersection. For those popular intersections, processing might be done at the end systems connected to the cameras to avoid redundant processing at every display.

Now assume that there are two major tasks at hand that are popular, but they require conflicting use of the same sensors. It would be practical to perform the processing before transmitting to the displays, but since they tasks depend on the same sensors it might make sense to add a “third party” nodes that perform the processing on behalf of the displays before the transmitting to large numbers of displays. When expanding this argument to a much larger network that has multiple popular applications that are needed concurrently, there exists a reasonable argument that processing could be applied to video throughout the network to efficiently utilize resources.

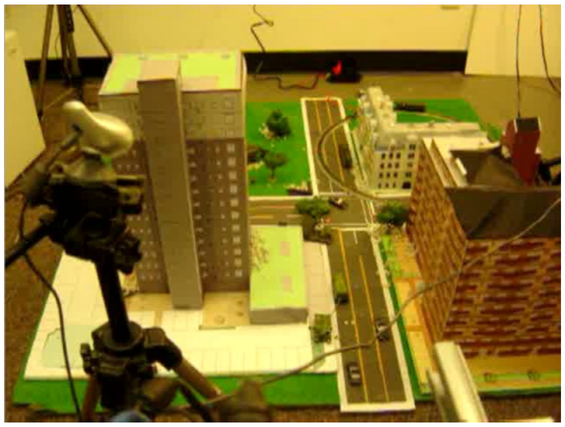


Figure 2.1: Video sensors arranged to monitor traffic conditions of an urban model

Sensor networks might monitor traffic conditions, and require overhead views as discussed in the previous chapter. Besides those overhead views, applications such as fire fighting might also need to share the sensors to access the views that show the sides of a certain building. Although one application might have higher priority, if processing is performed in the network, then the sensors can satisfy multiple applications.

2.2 Architecture

Our video sensor network can be modeled as a directed graph. At the edge of the graph are the cameras, which collect the source data. Within the graph are format engines, transform engines, and composition engines. Each of these components sinks one or more incoming video streams and outputs only one video stream, which can be multicast to groups.

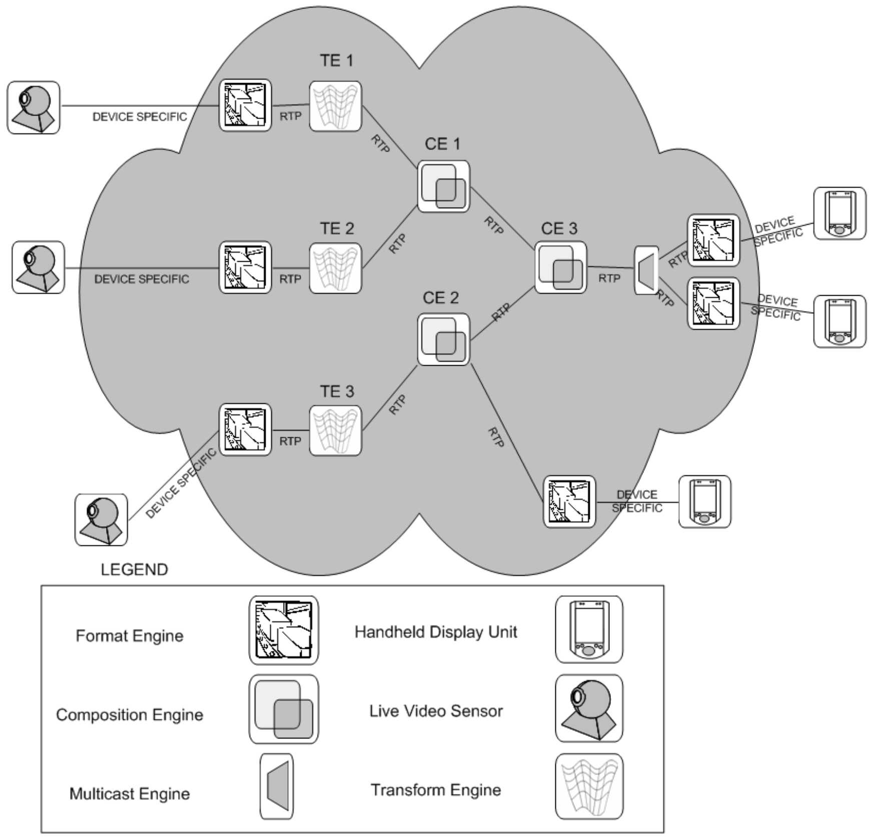


Figure 2.2: Logical components and connection diagram

A diagram of the data paths between sensors and displays for an example sensor configuration is shown in Figure 2.2. Data is collected from the cameras at the left, flows through the format, transform, composition, and multicast, format engines, and then is displayed on an end-system device.

In general, video is sent by the capture device to a format engine (FE). The format engine adapts the video data from the device specific format of the capture device to the network format. The FE directs the video stream to a transform engine (TE), where the video is transformed. The TE streams to a composition engine (CE). The CE outputs a fused video stream consisting of combination of stitched regions from the input streams. The fused stream from a CE is transmitted to either another CE or a display.

The control path for now is only concerned with programming the subtasks at each node. Logically there is one a subtask performed at every node. Task can be performed either in hardware or software. Nodes can be programmed based on popular demand to perform another subtask.

2.2.1 Transform Engine

Transform engines warp a video show to another selected view. Traditional methods for rectifying and aligning photographs were reviewed [5]. These methods included perspective correction based on vanishing points, radial correction, and scaling and rotation. The transform engines apply multiple transforms using a lookup table, which has a nice property that transforms can be combined in a way that there is no additional run-time penalty. Some transformed views are shown in Figure 2.3.

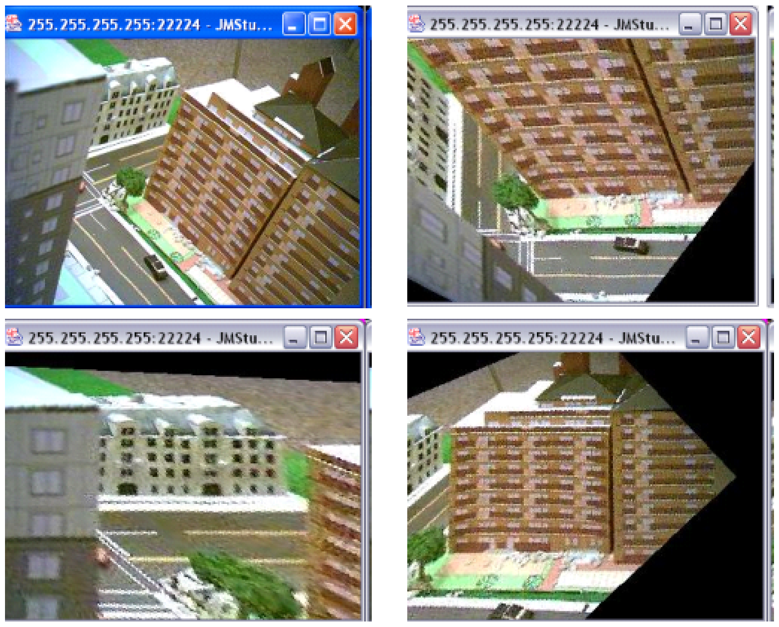

255.255.255.255:22224 - JMStu...
255.255.255.255:22224 - JMStu...
255.255.255.255:22224 - JMStu...
255.255.255.255:22224 - JMStu...

Figure 2.3: Example transformed views (clockwise): (a) original, (b) view of street, (c) view of rear building, (d) view of side building

2.2.2 Composition Engine

The top-down view will combine pixels from multiple cameras together into a common video frame, as shown in Figure 2.4. For the sake of simplicity, this application is currently only concerned with combining video views that show planar surfaces.

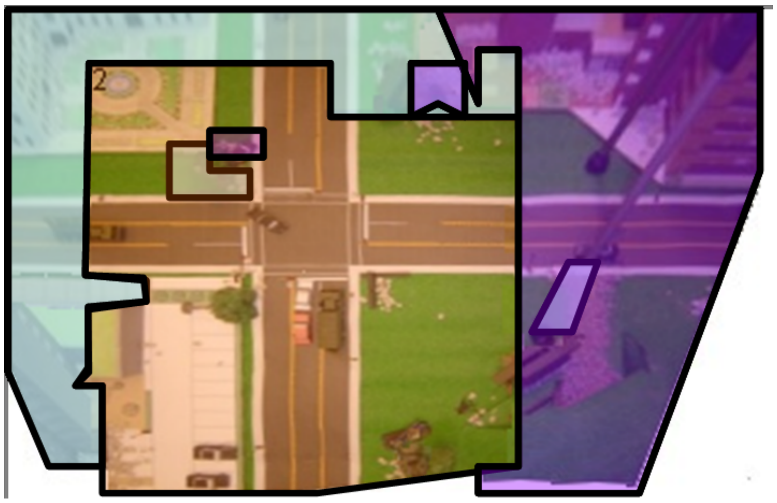


Figure 2.4: Example composite view formed from three cameras

2.3 Side discussions

In the architecture, video sensors could be addressed based on their position in terms of longitude and latitude coordinates. Hopefully close regions of overlap will exist between neighbors and sensors can be ranked in priority based on the value of their location and the quality of samples that they provide to the network applications. Avoiding redundant sampling at regions with overlap can reduce the number of transmissions and extend the life of the sensor network. Alternating periods of activity between close neighbors can also extend the life of the network. If one sensor is lost, then neighbors from overlapping regions could become candidates to replace those portions of the view.

In general, the architecture might be applied to other types of sensing devices such as thermal, vibrational, or acoustic sensors. In the case of video, there are other characteristics that might need transformation besides alignment. These might include need to correct for lens aberrations and/or changes in color and lighting.

Chapter 3

Software Implementation

This chapter describes the software implementation of the modular components of the video sensor network, which run on top of the Java Media Framework (JMF). JMF provides an extensible API for processing and presenting time-based media[12][6]. The JMF website provides documentation and sample code that was adapted to implement a simple video capture application, the transform engine application, and the composition engine application. The simple capture application is a reduced version of `AVTransmit.java`, which is one of the JMF guides for creating a `Processor`. `Processors` control the processing performed on an input media stream. The transform engine and composition engine applications extend the basic `Processor` using `Effect` plug-ins to perform custom processing on media streams.

3.1 Simple capture application

The simple capture application initializes a specified capture device as a `DataSource`, creates a JPEG/RTP transmitter as a `DataSink`, and installs a `Processor` in the intermediate data path. The purpose of the simple capture application is to stream

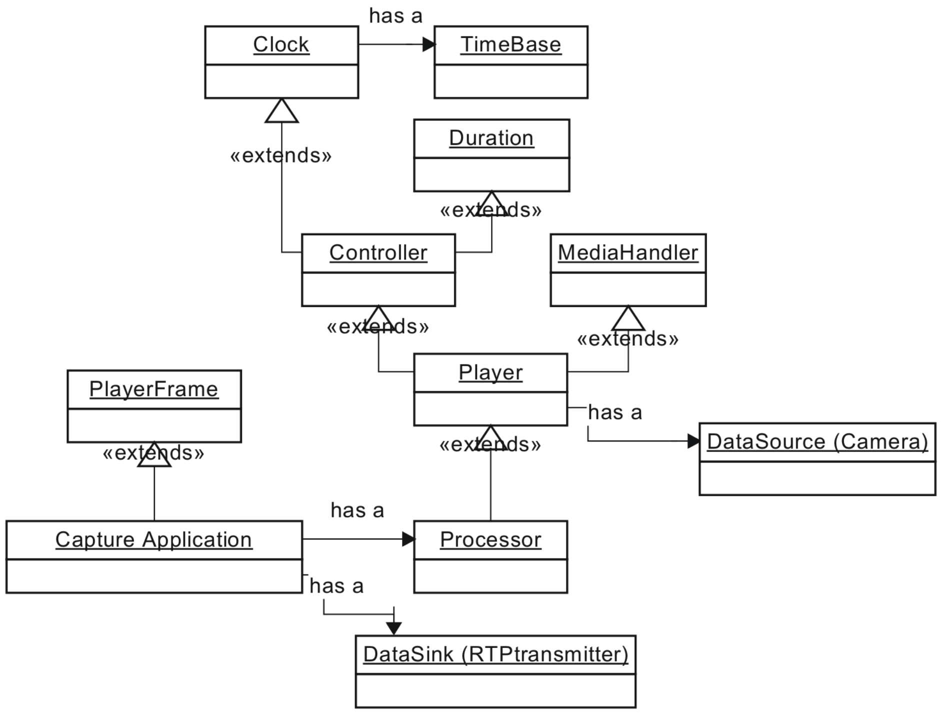


Figure 3.1: Class diagram for the simple capture application

raw video from the capture device to a transform engine, and so the `Processor` is passive for this particular application. The class hierarchy for the simple capture application, shown in Figure 3.1, shows the relation of the capture application class to other JMF classes. The `Processor` is actually a descendent of the `Player` that can modify the content of a media stream, with an input `DataSource` and output `DataSource`. Since `Players` and `Processors` extend from the `Controller` class, which manages some of the time-sensitive functions of the media streams, the capture application must implement the `ControlListener` interface to allow it to respond to control events.

