Uniform Access to the Cameraverse

Gregor Miller and Sidney Fels
Human Communications Technology Laboratory
University of British Columbia
Vancouver, Canada
{gregor,ssfels}@ece.ubc.ca

ABSTRACT

We introduce a novel framework for camera access which provides a uniform interface to many different camera types as well as a novel camera-oriented addressing scheme. Previous attempts to provide simple access to cameras are generally OS specific or lacking in functionality. We present the *Unified Camera Framework*, a novel camera access scheme which works across operating systems, provides an image descriptor for access to native images and defines the *cameraverse* using a unique addressing protocol. A unified configuration model is presented to allow manipulation of camera parameters to the level each camera supports. Validation of the ideas presented is given in the form of a proof-of-concept implementation called the *All Seeing Eye*.

Categories and Subject Descriptors

I.4.1 [**Computing Methodologies**]: Image Processing and Computer Vision—*Digitization and Image Capture*; I.4.9 [**Computing Methodologies**]: Image Processing and Computer Vision—*Applications*

1. INTRODUCTION

Accessing image data captured by cameras is one of the most commonly performed tasks in computer vision yet it is also one that is often overlooked. Various camera access solutions are available, but they are either specific to a package [10], a particular type (IIDC or Point Grey Research) or operating system (Quicktime$^{\text{TM}}$ DirectShow or VideoForLinux), or the functionality they provide is lacking in basic access[4]. No solutions exist which provide an abstraction over camera access and configuration (above the system level) which is extensible to existing drivers, or for multiple cameras. This paper presents the *Unified Camera Framework* (UCF), a novel camera access scheme for uniform camera access and configuration.

The universal need for access to cameras for the vision community and hobbyists is our main motivation given the current lack of a sufficient abstraction over existing technologies. The creation of another standard is not ideal since it is difficult to establish support and unlikely to improve the current situation, so we have instead created an abstraction layer over image and camera access which supports existing standards, and can support new cameras through its extensible framework. The contribution of this work is a set of abstractions which provide a uniform access method, and include: a generic image description; uniform access to cameras across operating systems and networks; configuration of all cameras up to their capabilities; and an addressing scheme to allow access to any camera device on the network.

This paper's contributions add to the field of computer vision and to camera technology in general, however we combine them to form a contribution to those unfamiliar with accessing camera data. For instance, by providing a generic image header, any additional component in a system which supports this description does not need to export image specifics to users; all that is visible is the image as its own object, without revealing details such as pixels or format (although these are still available). The Unified Camera Framework also contributes to computer vision researchers: the same code will work across operating systems since the access method is uniform, and cameras are accessible locally and on the network through the abstraction.

There have been many attempts to create open repositories of software supporting the vision community [4, 13, 5], many of which support camera or image access. Unfortunately these frameworks usually include a combination of image capture, convenience utilities (such as image I/O) and specific algorithms for image processing and understanding contained within these libraries, and the image format used for images is specific to that library. This combination of features demonstrates that these frameworks suffer from a lack of sufficient conceptual organization of the vision problem's constituent tasks. Previous research has shown that lack of scope definition and overlap across frameworks leads to a breakdown in component reusability[8]. Therefore we propose the following classification of scope for computer vision:

Access	:	Retrieval of data
Transfer	:	Mediation between modules
Convert	:	Conversion into required format
Modify	:	Applying filters, crop, transforms, etc.
Analyze	:	Using vision to model a scene

Decomposing the problem in this way promotes code re-use as well as focussing development effort on well-defined parts of computer vision. Under this classification we present UCF as a solution to the **Access** problem. The other components in the computer vision pipeline have various example solutions, such as Hive[2] for transfer, ImageMagick for conversion and CoreImage^{TM} for modification (although these last two also expand out of our defined scope). As yet there are no known solutions to the **Analysis** problem which limit their scope to analysis while also being comprehensive and accessible to those not specialized in computer vision techniques.