The capture application, with class description shown in Figure 3.2:(a), accepts two command line arguments: an input `MediaLocator` and an output `MediaLocator`. The input `MediaLocator` is in the format of a video for windows (vfw) device driver reference, such as `vfw://0` or `vfw://1` (the number depends on the assigned number

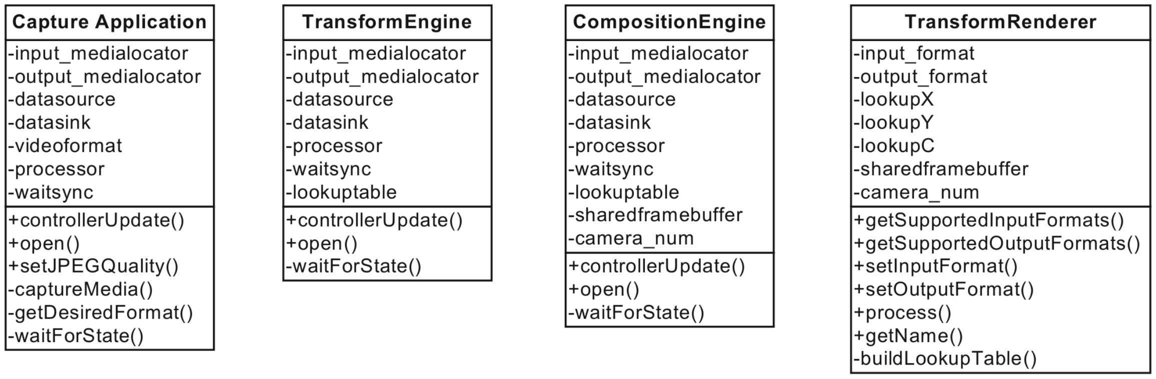


Figure 3.2: Class diagrams for applications: (a) simple capture application, (b) transform engine, (c) composition engine, (d) transform renderer

of the device driver). The output `MediaLocator` is in the format of an RTP URL address, such as `rtp://host_address:port`. Traffic on the specified output port will contain video and also bidirectional control information including the type of media, number of tracks, and connection status. JMF requires that the specified port number to be even.

The capture application sets up the media stream as described in this paragraph. The function `captureMedia()` supplies the input `MediaLocator` to the default JMF `Manager`, which then returns a handle to a new `DataSource` from the capture device. Next, the function creates and configures the `Processor`. The output format of the processor is set to JPEG/RTP with the default compression quality. An RTP transmitter is created to transmit to the address specified by the output `MediaLocator`. The RTP transmitter is connected to the output of the processor, and then the method realize() is called on the `Processor`. Once the `Processor` enters the `realized` state, the video stream begins and transmits to the specified URL of the transform engine.

3.2 Transform Engine

The code for the transform engine, for the most part, is similar to the simple capture application with the main exception being that the `Processor` actively processes the video stream using the `TransformRenderer` plug-in, see Figure 3.3 . The transform engine takes in three command line arguments: input and output `MediaLocators` and a lookup table filename. The `open()` function performs most of the same operations that the `captureMedia()` function did in the simple capture application. As part of the configuration step for the `Processor`, the `TransformRenderer` is specified as part of the codec chain. The lookup table filename is passed to the `TransformRenderer` plug-in as one of the class constructor parameters.

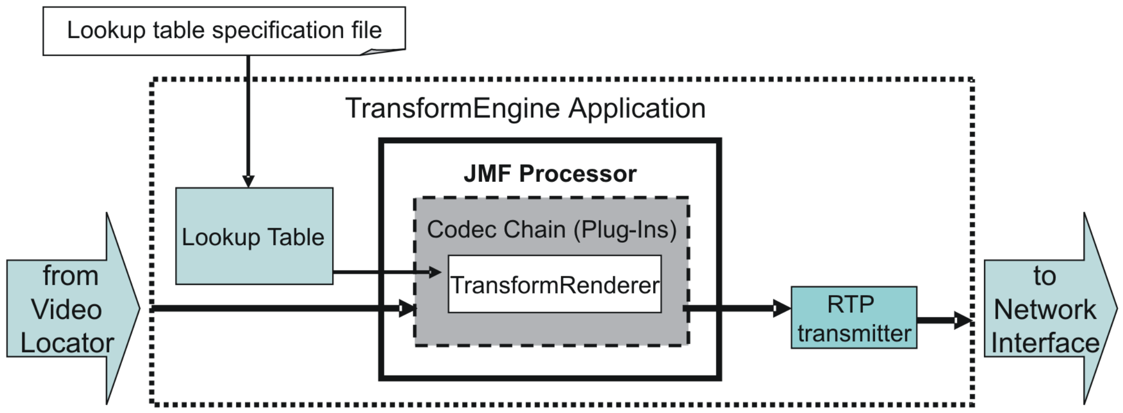


Figure 3.3: Data flow diagram for the transform engine application

The `TransformRenderer` plug-in begins by initializing the `lookupX`, `lookupY`, and `lookupC` (camera) arrays, which are 2D arrays of doubles, from the values stored in the lookup table file. The `TransformEngine` passes in the value 0 for the camera number. A separate application for generating the lookup table specification file for the transform engine is described in Appendix B. The code for the `TransformRenderer` mainly sits in a tight loop over the `process()` function. Pseudo-code for the `process()` is shown in Table 3.1.

Table 3.1: TransformRenderer::process(), for single camera function

```
–Called on every frame
output_index = 0;
for each y in lookupY do
  for each x in lookupX do
    ...
    input_index = width*y + x*pixel_size;
     – set R, G, B values for each pixel
    outputbuffer[output_index++] = inputbuffer[input_index++]
    outputbuffer[output_index++] = inputbuffer[input_index++]
    outputbuffer[output_index++] = inputbuffer[input_index++]
    . . .
  end for
end for
```

The transform described by the lookup table is applied to every frame of the live video. The screenshots in Figure 3.4 are from a JMF player connecting to the output RTP stream from the transform engine.

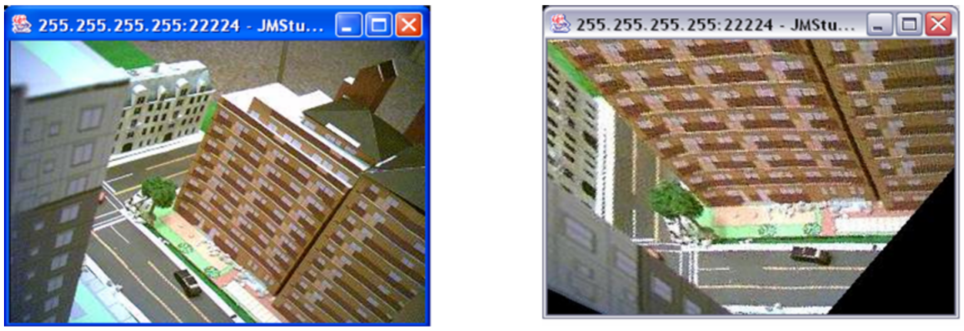

255.255.255.255:22224 - JMStu...
255.255.255.255:22224 - JMStu...

Figure 3.4: Sample screenshots: (a) original, (b) transformed

3.3 Composition Engine

The code for the `CompositionEngine` is likewise similar to the transform engine, but there is another level of added complexity involving multiple threads and a shared frame buffer. The composition engine begins by taking the following command line arguments: three (3) input `MediaLocators`, an output `MediaLocator`, and a lookup table filename. Three instances of the composition engine class are created as separate threads, one for each of the input `MediaLocators`, as shown in Figure 3.5. A camera number is also passed in as an argument with the `MediaLocators`, which is based on the order of the input arguments. This order needs to be consistent with the ordering in the lookup table specification. Details of the tool to generate the alignment lookup table appear in Appendix C. The `open()` method performs the same steps and initializes the `TransformRenderer`, similar to before, but now also passes a pointer to the shared frame buffer. Pseudo-code for the `TransformRenderer`, shown in Table 3.2, is provided below with added sections relevant to the multiple cameras of the composition engine.

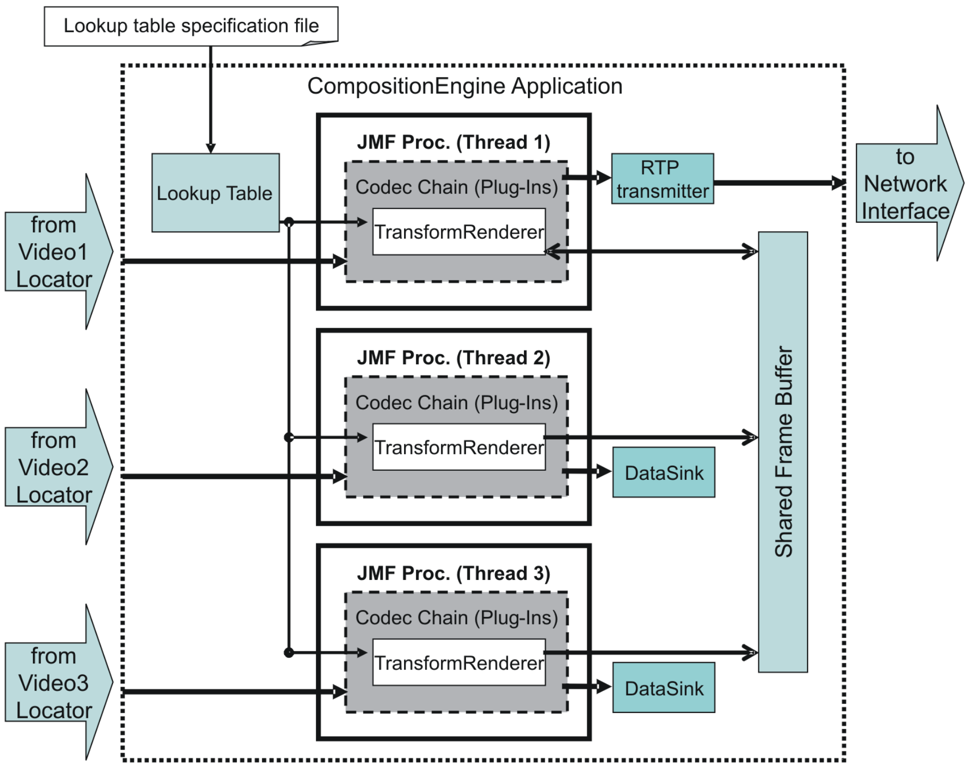


Figure 3.5: Data flow diagram for the composition engine application

Table 3.2: TransformRenderer::process(), for multiple cameras function

```
–Called on every frame
output_index = 0;
for each y in lookupY do
   for each x in lookupX do
      input_index = width*y + x*pixel_size;
      if c == 0 then
         –set R, G, B values for each pixel
         outputbuffer[output_index++] = inputbuffer[input_index++]
         outputbuffer[output_index++] = inputbuffer[input_index++]
         outputbuffer[output_index++] = inputbuffer[input_index++]
      else if lookupC ==c then
         –if not the main output stream,
         –then just copy to the shared frame buffer
         sharedframebuffer[output_index++] = inputbuffer[input_index++]
         sharedframebuffer[output_index++] = inputbuffer[input_index++]
         sharedframebuffer[output_index++] = inputbuffer[input_index++]
      else
         output_index += 3
      end if
   end for
end for
```

3.4 Remarks

There's the potential to perform synchronization between the multiple video streams based on the RTP timestamp. This has not yet been implemented, but it might be as simple as comparing the timestamp to a global timestamp variable in the process function. The global variable might be protected by a mutex and updated to the largest timestamp value received. Video regions would be copied into the shared frame buffer only if the current timestamp was greater or equal to the global value.

Chapter 4

Hardware Implementation

4.1 Transform Engine

4.1.1 Overview

The transform engine is a circuit that was designed to be tested in the reprogrammable application device (RAD) FPGA of the FPX platform, which has previously been programmed to perform string matching[17][3], IP lookup[20], video transcoding[11], and the functions of an Internet firewall[10]. This chapter discusses the two versions of the hardware transform engine that were implemented, one to process a single video stream and the other to process two video streams concurrently. The single video stream transform engine outputs pixels in the order specified by a transform lookup table. The dual video stream transform engine applies separate transform lookup tables to each video stream.

For this design, the video streams consist of uncompressed, 32-bit RGB video because it is simple to work with, but a system that needs to save on transmissions would probably employ video compression. Still-image compression could be applied to each video frame using Motion-JPEG (MJPEG) or MJPEG-2000 algorithms to provide an order of magnitude savings in the amount of data to be transmitted. An estimate is that the 70Mbps rate for uncompressed 320x240 video could be compressed to approximately 1-2 Mbps using MJPEG and still retain reasonable quality. Further, compression algorithms that account for motion prediction can improve upon this by another order of magnitude. The author was unaware of any freely available hardware cores to perform the video compression and decompression at the time of this design. For that reason and the additional design complexity and time that would have been needed to design and debug the circuit, a decision was made not to support MJPEG for this version of the hardware transform engine.

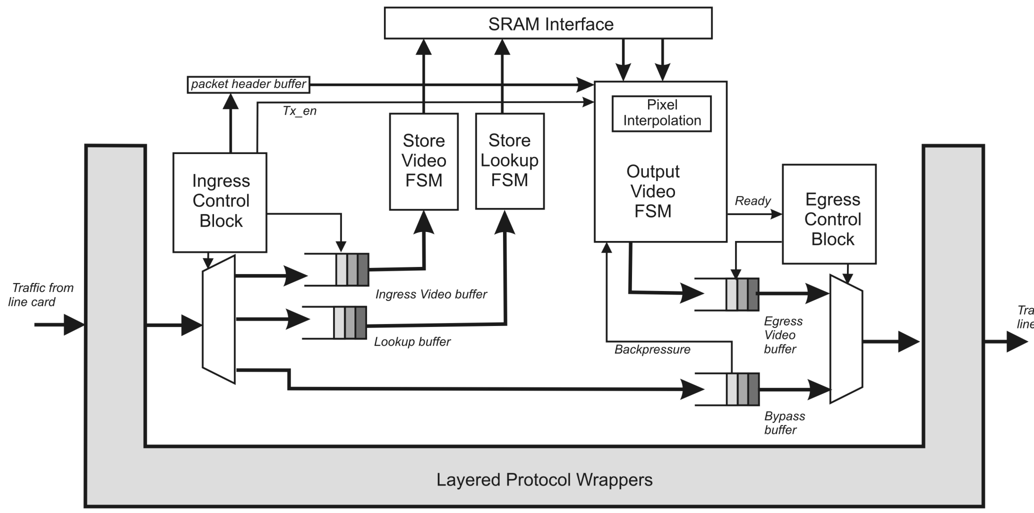


Figure 4.1: Block diagram of transform engine shows major components and data flow

A block diagram of the single video stream transform engine is shown in Figure 4.1. Traffic from the RAD line card interface enters into the Layered Protocol wrappers on the left. Only Internet protocol (IP) packets will pass through all the layers of the wrappers to the application inside; other types of traffic will be routed around the application by the wrappers. On the left inner side of the wrappers packets are demultiplexed by the ingress control block. Packets are classified as video data; lookup table entries; or bypass traffic, based on the destination port number contained in the UDP header, see Table 4.1. The ingress control block writes packets into the FIFO buffers according to type. The application includes three finite state machines (FSMs): store video, store lookup, and output video that process data contained in the packet payloads.

Table 4.1: Packet types used by the hardware modules

Packet type	**UDP Port**
Video stream 1	900
Video stream 2	901
Lookup table 1 entry	916
Lookup table 2 entry	917

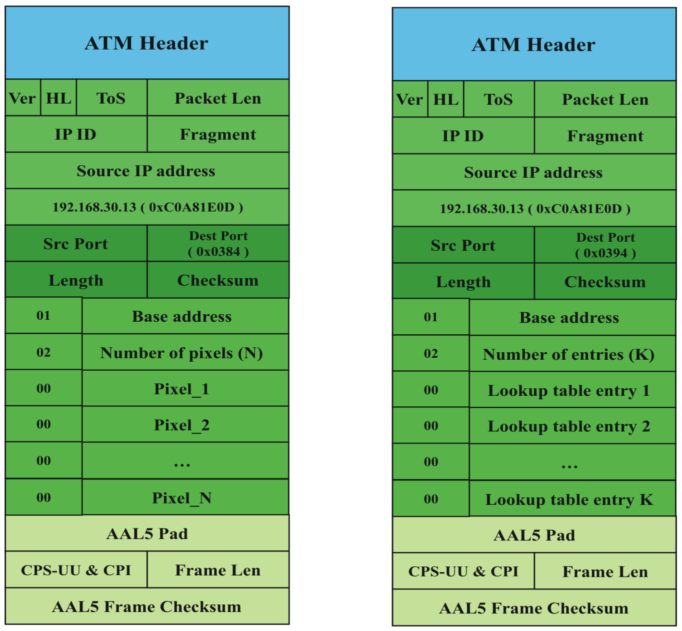


Figure 4.2: Internal packet formats, 32-bit words: (a) Video packet, (b) lookup table info packet

To allow for some design reuse, packets that contain video and packets that contain lookup table entries were formatted to be similar. The internal format of both video packets and lookup info packets are shown in Figure 4.2. For historic reasons the FPX uses asynchronous transfer mode (ATM) cell-based formats internally, and so these diagrams also show the ATM header and AAL5 trailer. The first four words, shaded in light green, constitute the IP header. The destination address is set to the IP address of the GigE line card. The next two words, shaded in darker green, make up the UDP portion of the header, which includes the field for the destination port number. Intentionally, the first two words of the payload have a prefix in the top byte (bits 31 downto 24) that distinguishes them as control words. The first word of the payload in both formats specifies a base address in SRAM, where the first data value should be written. The second word of the payload instructs the state machines how many values are contained in the control packet. The remaining portion of the payload contains either pixels or lookup table entries.

4.1.2 Store Video FSM

The store video FSM basically reads packets from video FIFO and writes the pixel values to SRAM, see Figures 4.3. Similarly, the store lookup FSM, which is another instantiation of the store video FSM, writes the lookup table entries to SRAM. Video data is stored in the lower address range (0x000000-0x01FFFF) of SRAM, and the lookup table is stored in the upper address range (0x200000-0x02FFFFF).

The operation of the store video FSM is fairly simple, see Figure 4.4. When the video buffer FIFO becomes non-empty, then store video requests control of SRAM. Store video reads the first two words of the packet payload containing the base address and number of pixels. Store video then transitions to the state `s_read_pix`, where it reads the remaining packet payload from the FIFO and outputs the pixel values on the `mod_d_out bus`. Meanwhile, store video performs a burst write to store those values in SRAM using a concatenation of row and column as the offset address. The upper bits of data are padded with zeros to fill the 36 bit width. The waveform

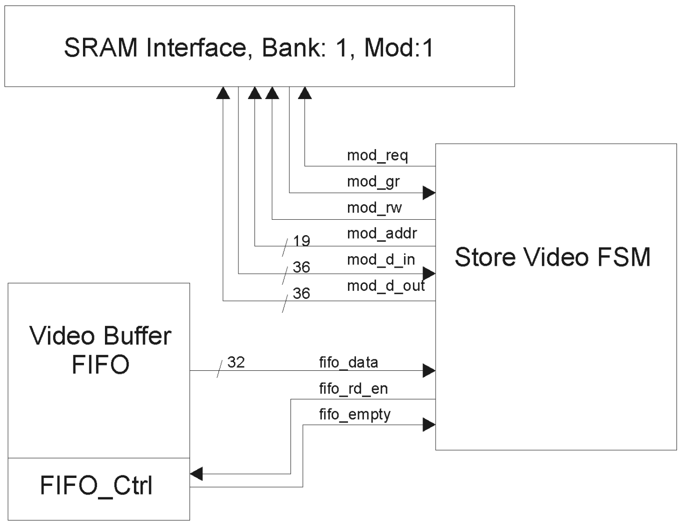


Figure 4.3: Block diagram illustrates store video FSM interfaces to memory components

in Figure 4.5 shows that after the FIFO became non-empty, the first two control words are processed by the state machine and the first value is written to the SRAM interface.

The dual transform engine circuit contains two instances of the store video circuit and consumes an additional memory for another frame buffer and lookup table. The second store video FSM also requires an added FIFO and the FSM interfaces to SRAM bank 2, module 1. There are four module interfaces on the extended SRAM controller interface. The second store lookup FSM interfaces to SRAM bank 2, module 3.

4.1.3 Output Video FSM

The output video FSM, or read video FSM, reads the lookup table and video frame buffer and then outputs full frames of video at an evenly paced rate, shown in Figure 4.6. At the beginning of each frame, the output video FSM steps through the list of addresses stored in the transform lookup table. For each address, it reads the

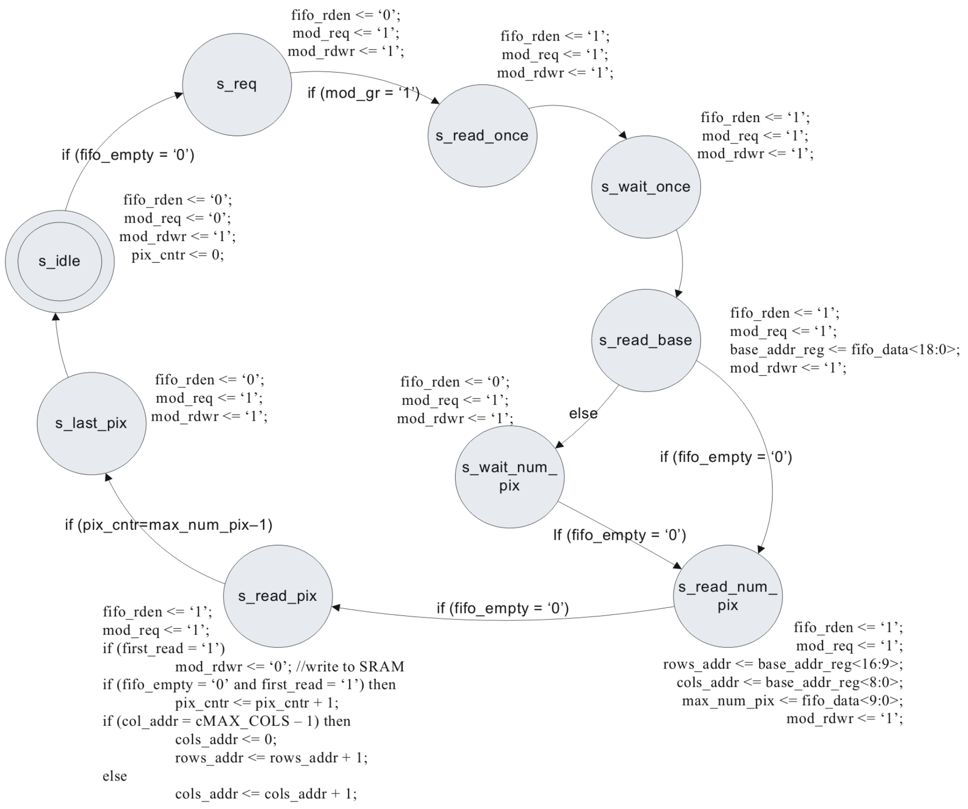
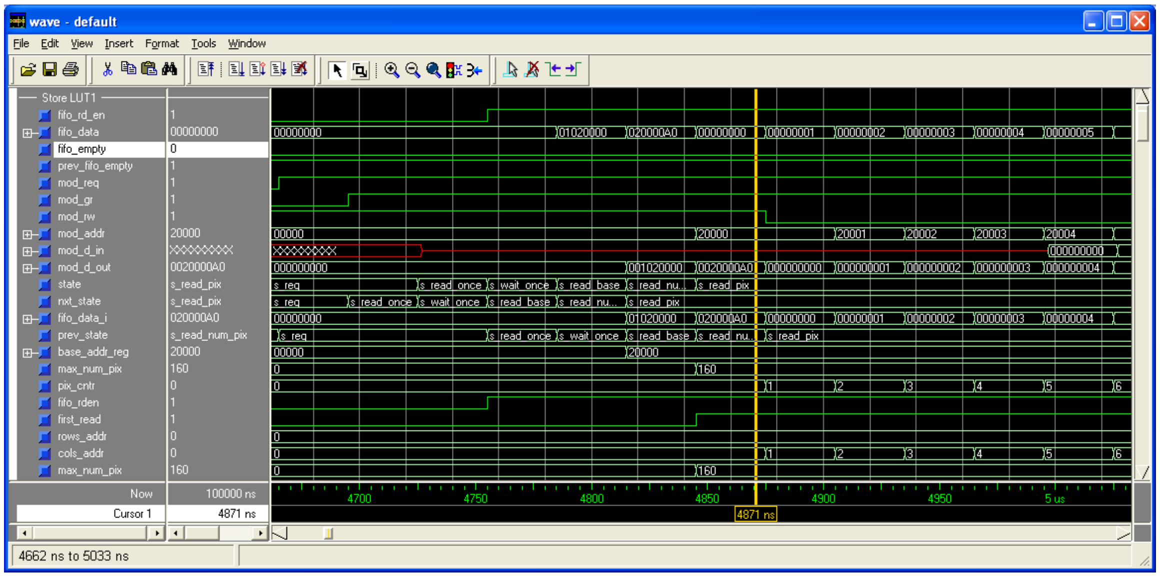


Figure 4.4: Abbreviated state machine for store video

corresponding pixel and writes it to the egress video buffer FIFO. It was decided to use separate request interfaces to memory for the lookup table and video frame, to allow for a future implementation in which the lookup table would be stored in SDRAM. The available SRAM controller infrastructure was extended from 2 interfaces to now support 4 interfaces for this design.

The output video state machine is also simple, it basically waits for a timer and then performs scores of memory reads, see Figure 4.7. The frame_timer is active whenever there is a `transmit_enable` signal and no TCA backpressure, shown in Figure 4.8. The timer counts to some maximum value, currently a user-defined constant in the VHDL code, and then pulses the frame timer signal, see cursor 1 in 4.9. Once this happens, the state machines transitions *idle* to reset internal counters and request SRAM in order to read the lookup table, see cursor 2 in Figure 4.9. The



Store LUT1	
fifo_rd_en	1
fifo_data	00000000
fifo_empty	0
prev_fifo_empty	1
mod_req	1
mod_gr	1
mod_rw	1
mod_addr	20000
mod_d_in	XXXXXXXX
mod_d_out	0020000A0
state	s_read_pix
nxt_state	s_read_pix
fifo_data_i	020000A0
prev_state	s_read_num_pix
base_addr_reg	20000
max_num_pix	160
pix_cntr	0
fifo_rden	1
first_read	1
rows_addr	0
cols_addr	0
max_num_pix	160
Now	100000 ns
Cursor 1	4871 ns

4662 ns to 5033 ns

Figure 4.5: Waveform shows store FSM writing value to SRAM interface

output video FSM reads the lookup table at the address created by concatenating the row and column counters to fetch the address of the pixel from the frame buffer, see cursor 3 in Figure 4.9. Then, that pixel address from the lookup is buffered, and the memory interface to the frame buffer is requested. Bit 19 of the address selects between banks 1 and 2. The pixel is read from memory and the internal counters are updated. The pixel is written to the egress video FIFO, see cursor 4 in Figure 4.9. The state machine loops back to request the lookup table again until it has read a complete video frame. Afterwards, the state machine returns to *idle* and the timer is reset. The regularly paced output of multiple video frames is shown in Figure 4.10.