UCF addresses the camera problem by defining the problems as increasing levels of access (image, interface, general camera), and providing a solution for each. The image access problem, described in Section 3.1, is that of describing a general image with a single header, to give access to the various different types supported by digital cameras. The next problem is getting access to a camera: many frameworks exist for this purpose, but few work across systems or networks, or provide a configuration mechanism to the level which each camera supports. We outline a uniform access definition in Section 3.2 to capture images from any camera, regardless of the platform or camera location. This access method is designed to work with existing camera libraries, but allows specific drivers to be incorporated for greater control. The final issue is accessing any available camera, not just those local to the current machine. We define an addressing scheme which incorporates existing networking architectures into UCF to allow access to any camera on the network. Our proof of concept implementation (which will be made freely available) of UCF is called the All Seeing Eye, and is discussed in Section 4.

A previous version of the initial work (descriptor, uniform interface) has been presented as a work-in-progress[9] at a workshop. Numerous changes have been made to the abstraction of the descriptor and interface and as such this is a more complete version. This paper also includes a camera access scheme which has not been presented before.

2. PREVIOUS WORK

There have been a number of efforts to create standardized formats for device manufacturers such as IIDC 1394-based Digital Camera Specification[1] and VAPIX network camera communication specification[3]. IIDC standardizes access to camera devices that use FireWire as the camera-to-PC interconnect. The IIDC standard specifies the camera registers, fields within those registers, video formats, modes of operation and controls. VAPIX is a HTTP protocol developed by Axis Corporation for communication with network cameras via TCP/IP and the server client model. Using VAPIX the image data and camera configuration data in VAPIX are sent as HTTP commands to and from the camera device allowing uniform communication to any network device that implements VAPIX. Although these standard formats present a theoretically valid approach, a convergence of such standards by manufacturers is unlikely.

Video4Linux (V4L) is an example of a class of solutions that attempt to provide seamless access to source via a uniform interface[12]. V4L provides standardized access to video devices by including a kernel interface for video capture. This approach utilizes the Linux paradigm of treating all input and output communication as reads and writes to a file and presents imaging devices as file handlers to users. V4L defines standard types for devices and video properties, and provides functions for opening and closing devices, changing device properties, data formats, and input and output methods that are implemented via system calls. Using these defined types and methods, programmers have access to the sources that are installed on a particular machine. Although V4L provides an abstraction over specific camera protocols (e.g. IIDC) to the user quite effectively, it has two drawbacks. It is platform dependent and there is a barrier to adding support for new devices: in order to add support for a new device (or class of devices) a developer needs to write kernel drivers which is a cumbersome task and eliminates any hope of an opportunity for platform independency.

QuickTime is a media framework developed by Apple Inc. for managing and handling various multimedia requirements[11]. In addition to its ability to manage audio, animation and graphics, QuickTime provides functionality for capturing, processing, encoding, decoding and the delivery of video data through a framework called QTKit. QTKit's view of vision data is based on the concept of video clips or as QuickTime calls it 'movies'. QTKit provides a set of classes for accessing vision data from sources (capture devices and files) that provide high-level abstractions over the source's low-level details. QuickTime also provides a very comprehensive, high-level mechanism for decoding and encoding video between a large number of different formats. There are two limitations with QuickTime's approach with respect to vision based system development. The first issue is it does not provide a simple mechanism to retrieve the images from inside the system once they have been captured from the camera. The second limitation is that it is restricted to certain operating systems and so is not platform independent.

DirectShow[6] is a multimedia framework developed by Microsoft to provide a common interface for managing multimedia across many programming languages. DirectShow is an extensible filter-based framework that provides data capture, filtering, conversion and rendering of video and audio data. DirectShow interfaces with the Windows Driver Model in order to provide access to a large number of capture and filter devices. DirectShow insulates the application programmer from the details of accessing these devices; however, it also suffers from the same drawbacks as other multimedia frameworks as it uses its own image format and is not platform independent. Video4Linux, Quicktime and DirectShow are all platform dependent and do not provide any mechanism for data transport.