There are a couple aspects of the design for the output video FSM that were intentionally simplified but could later be extended or further optimized. The read operations can be optimized for efficiency by adjusting the read order and performing burst reads. Much time is currently spent requesting memory between the lookup table and again for each pixel read. Reordering the reads could provide more memory cycles and improve support for higher resolution video. An extension that could

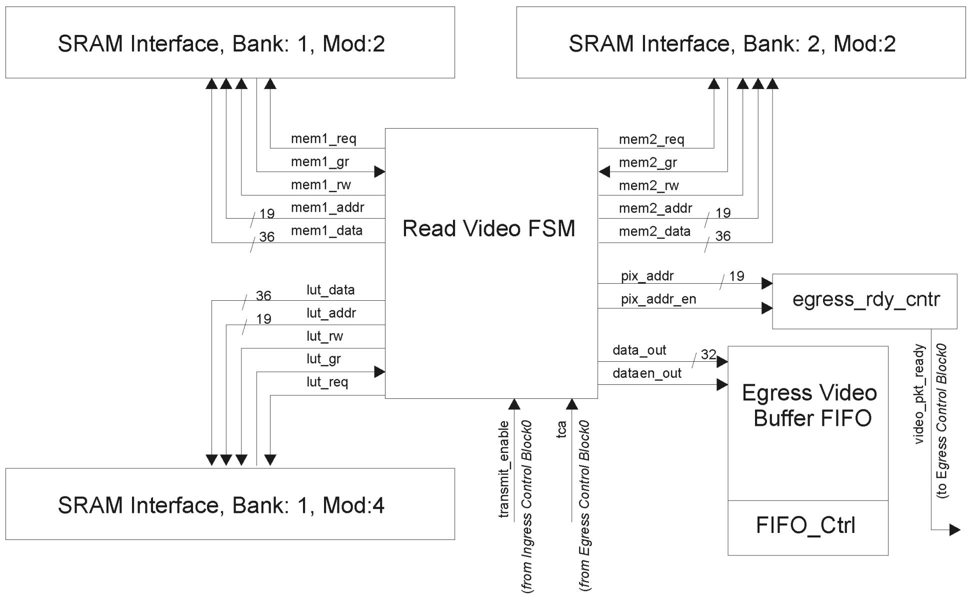
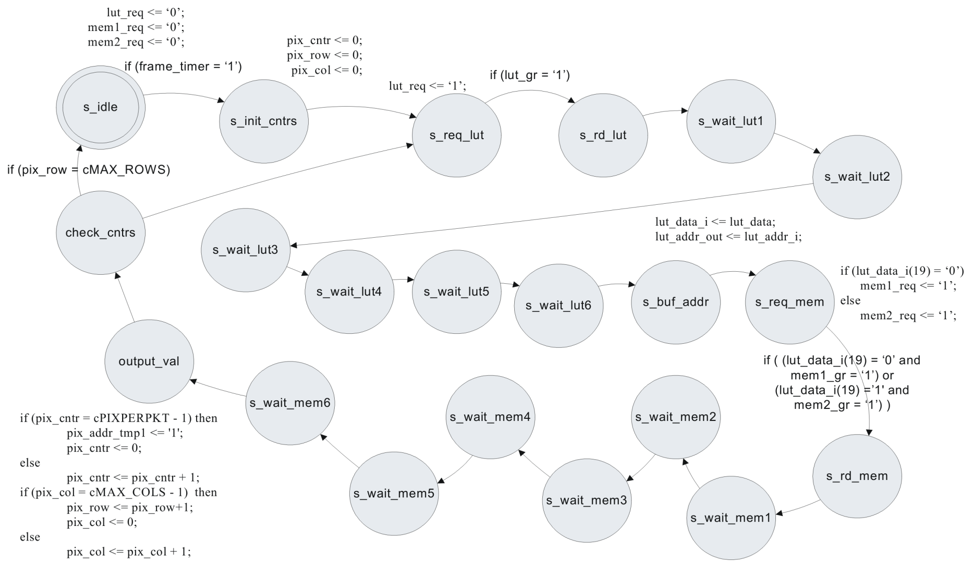


Figure 4.6: Block diagram illustrates output video FSM interfaces to memory components

enhance the quality would be to perform bilinear interpolation. Currently, the lookup table is rounded to the nearest neighbor at the time it is generated. The motivation for currently constructing the address by concatenating the row and column was to make it easier to later go back and add bilinear interpolation. Under this scheme, the addresses for four neighboring pixels can be determined easily, requiring only addition and subtraction operations. There is an important design note in section 4.3.1 that would affect designs that perform bilinear interpolation.

4.1.4 Egress Control Block

While the output video FSM stores pixels in the egress FIFO, the egress control block forms the packets for transmission. The `egress_rdy_cntr` signals to the egress control block when the number of pixels stored in the FIFO is enough for an output packet. There is a constant for this, which is set to 160 pixels. The egress control block uses stored packet header from the ingress control block. Currently, the source address



```
lut_req <= '0';
mem1_req <= '0';
mem2_req <= '0';
```

```
pix_cntr <= 0;
pix_row <= 0;
pix_col <= 0;
```

```
lut_req <= '1';
```

```
lut_data_i <= lut_data;
lut_addr_out <= lut_addr_i;
```

```
if (lut_data_i(19) = '0')
     mem1_req <= '1';
else
     mem2_req <= '1';
```

```
if ( (lut_data_i(19) = '0' and
   mem1_gr = '1') or
 (lut_data_i(19) ='1' and
   mem2_gr = '1') )
```

```
if (pix_cntr = cPIXPERPKT - 1) then
          pix_addr_tmp1 <= '1';
          pix_cntr <= 0;
else
          pix_cntr <= pix_cntr + 1;
if (pix_col = cMAX_COLS - 1)  then
          pix_row <= pix_row+1;
          pix_col <= 0;
else
          pix_col <= pix_col + 1;
```

Figure 4.7: Abbreviated state machine for output video

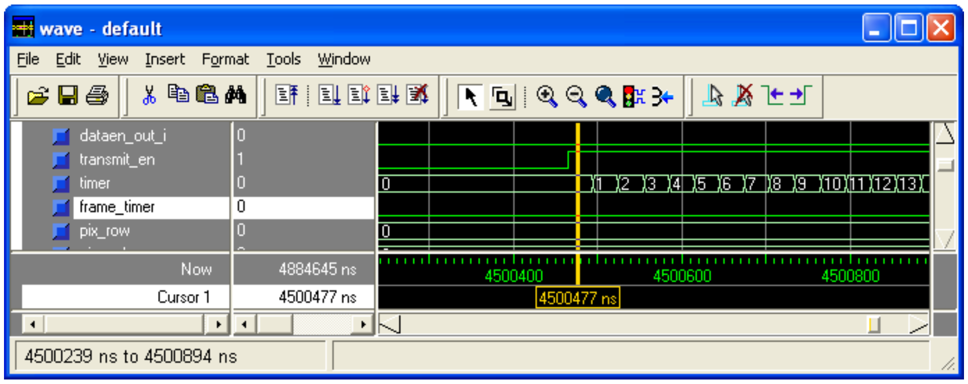
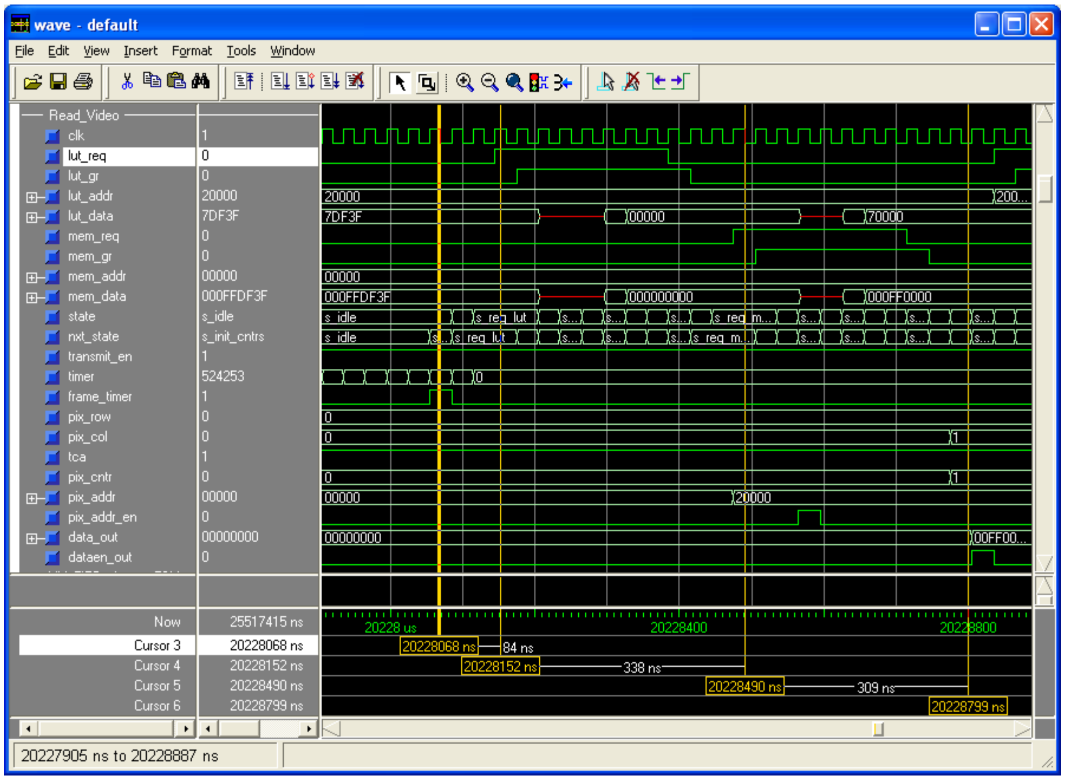


Figure 4.8: Waveform shows output video responding to transmit_en

from the incoming video stream is used as the destination address. It was planned that the bypass buffer could exert a backpressure signal when it becomes non-empty, which would pause the output video FSM. The egress control block would give the bypass packet priority for transmission. This has not been fully implemented, but it could be useful to keep the buffers from overflowing.

The dual transform engine has modifications to the egress half of the design. It has an additional egress video buffer FIFO and another instantiation of the output



clk	1
lut_req	0
lut_gr	0
lut_addr	20000
lut_data	7DF3F
mem_req	0
mem_gr	0
mem_addr	00000
mem_data	000FFDF3F
state	s_idle
nxt_state	s_init_cntrs
transmit_en	1
timer	524253
frame_timer	1
pix_row	0
pix_col	0
tca	1
pix_cntr	0
pix_addr	00000
pix_addr_en	0
data_out	00000000
dataen_out	0
Now	25517415 ns
Cursor 3	20228068 ns
Cursor 4	20228152 ns
Cursor 5	20228490 ns
Cursor 6	20228799 ns

20227905 ns to 20228887 ns

Figure 4.9: Waveform shows output video when timer expires

video FSM. The egress control block selects between outputting the two video streams and the bypass traffic.

4.2 Composition Engine

The composition engine design for two cameras is implemented as a hybrid between the single transform engine (TE) and dual TE because it takes two input video streams and outputs one fused video stream. From the block diagram, shown in Figure 4.11, the ingress side is similar to the dual TE, and the egress side is similar to the single TE. The major modification is performed when creating the lookup table entries, in this case bit 19 of the address toggles between video streams. The store video and store lookup state machines have the same behavior as discussed earlier.

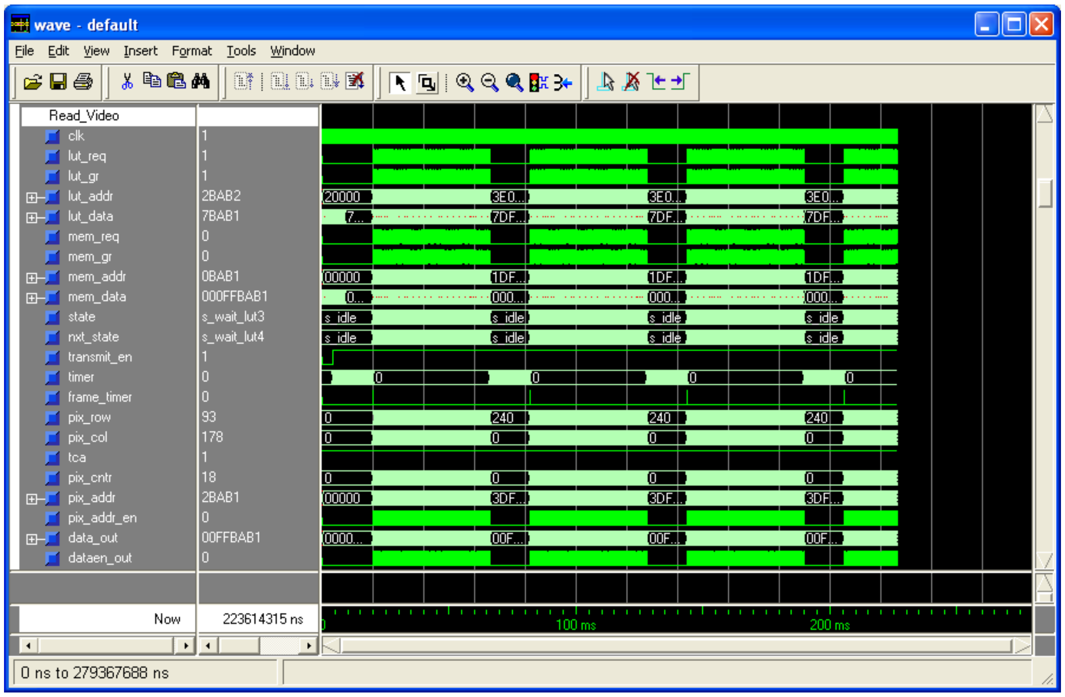


Figure 4.10: Waveform shows output video reading multiple frames

4.3 Support Applications

4.3.1 Video Sender

The video sender application was written to capture and send video from a Windows capture device. It is based on two example applications from the Microsoft DirectX 9.0 SDK, playcap.cpp and stillcap.cpp. In short, the program configures the network socket, initializes a capture graph, and then displays and transmits the video.

The network socket is configured using the WinSock API. The destination address and destination port are set at compile time, using compiler macros.

The setup for the capture graph requires connecting to the device and specifying connecting components for the data flow. The capture graph for the video sender, shown in Figure 4.12, shows the following connections: device driver, connected to a demultiplexor, connected to format adapter, connected to frame grabber, connected

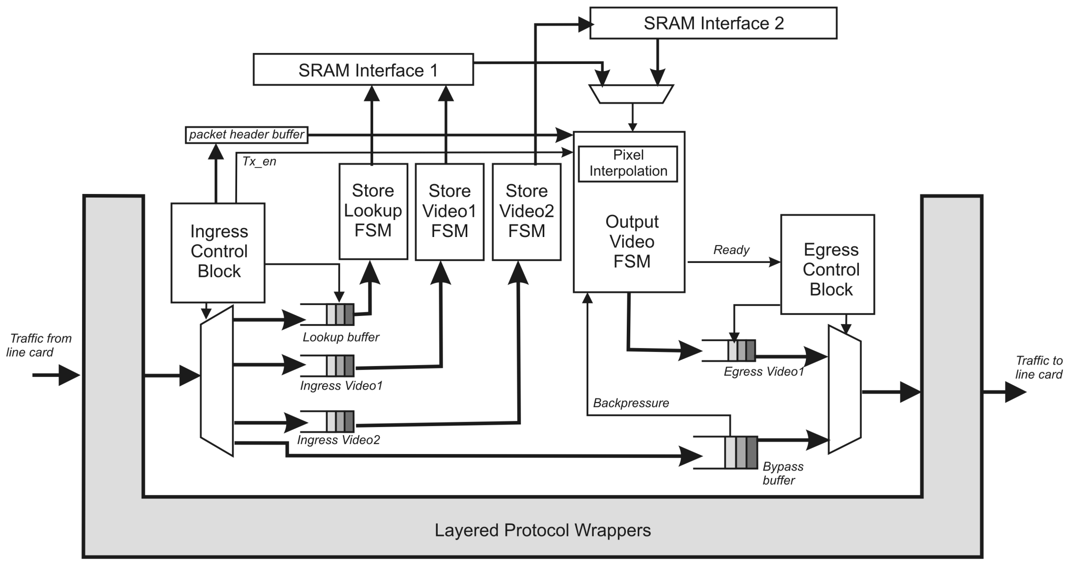


Figure 4.11: Block diagram of composition engine shows major components and data flow

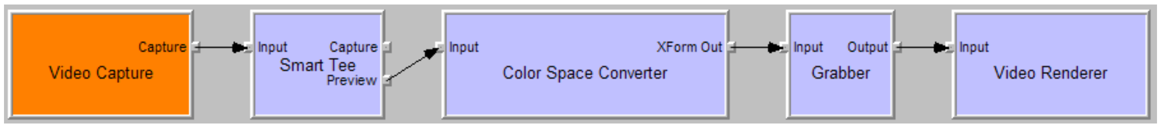


Figure 4.12: Video Sender capture graph screenshot from GraphEdit DX9.0 SDK tool

to the renderer display. The device driver can be any non-DVI capture device, for example USB cameras should work. The smart tee is a software demultiplexor for the device driver to provide a set of capture pins and preview pins to the DirectX API. Some device drivers have this capability built in. The color space converter adapts the video format from the native 24bpp to 32bpp, so the video format is consistent with the hardware designs. The sample grabber, sets up the call back function, which is actually done by creating a semi-COM object in the example. The callback function copies the frame to a frame buffer, segments the frame, and transmits small 160 pixel UDP packets. The renderer uses DirectDraw to scale and display the live video, see Figure 4.13. After the graph is complete, the application begins displaying and transmitting the video.

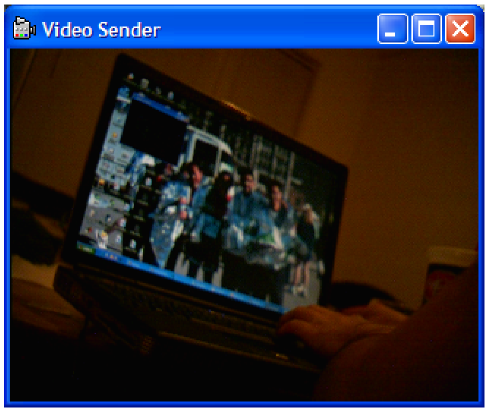


Figure 4.13: Video Sender previews and transmits live video

An important note is that there was a design oversight in this application that involved the differences in byte ordering between the Intel-based host and the network order. The video capture callback function returns a pointer to the frame buffer, and then a plain `memcpy()` is performed to copy data from the pointer address to an address within the outgoing packet's buffer. That is why the byte ordering of pixels from VideoSender application is actually reversed from the intended format, but as far as the hardware is currently concerned pixels are 32-bit values written to and read from memory. If the hardware design is modified to process video data, then it should take this into account, either in software or hardware.

4.3.2 Video Receiver

The video receiver is an even simpler application that listens to a UDP socket and displays the received video. The program initializes an empty frame buffer and configures the network socket. The Winsock select function `WSAAsyncSelect` is used to listen for read and write events (`FD_READ | FD_WRITE`) to the socket descriptor. On an event the `rcv_packet()` function is called that copies the pixels to the frame buffer. The display is currently updated, by declaring the window region invalid and calling `SetDIBitsToDevice` to copy the frame buffer to the window region, whenever packets with base address set to zero are received.

4.3.3 LookupSender

A simple program was written to create control packets and program the lookup table. This program was used to generate input packets for simulation and later modified to transmit packets these packets to control the actual hardware. Unlike the `VideoSender`, the `LookupSender` formats the packets in correct network byte order.

4.4 Remarks

Both the transform engine and composition engine are circuits that were designed to be tested in the RAD FPGA of the FPX platform, but they could be deployed in future FPGA platforms. The FPX was chosen as the development platform because of its existing UDP protocol wrappers and SRAM controller infrastructure. At outset of development there was discussion that a small, lightweight version of the FPX, called the wireless access sensor pod (WASP), might eventually be implemented to serve as the platform for the sensor network processing nodes. This transform engine design could later be deployed in a WASP environment, provided that the WASP provides similar interfaces to supporting infrastructure.

In another video project, the author implemented a hardware-based video transcoder for the FPX to process MJPEG video. The video transcoder module throttles the bandwidth of MJPEG video by selectively discarding high frequency coefficients, which can be useful to perform during network congestion. The module decodes the DCT frequency coefficients from the Huffman entropy-encoded segment.

Tile-size regions of the video frame can selectively retain their high frequency detail based on a user-defined 8x8 image map. Alternatively, the DC coefficient of each 8x8 pixel blocks can be used instead to preserve the detail within the regions containing particular colors of interest. Adding MJPEG support and also some of the functions from the video transcoder like filtering regions of disinterest might fit well in a future version of the designs. The video-compression-related details of that design made it more complex than the simple circuits described here, so any hardware projects requiring video compression should budget a major amount of time for just the compression details.

Chapter 5

Tests and Performance Results

This chapter describes the relative advantage of each system, since the hardware and software versions were designed to support different features. The Java applications support three cameras, employ video compression, and relied on shared execution environments. The hardware modules supported two cameras, did not support compression, and the design requires dedicated hardware to be available. Keeping in mind what aspects of the designs worked well for each system, latency and throughput are compared.

5.1 JMF Software

5.1.1 Configuration

The system that was implemented for the JMF tests combine video from three webcams. The physical configuration included a dual processor Athlon 2800 MP system, a dual processor Xeon 2.0 GHz system, and two Athlon workstations-all connected by 100Mbps Ethernet, as shown in Figure 5.1. Color-coded symbols are used in the Figure 5.1 to match the input streams to their source. USB webcams were connected

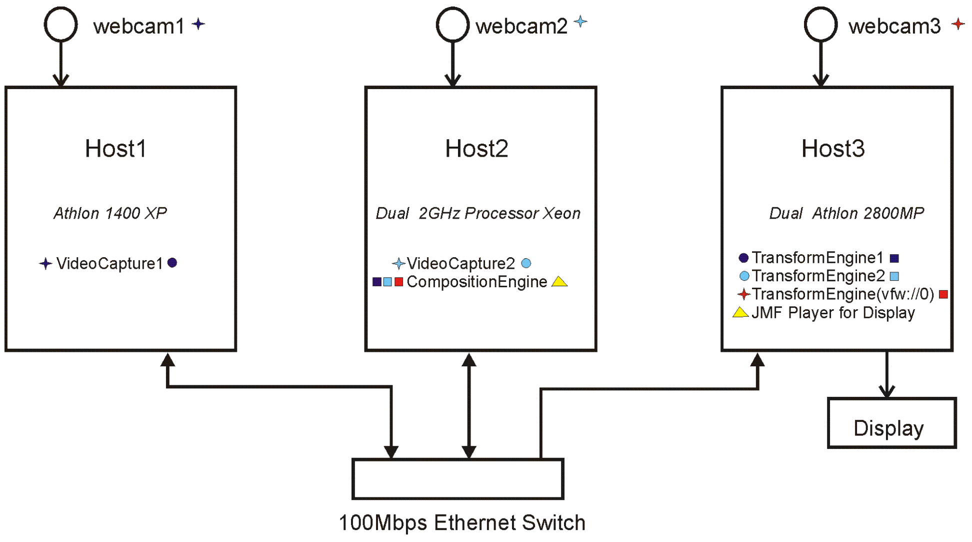


Figure 5.1: JMF software physical connection diagram (symbols left of text are inputs; right are outputs)

to each workstation. Host3 ran a separate instance of the `TransformEngine` for each of the input streams. Host2 ran the `CompositionEngine`, which turned out to be a surprisingly computationally intense task to fuse these transformed video streams.

5.1.2 Performance

Table 5.1: JMF Application Performance (values are approximate)

	Frame Rate	**Latency**	**Output Bandwidth**
	frames-per-sec	seconds	Mbps
VideoCapture	24-27	less than 1	2.5*(2*instances)
TransformEngine	20	1	2.5*(2*instances)
CompositionEngine	10-20	2-4	2.5

As Table 5.1 shows, the output video streams of each of the `TransformEngines` was better than the combined output video of the `CompositionEngine` in terms of latency. The output of the `CompositionEngine` lagged by a few seconds. It is estimated that the cause of this latency was due to the overhead at each stage related to video compression, which occurs several times, as illustrated in Figure 5.2. This figure shows that operations to encode and decode JPEG/RTP were performed at each of these three transmission stages. Even though Host2 and Host3 had heavy loads because they executed multiple applications in this configuration, it must be taken into account that those machines had much greater processing capability than the typical sensor mote platform. Due to the current granularity of this code design and the disappointing performance of the `CompositionEngine`, it is estimated that adding machines would not cause much improvement in latency.

Aside from latency, the overall quality of the video was good and the bandwidth requirements were reasonably low. Each stream had a variable bit rate of approximately 2.5 Mbps, using typical compression quality.