Java Media Framework (JMF)[7] is a cross-platform multimedia framework similar to QuickTime that provides capture, playback, streaming and transcoding of multimedia in a number of different formats for Java developers. The architecture of JMF consists of three stages: input, processing and output. The input stage provides routines for accessing video data from capture devices, files and network inputs. The processing stage deals with converting data using different codecs and adding common video effects. The output stage deals with rendering the video data, saving it to disk and sending the data via network. The fundamental limitations of JMF are similar to the QuickTime framework, in that it does not provide an abstraction over data transport.

The Open Computer Vision library (OpenCV)[4] is a comprehensive and widely used vision processing framework. The overall design of OpenCV relies on declaring data type definitions for vision entities and providing functions for operating on and extracting data from them. OpenCV pro-

Property	Type	Description of
Dimensions	32-bit integer	Width and height in pixels of the image
Frame Number	32-bit integer	Frame number from originating camera
Timecode	32-bit integer	Time at which the frame was captured (may be synchronized)
Synchronisation Number	32-bit integer	Stores the synchronisation code when using multiple cameras (can include camera group ID, camera number and sync code)
Pixel Format	8-bit enum	Encoding of pixel e.g. RGB, YUV422, floating-point
Number of Channels	8-bit integer	Number of channels per pixel in the image (3 for RGB, 4 for RGBA, etc.)
Bits per Channel	8-bit integer	The precision of each channel
Bits per Pixel	8-bit integer	The number of bits per pixel
Image Format	8-bit enum	Encoding of image in memory, e.g. JPG, PNG, raw
Origin	8-bit enum	Location of origin for indexing image
Size	32-bit integer	Size of the image data, not including this description
Total Size*	32-bit integer	Size of the image data including this description

* This value can be computed based on other values on image reception to save space.

Figure 1: This table represents the description of a general image, designed to accommodate as many types of image as possible without creating an extremely complicated structure.

vides a framework for accessing data from cameras installed on the system that utilizes an OS specific framework such as V4L, with support for multiple cameras although the authors had difficulty getting this to work. Limitations such as lack of support for distribution, multithreading, limited source access and image data manipulation, force developers to create custom frameworks (or utilize other existing frameworks) that employ OpenCV as a complementary framework.

Existing camera access frameworks usually define their own image formats instead of a description which can accept multiple formats, as outlined in Section 3.1. Additionally, data transport is often ignored when implementing vision system solutions. While all frameworks define an interface to access the cameras, they target only a subset of the available camera systems (usually entirely ignoring network cameras) instead of providing a uniform access interface which can retrieve images from any camera, as we demonstrate how to do in Section 3.2.

3. THE UNIFIED CAMERA FRAMEWORK

The contribution of this paper is the Unified Camera Framework (UCF), which contains a number of components to allow uniform and simple access to cameras, regardless of type. There are three levels of access required to provide an abstraction over source details:

1. Access to an image
2. Uniform access to a camera
3. Configuration of any camera up to its capabilities
4. Access to any addressable camera

Through these four access levels we provide solutions to the main problems encountered by computer vision researchers and developers when performing data capture from cameras: getting access to images through a generic image description; accessing and configuring any of the available cameras through a single interface; and using the same interface to access cameras which are not locally connected by integrating it with a transport mechanism. The immediate gain from these is uniform access to any camera on the local machine, the network or connected to a machine which is on the network. Access to multiple cameras simultaneously is supported (including synchronisation information). The camera access protocol uses a name server to redirect queries based in URI format to the camera device itself (which can be accessed using the uniform interface). This mechanism is performed by the UCF driver and is hidden from the user. The following three sections discuss the levels of access defined in the Unified Camera Framework.

3.1 Image Access

Our definition of image access is to provide a representation of many different formats which can be described succinctly. This allows applications to understand images in the native formats from various cameras, or to provide a description of a format for conversion. We present a generic image description to make image access transparent, to keep the highest quality image available, and to provide a simple means of image conversion. The image description must:

- Describe as many image formats as possible with a minimal description
- Hold information on synchronisation and timecodes (for multiple camera capture)
- Be extensible to allow for proprietary information and support of future image types

The description can be transported as a header along with the image data, and interpreted either by user code or by another system which supports the description. For a sequence of images the full header may only be sent once, and again only if the image type changes. A shortened header of frame number, timecode, synchronisation and size can be used for each individual image.

The motivation behind the generic description is to allow images from cameras supported through UCF to be received in the camera's native format. This is to maintain the highest possible level of quality until the point is reached for processing or, if needed, conversion. If a conversion is required then it is hoped that by using this mechanism the number of these is minimized; conversion between non-lossy formats can still result in quality degradation (e.g. RGB to HSV or YUV).

Providing a generic image description gives the first level of access. From the description applications can accept images in the native format of the camera and process in this format if supported. Our general image description is shown in Figure 1, along with the associated type of each item. The number of bits assigned to each item is conservative and could certainly be reduced. The general image description can also provide a layer of abstraction from image access. Given a set of components that support the UCF image description a developer need only pass the image as a whole among components. The details of the pixel types, image format etc. are hidden from the developer through the abstraction layer.

Some assumptions are made on the type of images produced by cameras. First, that camera data is supplied as discrete frames and those frames are rectangular in nature: while not true for range scanners, light field imagers etc. there is usually an acceptable mapping (e.g. spherical or cylindrical). Secondly, that individual pixels can be represented by a number of channels of a particular datatype. This type could be integer or floating-point, which again allows for data from range scanners, but also for a general raw image type using floating-point notation (common in HDR images or light maps for relighting constructed models in computer vision). Third, that if the image is compressed it is in one of the general formats such as JPEG or PNG (although others could be added). Finally, the image is exactly the dimensions stated, and the rows are not padded with extra bytes (e.g. to make it a multiple of four) when uncompressed.

The description is deliberately kept as small as possible, to try and represent the greatest number of image types with the smallest description. The width and height are stored as 16-bit integers within a single 32-bit value. The frame number, timecode and synchronisation number are provided mainly for use with multiple cameras, however the frame number and timecode could also be useful in non-synchronized systems (such as surveillance).

The pixel type of the image is described through a format (pre-defined, such as RGB or YUV422), the number of channels, the number of bits per channel and the number of bits per pixel. The format provides the pixel encoding under which the other values are interpreted, e.g. YUV422 has a different packing method to RGB. The number of bits per pixel is provided to allow padding of the pixel storage to round up to the nearest byte. This is to accommodate image types from HD or SLR cameras which can have a higher dynamic range such as 14 bits per pixel. For an RGB pixel, this is 42 bits: if the number of bits per pixel is set to 42, then there is no padding and each pixel will need to be extracted sequentially from the data; if the number of bits per pixel is set to 48 then each pixel is contained within 6 bytes, and can be addressed as such (and each channel extracted from the 48-bit value).

The image format is defined to allow for different methods of compression of the image data, such as JPEG or PNG. If the image is not compressed then the format is defined as 'raw'. The format may also indicate a progressive compression scheme such as MPEG; the precise setup of this would be done in the camera configuration (see Section 3.2). The same system would also deal with any digital rights management (DRM) the camera may use.

```
// Pseudo-code definition
base class Camera
  - String Name()
  - String Address()
  - String Driver()
  - Error Initialize()
  - Error Get(Image &, bool)
  - Error Query(Capabilities &)
  - Parameters Config()
  - Error Config(Parameters &)
```

Figure 2: Definition of the Camera class used to encapsulate the functionality of a basic camera. The name is ideally unique, such as a make/model plus serial number; the address is the URI as specified by the CAP in Section 3.3; the driver specifies how UCF is interacting with the camera; initialization is often required, especially for synchronization issues; the most often used method is Get(), which provides the latest image captured by the device; finally, the configuration parameters are specified through a query-based interface - this is extended within each camera driver to provide more control given a known camera type.