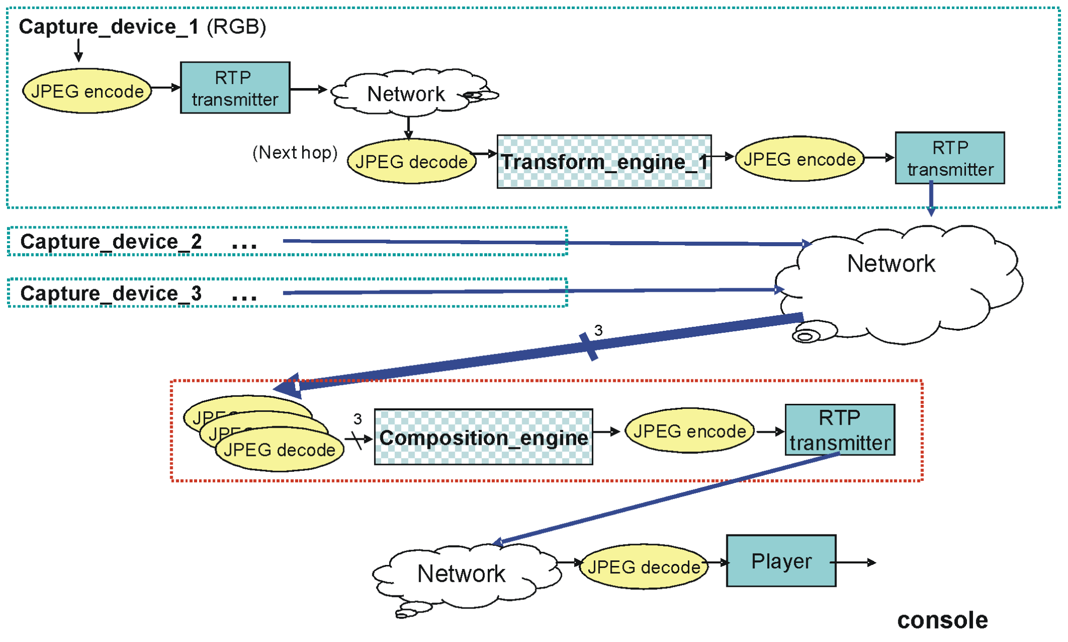


Figure 5.2: The multiple stages of encoding and decoding the video streams contributed to overall latency

5.2 Hardware Modules

5.2.1 Design resources

After coding and simulating the designs for the transform engine and composition engine, the designs were mapped to actual hardware. The VHDL code was compiled and synthesized using the Synplify Pro tool by Synplicity. This mapped the behavioral and structural code to an EDIF description of the required hardware resources on the FPGA. Sythesis results are shown in Table 5.2. To continue this topic of resource utilization, Table 5.3 shows the amount of off-chip SRAM utilized by the hardware modules. Xilinx ISE 6.2 CAD tools were used to place and route the EDIF files and generate the binary FPGA program configuration. Timing information from the place and route results is shown in Table 5.4[1].

Table 5.2: Synthesis Results – XCV2000E Resource Utilization

	# LUTs	**% LUTs**	**# BlockRAM**	**% BlockRAM**
Single video TE	820	2	55	34
Dual video TE	1230	3	79	49
Composition engine	1032	2	63	39

Table 5.3: SRAM Utilization

	Frame Buffer	**Lookup Table**
	MB	MB
Single video TE	0.5	0.5
Dual video TE	1.0	1.0
Composition engine	1.0	0.5

Table 5.4: Place and Route Timing Results

	Timing
	MHz
Single video TE	48.6
Dual video TE	49.6
Composition engine	45.4

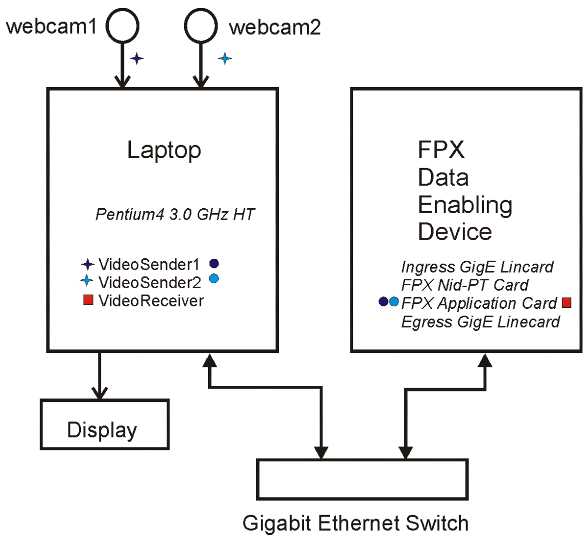


Figure 5.3: Hardware tests physical connection diagram

5.2.2 Configuration

Each of the hardware modules was developed and tested separately for functionality, rather than demonstrating the combined application. The test configuration, shown in Figure 5.3, consists of a laptop connected to an FPX gigabit Ethernet system. The operation of the single video TE, dual video TE, and composition was tested individually. The `VideoSender` transmits a video stream to the FPX GigE system. The hardware modules currently swap the source address to be the destination address, when generating the output video streams.

[1]The BlockRAM results were adjusted to subtract the BlockRAMs used by the Xilinx Chipscope analyzer, however, the timing results were not altered

5.2.3 Performance

Approximate performance results are shown in Table 5.5. At the moment the quality appears to be limited to some extent by the software applications running on the host. The transmission between `VideoSender` and `VideoReceiver` applications is one of the current bottlenecks. Tests show that these applications running on the same host will produce packet loss when trying to send full frame rate video, which is most likely due to the high data rate of the uncompressed video 70Mbps per stream. Since packets contain 160 pixel chunks, packet loss produces almost a scanline type of visual effect.

There were also occasions when a few random pixels appeared with a noticeably different color in some frames. The current estimate for why this happens is that a memory-request bug in the output video FSM caused it to skip the *s_mem_req* state. This bug has recently been fixed.

This design does well, however, in terms of having low latency. Testing revealed that there was no major latency, even when combining two video streams using the hardware composition engine.

Table 5.5: Hardware Module Performance (values are approximate)

	Frame Rate[a]	**Latency**	**Total Bandwidth**[b]
	frames-per-sec	seconds	Mbps
Single video TE	30	less than 1	80
CompositionEngine	30	less than 1	125

[a]There was packet loss between just the VideoSender and VideoReceiver, and so there were noticeable gaps of pixels in frames

[b]This was not accurately measured; these are the observed rates from Windows task manager

5.3 Some Lessons Learned

There is obviously much less overhead in the simple hardware designs, without video compression, but even if there was compression, a hardware-accelerated transcoder could be designed that would only add a few milliseconds of latency per frame for encode and decode operations. This hardware assist could be designed as a plug in to support different codecs and implemented in an FPGA, or it could be implemented as a low power ASIC, provided that there was a common video format.

If we take a moment to revisit the motivation for using distributed processing nodes for this design, it was expected that using distributed processing nodes would lead us to more scalable design than one with a centralized processing core. Since the number of sensors used in the test environments was so small, however, the cost of the additional processing overhead from video compression was increased because the distributed design has more transmission stages between nodes. This increased cost for the small environment appears to outweigh the benefit of distributing the load. So, it seems like some cost function could be used to determine an appropriate distribution strategy. It might be good to note that the cost for some video compression algorithms in hardware can be lower than the relative cost that it might have in software.

Right now, it is not clear when applying transforms and performing composition at separate nodes becomes the right level of granularity to use as an architecture to solve this problem. A small configuration, consisting of three video sensors and a single host, provided the host uses one lookup table that mapped 3 frame buffers, could possibly outperform a system that requires extra stages of transmission and compression overhead. To characterize this trade-off in future experiments, simulations are needed.

Chapter 6

Conclusions

Hardware and software have been implemented to perform a video surveillance application for a video sensor network. The services of this application include a Transform Engine (TE) and Composition Engine (CE), which exist as modules that can be programmed into processing nodes that are distributed throughout a network. This paper describes how data from multiple live video sensors can be processed and then fused to generate custom views that are not restricted to actual camera position. The functions that alter perspective, correct radial distortion, zoom, and rotate are performed off-line to calculate lookup tables that are stored and applied at nodes. Data in the network is transmitted as UDP Internet Protocol (IP) packets.

The reconfigurable hardware modules were coded in VHDL and have been prototyped using Washington University's Field Programmable Port Extender (FPX) platform. The transform engine circuit utilizes approximately 34 percent of the resources of a Xilinx Virtex 2000E FPGA, and can be clocked at frequencies up to 48 MHz. The composition engine circuit utilizes approximately 39 percent of the resources of a Xilinx Virtex 2000E FPGA, and can be clocked at frequencies up to 45 MHz.

6.1 Future Work

One aspect of the design that is currently sore, and appears to have the most potential for improving performance, is the lack of any hardware-accelerated support for video compression. This change could dramatically improve network performance and reduce throughput for the hardware modules. Good candidates for compression algorithms include MJPEG and MJPEG-2000. If MJPEG-2000 were selected, then this design could take advantage of region of interest coding and the special data ordering of high frequency coefficients.

Another good area of work would be to extend the hardware modules to implement a homography-based transform engine[1] that would store information about the transform in the form of functions rather than in the form of large lookup tables. The terms of the lookup table could be calculated at run-time based on the coefficients of the homography transform matrix.

It would be possible to implement software or additional logic to automate the alignment between videos as a way to compensate for sensor movement. Not to mention, this could also greatly simplify the basic configuration. Hardware or software could try to minimize a cost function to locate pairs of correspondence points between images. The homography transform coefficients could be periodically solved for, and used to generate and upload new transform lookup tables. This might be a good area of research because there are potentially many of ways to approach solving this problem.

A simple extension to the project would be to add bilinear interpolation. The current address scheme was chosen to make the implementation of bilinear easier.

[1]The homography-based tranform engine was suggested by Pless

This could slightly improve the quality of the transformed video, especially in regions near vanishing points.

Appendix A

Configuration Details for the FPX

A.1 Routes

The configuration details necessary to repeat testing of the hardware modules are described in this appendix. The single video tranform engine (TE), dual video TE, and composition engine modules were implemented on the line card interface of the reprogrammable application device (RAD) FPGA of the FPX. The following information is used to route traffic through the circuit for an "FPX-in-a-box" Gigabit Ethernet system, described in [18].

The combination of two FPXs, each with stacked Gigabit Ethernet line cards, is used as the test platform for each network node. The Gigabit Ethernet line cards provide physical connection interfaces to pass traffic from a gigabit Ethernet source through one FPX application device.

The configuration for routing traffic through an FPX is updated using the NCHARGE control software[19]. The network interface device (NID) FPGA of the FPX routes traffic between the line card and switch physical connection interfaces and `rad_lc` and `rad_sw` interfaces on the card. The following NCHARGE script configures the NID routing appropriately to pass traffic through the line card interface of the RAD. The VideoSender application transmits to the IP address of the GigE card, so that the packets are routed through the FPX as control messages.

```
#defaults
print "./basic_send 1.0 t 0 32 1 0 0 0";
print ‘./basic_send 1.0 t 0 32 1 0 0 0‘;
print ‘./basic_send 1.0 t 0 33 1 0 0 0‘;
print ‘./basic_send 1.0 t 80 32 1 0 0 0‘;
print ‘./basic_send 1.0 t 80 33 1 0 0 0‘;

# This is the bitfile for the composition engine
print ‘./basic_send 0.1 c comp_eng.bit‘;

# Route only control messages VCI=50 or 0x32 through rad_lc
print ‘./basic_send 1.0 t 80 32 0 0 0 1‘;
print ‘./basic_send 1.0 t 0 32 0 2 3 1‘;
```

A.2 VidSender and VidReceiver Notes

`VidSender` and `VidReceiver` are designed to run under Windows 2000/XP and their options are configured at compile time. In `VidSender`, the destination IP address should be set to the address of the GigE line card before compile time. The hardware circuits are currently programmed to swap the source address of the packets to use as the destination for outgoing video to return video to `VidReciever`.

Appendix B

Transform Tool

The transform tool[1] calculates the lookup table necessary for the transform engine to transform the camera perspective. The tool can be used to calculate the location of vanishing points and calculate the coordinates of source pixels that would rectify the view. The tool can apply radial correction to adjust for barrel distortion caused by the camera lens. This program is written in Java and extends JMStudio. The program is loaded by the snapshot function (`Ctrl-S`) in a modified version of `JMStudio` called `JMStudioUpdated`[2].

B.1 Vanishing point geometry

Perspective correction enables fine control over the tilt of the picture plane so that the target plane can appear perpendicular to the viewer. The vanishing points are determined by specifying pairs of non-parallel lines that and then calculating their intersection. Typically, the corners of a quadrilateral are specified as the four points that are used to solve for two vanishing point intersections. From the intersection

[1]This program was also coauthored by David Barnett and Christopher Zuver

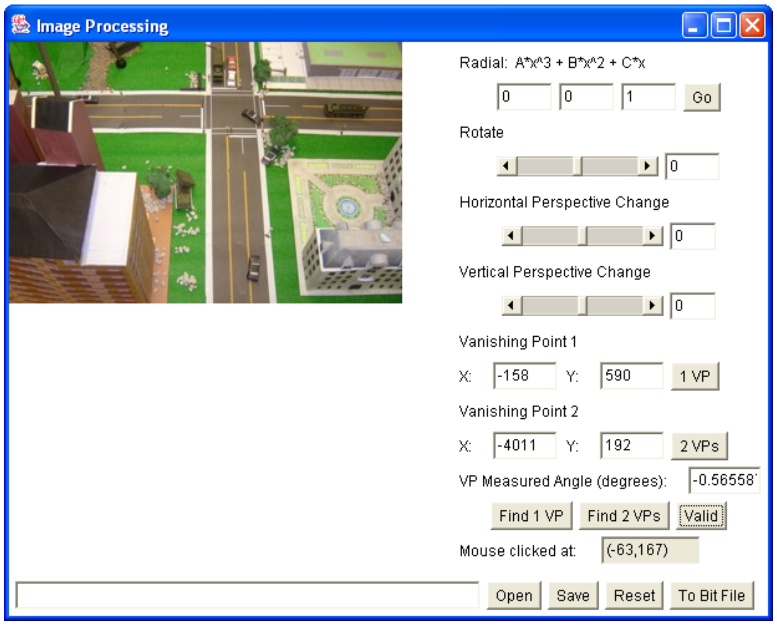
[2]The code for this application is specified in ImageProcess.java

points, a transform is calculated that would rectify the lines to parallel. The function for calculating the lookup table sets one vanishing point as the x-axis and the other vanishing point as the new y-axis, and during the intermediate calculation rotates and translates the coordinates, in order to produce an output view that is centered in the frame and vanishes along the display's x axis and y axis.