The origin is defined to take account of how the image should be indexed, although this could be extended to a full transformation description. The size of the image data is given, since although it can be calculated for raw images the size is not known for compressed images. The total size, that of the image and the description, is provided although this could be calculated separately. To save space a set of general descriptors could be defined such as RGB8 which would remove the need to specify the pixel format and properties. This may not be an important issue, as the descriptor is very small in comparison to a typical image size, and only needs to be sent once for every stream of images. However when the image is passed to the user each one will have a copy of the description (for possible use by other components or by the user).

3.2 Uniform Access

The next level of access beyond images is retrieval of those images from a camera. We have taken an object-oriented approach and encapsulated the basic functionality of a camera within a class. An abstract base class `Camera` is defined with the basic methods in Figure 2. Specific camera drivers can then be defined as derived classes and polymorphism can be applied to provide access to all camera types through a uniform interface.

Defining the camera interface this way also allows for different levels of access to a camera. Basic users who demand only an image and require very little configuration can simply use the base definition to access images from the requested camera. More advanced users can use the specific class definition to configure the camera or utilize some specific functionality. For example, Point Grey Research (PGR) cameras can be configured extensively while a basic webcam

Property	Description
Dimensions	Width and height of desired image
Frame rate	Frequency of image generation
Compression	Type and level of compression
Pixel type	Format and depth of pixel
Synchronization	E.g. timecode-based or frame-level

Figure 3: This table displays the parameters of a typical camera's configuration.

cannot. With uniform access the basic level access may be used for both or a PGR driver class may be used to access the advanced functionality of a PGR camera.

The base class definition provides for access to the camera name (supplied at camera setup), the camera ID (a UCF address, explained in the following section), access to image data, and basic camera configuration (parameters for configuration are shown in Figure 3). Not all cameras will support even this basic set of parameters, in which case the method will return an error code signifying which configuration request failed. The current configuration of a camera can also be queried through the `Config` method.

The base class interface is equivalent across platforms which allows the same code to run on multiple operating systems. Some definitions, such as that from PGR, are not cross-platform due to the direct support of the camera SDK (the FlyCapture SDK currently only supports Windows, although an upcoming version will also support Linux). However images from these cameras may still be accessed through the basic interface via the `Firewire` class.

Finally, one of the main advantages of this interface is that it provides an abstraction over the location of the camera: users can access cameras on the local machine or elsewhere on a network using the *Camera Access Protocol* (CAP) and UCF URLs to address the cameras. CAP is explained in detail in the following section.

3.3 The Camera Access Protocol

The last component of UCF is the camera access protocol, which provides mechanisms to access any addressable camera. We call the space of addressable cameras the *cameraverse*, and use a uniform resource identifier (URI) format to address these cameras.

CAP is an important contribution since it opens up access to cameras not just attached to the local machine but also any cameras attached to other machines which are accessible over the network. We use the definition of network here loosely, as it could mean TCP/IP over Ethernet, Bluetooth or some other connection, and so we assume that a network layer has been defined below UCF which may be used. An additional advantage to using CAP is that it provides an abstraction above programming language and platform, and so cameras can be accessed from any platform on any system which supports the UCF protocol.

The UCF URI is defined in the following format:

ucf://[gateway]/[sub-gateway]/[driver]/[camera ID]/

The possible values of the components in the URI are defined in Figure 4. This works in much the same way as a normal URL for the web, with a hostname, directory-like structure and then an identifier, although it supports a few

a) ucf://local/usb/camera3/

b) ucf://capturebox.university.edu/firewire/camera0/

c) ucf://capturebox.university.edu/group/multi1/

d) ucf://capturebox.university.edu/group/multi1/camera0/

e) ucf://floating/PGR/flea2/SN0375-0194/

f) ucf://camera.university.edu

Figure 5: Example URLs for addressing cameras.

more options. Our URI scheme name is "ucf" to identify the address as a unified camera framework destination.

The *gateway* provides the location of the machine which hosts the UCF camera. This may be a hostname or IP address for networked machines, or it can use the keyword *local* to signify the local machine. There is one additional option, *floating*: this mode can be used when the camera can be uniquely identified (through a serial number for example) and found through a discovery mechanism. This is especially useful when a camera needs to be connected to a different machine, since the URL for this camera does not change. Figure 5e shows an example URL for a floating camera.

The *sub-gateway* is an optional argument in the URI, and so does not need to be set for all camera addresses. However it introduces useful features such as addressing an entire group of cameras with a single address (which can be sent to a UCF name-server to retrieve the addresses of all cameras in the group). This is shown in the example in Figure 5c where the group *multi1* is addressed. Through this it is also possible to address specific cameras in the group, as shown in Figure 5d. Groups are set up by the user in advance, and provide a simple mechanism to work with multiple cameras. The sub-gateway also allows cameras to be addressed by their make or brand-name, after which they can be referenced by model and/or serial number.

The *driver* is used to specify the driver used by the host machine to access the camera, or the model of the camera if using the brand/make in the sub-gateway. This not only allows us to provide a unique path to the camera but also a user-readable URI which is meaningful, since the user can see how the camera is connected to that machine. The example values shown in Figure 4 cover the kind of connections used by most cameras. The examples in Figure 5a (local machine with usb camera) and Figure 5b (machine on a university network with a firewire camera) demonstrate the use of the accessor with connection type.

Finally, the *camera ID* identifies the specific camera being addressed which belongs to the previously defined gateway and accessor. This is required because machines often have more than one camera on a particular bus (e.g. three cameras on the firewire bus). This is also used for floating cameras to specify the serial number, providing a unique identifier with the make and model so that the machine is not required if auto-discovery is supported.

To apply UCF in a real-world setting there needs to be a name-server running which can translate URLs into connections to a camera. Each machine that hosts one or more

Addressor	Example Values	Describes
gateway	hostname, IP address, *local*, *floating*	Machine location
sub-gateway	*group*, brand, (optional argument, possibly not set)	Multiple cameras
driver	*usb*, *firewire*, *network*, *PGR*, *VAPIX*, model	Connection and driver type
camera ID	port number, serial number	Specific camera

Figure 4: Variations in the components of a UCF URI. Those values that are italicized represent keywords used in the URI, otherwise they are descriptive of the value used.

UCF cameras must run a name-server that performs two tasks: allow queries from local or external sources about the cameras currently connected locally or that the name-server knows about (such as network cameras), and facilitates connections to any of these cameras. Additionally a auto-discovery system is used for automatic update of camera connections on the network, and therefore automatic access to cameras given a unique floating URL.

The other option would be to have a UCF routing protocol, but this would require more low-level support. If UCF were to be extended and accepted as an addressing system in the future then it would need to be changed from sitting on top of a network infrastructure (as it is now designed to accommodate all existing cameras) to becoming a part of the infrastructure. If this was the case, then the example in Figure 5f could be used to address a UCF-supported camera directly and retrieve images.

4. SYSTEM ARCHITECTURE

We have developed an example implementation of the Unified Camera Framework which we call the *All Seeing Eye* (ASE). The architecture of the system is based on the definitions of UCF, with various additions to solve problems which arose during implementation.

Our system is based on a camera *Manager*, which administers cameras on the local system and provides access to the UCF name and image servers on the network. The Manager also operates as a UCF name server on the local system for other systems to access the cameras it administers. The implementation is based on a plug-in driver system. A driver in ASE is called an *Assistant Manager*, and provides basic methods to the Manager such as providing a list of cameras it can currently access as well as connection and disconnection of cameras (note that cameras in UCF are not capable of connection or disconnection, since this is outside the scope of a camera's operation). The Manager calls on all the Assistant Managers at its disposal to create a list of all locally accessible cameras which it can then provide to the user.