The user specifies whether there are one or two vanishing points that they wish to correct, a screenshot of the interface is shown in Figure B.1. Two vanishing point geometry is sufficient to transform views of planar surfaces. For one vanishing point the user identifies four points that define a pair of vanishing lines. For two vanishing points the user identifies eight points that define two pairs of vanishing lines. The intersection of each pair is calculated to identify two vanishing points in the original view. Translation, scaling, rotation, and radial correction parameters are applied at the time the transformation table is generated. The program writes the transformation lookup table to disk, which contains two arrays of x coordinates and y coordinates of the source pixel addresses. The display is updated to preview the transformation on a static image, shown in Figure B.2.

B.2 Radial correction

Radial correction adjusts for lens aberrations, such as barrel or pincushion distortions. The amount of radial distortion is specific to the lens of the camera, so each camera is first placed in front of a test pattern of straight lines. Using control points to measure the distortion, a polynomial function is determined to construct a lookup table that corrects the distorted radial distance to map to a predetermined radial distance B.3. The same parameters can be applied to correct other images taken using that camera lens[2].



Radial: A*x^3 + B*x^2 + C*x

0 0 1

Rotate 0

Horizontal Perspective Change 0

Vertical Perspective Change 0

Vanishing Point 1

X: -158 Y: 590

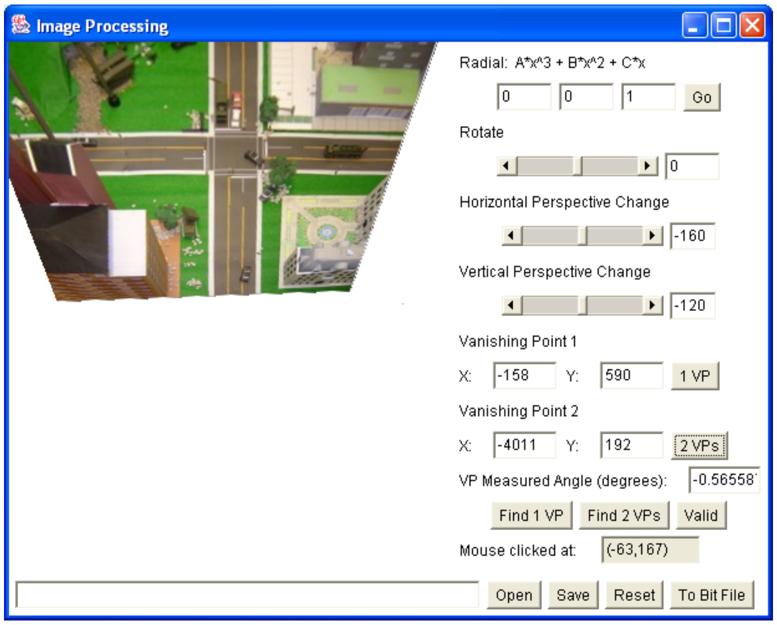
Vanishing Point 2

X: -4011 Y: 192

VP Measured Angle (degrees): -0.56558

Mouse clicked at: (-63,167)

Figure B.1: Transform tool interface with original perspective



Radial: A*x^3 + B*x^2 + C*x

0 0 1

Rotate 0

Horizontal Perspective Change -160

Vertical Perspective Change -120

Vanishing Point 1

X: -158 Y: 590

Vanishing Point 2

X: -4011 Y: 192

VP Measured Angle (degrees): -0.56558

Mouse clicked at: (-63,167)

Figure B.2: Transform tool interface with rectified perspective

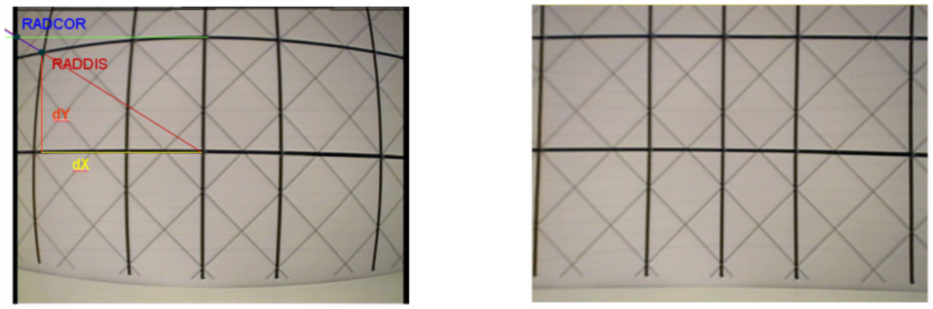


Figure B.3: Radial correction (a) original distorted image, (b) rectified image

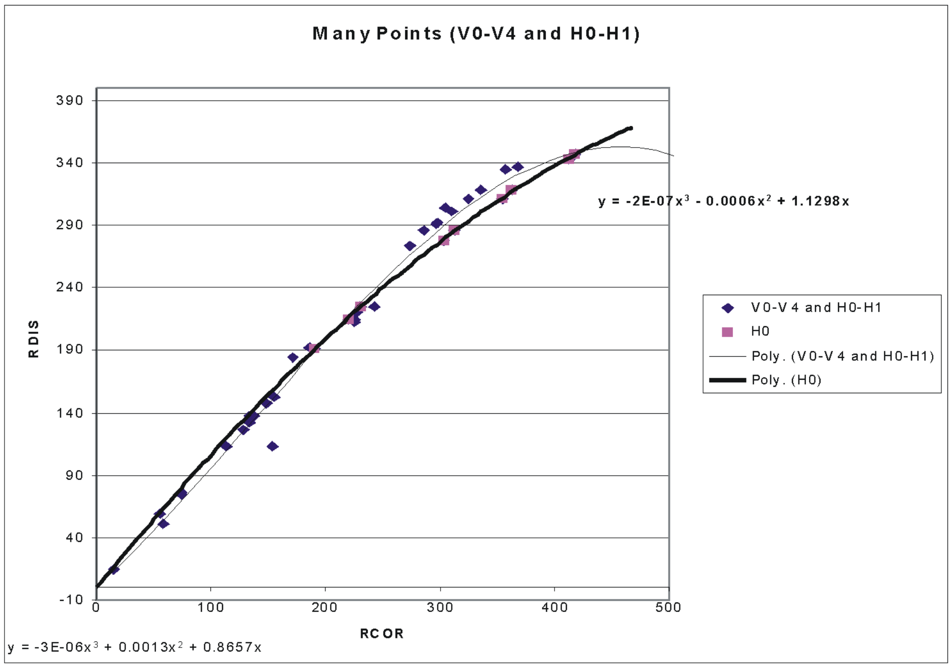


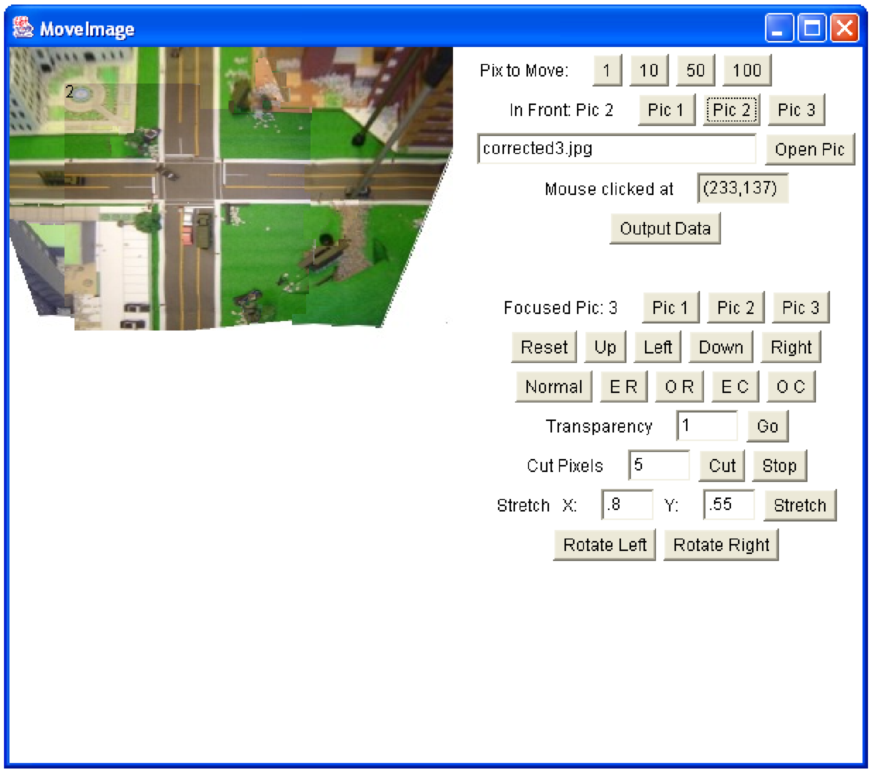
Figure B.4: Relationship between distorted radius and expected radius for this example can be modeled as 3rd order polynomial, which is used to calculate the inverse transformation

Appendix C

Alignment Tool

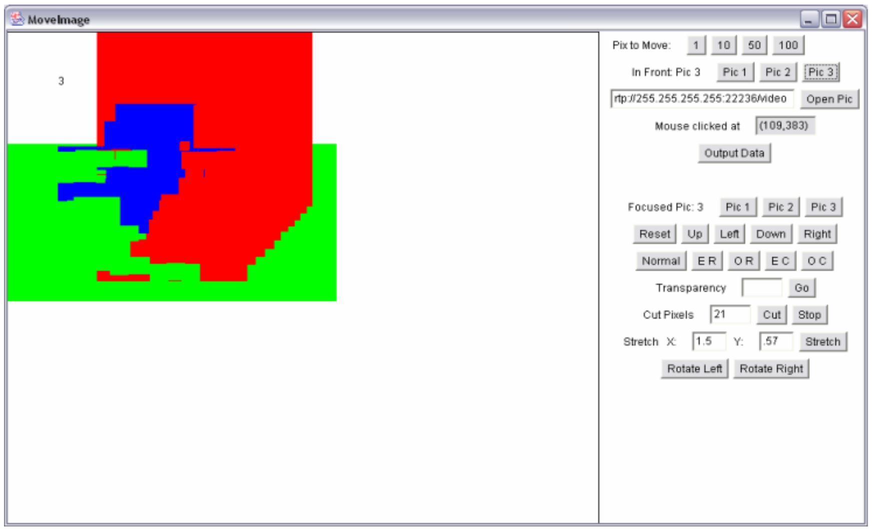
The composition tool[1], written by David Barnett, is a simple, stand-alone application used to specify the alignment of the video feeds after perspective correction. Up to three views can be manually scaled, rotated, and positioned until their alignment overlaps. The views currently overlap as an ordered stack of images. Then, regions of interest can be manually defined by cutting away sections from overlapped regions to view the underlying images. The graphical interface is shown in Figure C.1 with an example alignment of three views[2]. A lookup table that specifies camera, x coordinates, and y coordinates is generated as output, containing a mapping of pixels from the multiple views to the composite view. Figure C.2 shows the contribution from each camera by shading regions according to camera number[3].

[1]The code for this application is specified in MoveImage.java
[2]The composition was aligned by Christopher Zuver
[3]Specify the value 21 as the cut pixels parameter to show camera regions



In Front: Pic 2
corrected3.jpg
Mouse clicked at (233,137)
Focused Pic: 3
Transparency 1
Cut Pixels 5
Stretch X: .8 Y: .55

Figure C.1: Applet interface shows the composition of three views



In Front: Pic 3
rtp://255.255.255.255:22236/video
Mouse clicked at (109,383)
Focused Pic: 3
Transparency
Cut Pixels 21
Stretch X: 1.5 Y: .57

Figure C.2: Applet interface shades the regions based on camera number

Appendix D

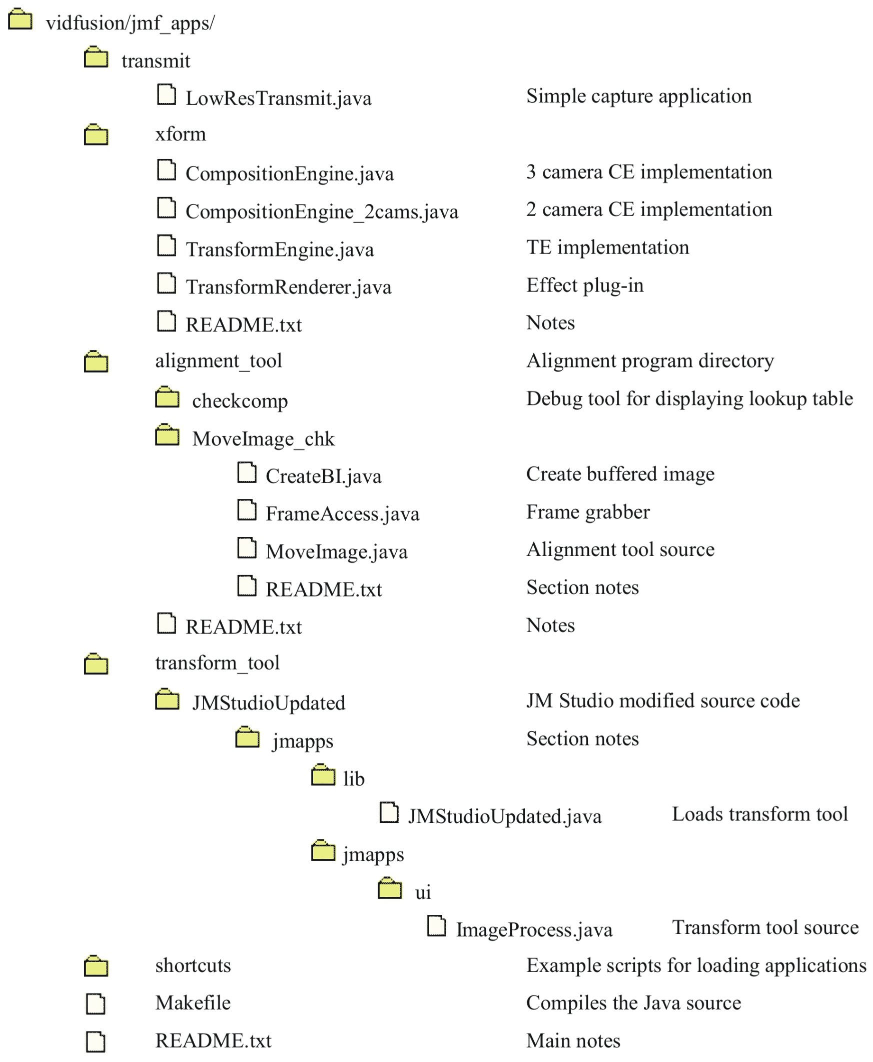
Archive of Source Files

D.1 Source code

This section describes the files that have been archived for this project. The Java implementation, hardware modules implementation, and support tools are contained in this archive. This archive has been put into the CVS tree at the following location: `cvsroot/FPX_ROOT/RAD/MODULES/VIDFUSION`. Also, a tarball copy of the files has been placed in `/project/fpx/vidfusion` on the network drive.

D.2 Java implementation and tools

The Java source and files are contained in a main subfolder `jmf_apps`, which is shown in Figure D.1. A `Makefile` has been provided to show examples of how to compile the applications. The Java SDK v1.4 or later and the Java Media Framework (JMF) v2.1.1e or later are needed for this process. `README` files have been created to describe the important files and brief notes in some of the directories. Example scripts for running the applications are included as batch (`.bat`) files in the `shortcuts` directory.



```
vidfusion/jmf_apps/
    transmit
        LowResTransmit.java                 Simple capture application
    xform
        CompositionEngine.java              3 camera CE implementation
        CompositionEngine_2cams.java        2 camera CE implementation
        TransformEngine.java                TE implementation
        TransformRenderer.java              Effect plug-in
        README.txt                          Notes
    alignment_tool                          Alignment program directory
        checkcomp                           Debug tool for displaying lookup table
        MoveImage_chk
            CreateBI.java                   Create buffered image
            FrameAccess.java                Frame grabber
            MoveImage.java                  Alignment tool source
            README.txt                      Section notes
        README.txt                          Notes
    transform_tool
        JMStudioUpdated                     JM Studio modified source code
            jmapps                          Section notes
                lib
                    JMStudioUpdated.java          Loads transform tool
                jmapps
                    ui
                        ImageProcess.java         Transform tool source
    shortcuts                               Example scripts for loading applications
    Makefile                                Compiles the Java source
    README.txt                              Main notes
```

Figure D.1: Abbreviated list of files for JMF applications

The transmit directory contains the simple capture application. The xform directory contains the TranformEngine, CompositionEngine, and TransformRenderer. The alignment tool directory contains the modified JMStudio source code. It can be run as shown below:

```
> cd \transform_tool\JMStudioUpdated\jmapps\lib

> java JMStudioUpdated [input_media_locator]
```

'`Ctrl-S`' will capture a screenshot and load the transform interface. Output .bit files will be written to this directory. The `.bit` file should be passed in as a command-line argument to TransformEngine.

D.3 DirectX support applications

The source for the DirectX applications are contained in the `dx9_apps` subfolder. These applications require Visual Studio.NET to compile. The parameters are currently set at compile time. The important files are listed in Figure D.2.

📁 vidfusion/dx9_apps/				
	📁 LutSender			
		📁 Debug		
			🗋 wsudpsnd.exe	Lookup table sender executable
		🗋 lutsender.cpp		Lookup table sender source
		🗋 wsudpsnd.sln		Visual Studio.NET project solution
	📁 VidReceiver			
		📁 Debug		
			🗋 vidreceiver.exe	VidReceiver executable for display
		🗋 vidreceiver.cpp		VidReceiver source
		🗋 vidreceiver.sln		Visual Studio.NET project solution
	📁 VidSender			Alignment program directory
		📁 Debug		
			🗋 wsudpsnd.exe	Vidsender executable
		🗋 playcap.cpp		VidSender source (different from the SDK)
		🗋 wsudpsnd.sln		Visual Studio.NET project solution
	🗋 README.txt			Notes

Figure D.2: Abbreviated list of files for DirectX video tools

D.4 FPX modules

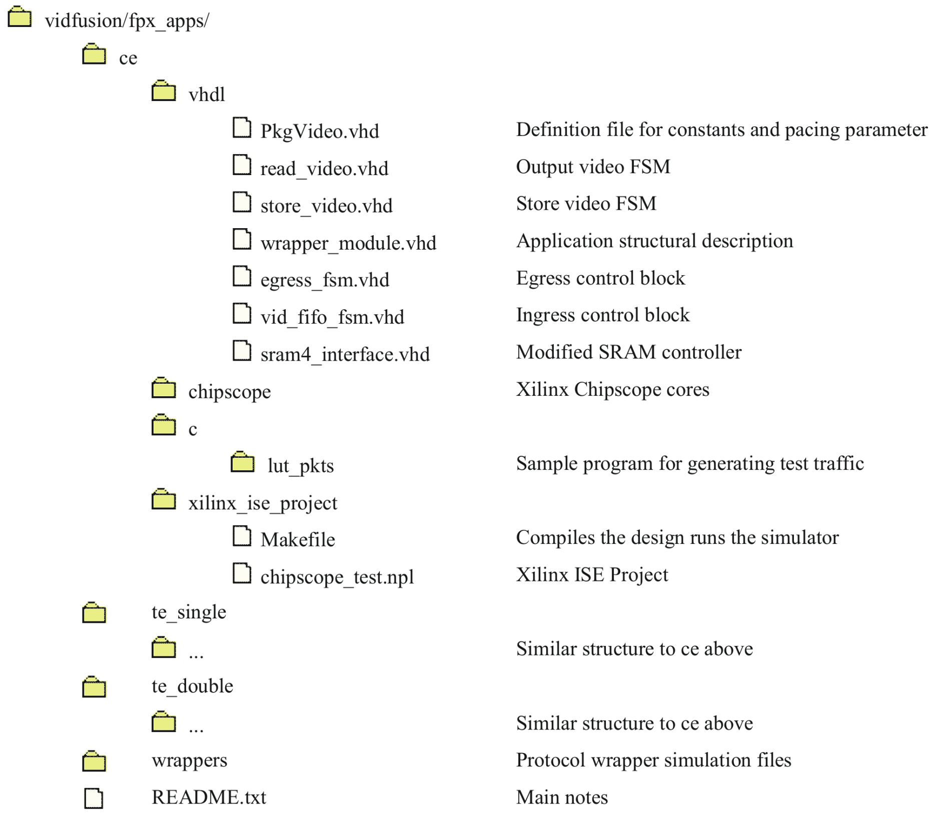
The `fpx_apps` directory contains the VHDL hardware description and project files for the composition engine, dual transform engine, and single transform engine designs.

The `ce`, `te_double`, and `te_single` directories are set up as projects for the Xilinx ISE Foundation 6.2 tools.

Simultations should be run by typing ‘make sim’ from the command line in the respective Xilnx project directory. This was a work around to simulate with the source code version of the protocol wrappers, without having to include those files in my project.

That `Makefile` in the project directory will need to be updated accordingly as changes are made.

Note that some of the files contain Chipscope debug cores (VIO, ICA, ILA) in `wrapper_module.vhd`. These components need to be commented out in order to run ModelSim simulations.



```
vidfusion/fpx_apps/
    ce
        vhdl
            PkgVideo.vhd              Definition file for constants and pacing parameter
            read_video.vhd            Output video FSM
            store_video.vhd           Store video FSM
            wrapper_module.vhd        Application structural description
            egress_fsm.vhd            Egress control block
            vid_fifo_fsm.vhd          Ingress control block
            sram4_interface.vhd       Modified SRAM controller
        chipscope                     Xilinx Chipscope cores
        c
            lut_pkts                  Sample program for generating test traffic
        xilinx_ise_project
            Makefile                  Compiles the design runs the simulator
            chipscope_test.npl        Xilinx ISE Project
    te_single
        ...                           Similar structure to ce above
    te_double
        ...                           Similar structure to ce above
    wrappers                          Protocol wrapper simulation files
    README.txt                        Main notes
```

Figure D.3: Abbreviated list of files for FPX hardware modules

References

[1] Florian Braun, John W. Lockwood, and Marcel Waldvogel. Layered protocol wrappers for Internet packet processing in reconfigurable hardware. In *Proceedings of Symposium on High Performance Interconnects (HotI'01)*, pages 93–98, Stanford, CA, USA, August 2001.

[2] Helmut Dersch. Correcting barrel distortion. `http://www.path.unimelb.edu-.au/~dersch/barrel/barrel.html`, 1999.

[3] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John W. Lockwood. Deep packet inspection using parallel Bloom filters. In *Hot Interconnects*, pages 44–51, Stanford, CA, August 2003.

[4] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *MOBICOM'99*, Seattle, Washington, August 1999.

[5] G. Fangi, G. Gagliardini, E. S. Malinverni, and C. Nardinocchi. Photointerpretation and small scale stereoplotting using digitally rectified photographs by geometrical constraints. In *CIPA'01*, Potsdam, Germany, September 2001.

[6] JMF 2.0 FCS. Java media framework api guide. `http://java.sun.com-/products/javamedia/jmf/2.1.1/guide/index.html`, 1999.

[7] R. Kumar, M. Wolenetz, B. Agagarwalla, J. Shin, P. Hutto, A. Paul, and U. Ramachandran. Dfuse: A framework for distributed data fusion. In *SenSys'03*, Los Angeles, California, November 2003.

[8] John W Lockwood. An open platform for development of network processing modules in reprogrammable hardware. In *IEC DesignCon'01*, pages WB–19, Santa Clara, CA, January 2001.

[9] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, February 2001.

[10] John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, September 2003.

[11] John W Lockwood, Todd Sproull, James Moscola, David Schuehler, Dave Lim, Sarang Dharmapurikar, and Chris Neely. Gigabit kits: FPX workshop. `http://www.arl.wustl.edu/arl/projects/fpx/-workshop_0602/agenda.html`, June 2002.

[12] Sun Microsystems. Java media framework api. `http://java.sun.com/jmf`, 2003.

[13] K. Nguyen, G. Yueng, S. Ghiasi, and M. Sarrafzadeh. A general framework for tracking objects in a multi-camera environment. In *DCV'02*, Clearwater, Florida, November 2002.

[14] R. Pless, T. Brodsky, and Y. Aloimonos. Detecting independent motion: The statistics of temporal continuity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):768–773, August 2000.

[15] G. J. Pottie and W. J. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*, 43(5), 2000.

[16] David V. Schuehler and John W. Lockwood. TCP splitter: A TCP/IP flow monitor in reconfigurable hardware. *IEEE Micro*, 23(1):54–59, January 2003.

[17] David V. Schuehler, James Moscola, and John W. Lockwood. Architecture for a hardware based, tcp/ip content scanning system. In *Hot Interconnects*, pages 89–94, Stanford, CA, August 2003.

[18] Haoyu Song. Secure remote control and configuration of the fpx platform in gigabit ethernet environment. Technical Report WUCSE-2003-68, Washington University in Saint Louis, Sever Institute, 2003.

[19] Todd Sproull, John W. Lockwood, and David E. Taylor. Control and configuration software for a reconfigurable networking hardware platform. In *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, April 2002.

[20] David E. Taylor, John W. Lockwood, Todd S. Sproull, Jonathan S. Turner, and David B. Parlour. Scalable IP lookup for programmable routers. In *IEEE Infocom 2002*, New York NY, June 2002.

[21] D. G. Thaler and C. V. Ravishankar. Distributed top-down hierarchy construction. In *INFOCOM'98*, San Francisco, California, April 1998.

Vita

Christopher E. Neely

Date of Birth	November 18, 1979
Place of Birth	Saint Louis, Missouri
Education	B.S. Computer Engineering, Washington Univ., May 2002
Publications	John W. Lockwood, Chris Neely, Chris Zuver, and Dave Lim. Automated Tools to Implement and Test Internet Systems in Reconfigurable Hardware. In *ACM SIGCOMM Computer Communication Review (CCR)*, vol 33, no 3, Jul. 2003, pp 103-110. John Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim. An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall. In *Proceedings of Field-Programmable Logic and Applications (FPL)*, (Lisbon, Portugal), Sep. 2003. David Lim, Christopher E. Neely, Christopher K. Zuver, and John W. Lockwood. Internet-based Tool for System-on-Chip Integration. In *IEEE Computer Society International Conference on Microelectronic Systems Education (MSE'03)*, (Anaheim, CA, USA), Jun. 2003. Christopher K. Zuver, Christopher E. Neely, and John W. Lockwood. Internet-based Tool for System-On-Chip Project Testing and Grading. In *IEEE Computer Society International Conference on Microelectronic Systems Education (MSE'03)*, (Anaheim, CA, USA), Jun. 2003.

May 2004