Using zeroconf (zero configuration) networking[14] the Manager is able to discover other UCF name servers and provide a list of accessible cameras on the network to the user. If a camera address is known, the Manager can then take the address and form a connection to that camera and provide the Camera interface back to the user (the ASE camera is identical to the UCF definition in Figure 2). Although the images being captured are transported over the network the interface is identical and so the networked operation is transparent to the user.

We have taken an object-oriented approach and encapsulated the basic functionality of a camera within a class. An abstract base class `Camera` is defined with the basic methods for access to a camera, summarized in Figure 2. Each camera currently being accessed by the user is represented locally by an instance which is accessed via polymorphism through the `Camera` class. Classes inherited from this base class implement the driver functionality of these basic methods for the type of camera they define. For example, a `Firewire` class could be defined to access all firewire cameras through the IIDC interface. An instance of this class would be returned to access a firewire camera, although the type returned to the user is still `Camera`. The `Driver` method in the definition of `Camera` returns the name of the inherited class (named after the driver by convention). Using this (or through prior knowledge from the UCF address) the instance can be dynamically cast to the actual object type. Then the user has full access to extended functionality defined within the derived class.

We have written drivers for various camera systems under ASE to access images from:

- VAPIX devices, e.g. Axis 206/207 IP cameras
- Windows supported devices, e.g. webcams or firewire, through DirectShow
- Mac OS X supported devices, e.g. webcams or firewire, through Sequence Grabber (SG is used to maintain low-level support and language consistency)
- Linux supported devices, e.g. webcams or firewire, through Video For Linux
- Point Grey Research cameras using FlyCapture drivers
- Cameras on Nokia phones (currently on N80, N82 and N95)

Various issues have arisen through the implementation of UCF. Initially we attempted to exclude the Initialize method and perform all initialization on connection. However, it was needed outside connection for the cameras to support synchronization, as some cameras (such as Point Grey Research) have additional initialization requirements which are simpler to support by keeping it separate. For our implementation we chose to have a multi-threaded based system for camera access, since it was impossible to maintain a correct frame number without this while grabbing images from the camera. The capture thread is also useful for synchronization and post-processing.

Supporting devices on different platforms is not trivial: we took advantage of other cross-platform libraries (such as boost) for access to network cameras, but a build-time library plug-in system was required to enable different cameras on different platforms. As such we currently have different versions of ASE depending on which cameras are being used. For example, to use PGR cameras the FlyCapture libraries are required, but if not using PGR cameras it does not make sense to include them. Ideally this will move to a dynamic plug-in architecture for future development, so that only a single library is required.

We have tested Axis 206/207 network cameras on Mac OS X, Windows and Linux, various iSight cameras on Mac OS X, Logitech Quickcams on Windows and Linux and mobile cameras on Nokia phones. Aside from the mobile platform, all have been tested at 640×480 and maintained 30Hz operation. The Axis cameras deliver 8-bit per channel RGB images in JPEG format; DirectShow on Windows delivers 8-bit per channel RGB RAW images; Sequence Grabber on OS X natively delivers YUV422 RAW images; all of these native formats are represented by the image descriptor, and on reception of the images our viewer (or writer) converts these to 8-bit per channel RGB RAW and displays them on the screen (or writes to disk). Higher resolutions are possible with the iSight camera (up to 1280×1024), still at 30Hz, and this is configurable with the ASE driver. The Nokia camera driver produces RAW RGB or JPEG files, and due to the limitations of the mobile platform, we have two operating modes: one which captures high resolution images at a low frame rate (1600×1200 at 1Hz) and another which captures low resolution at a higher frame rate (320×240 at 15Hz).

To access cameras across the network we chose to use the freely available vision transport system Hive[2] as the underlying network layer for ASE. By default we use port 2010 to represent "ucf" in our addressing scheme. Currently we can use the interface to access cameras on other machines using zeroconf networking to identify UCF name servers, which then provide the UCF address for each UCF image server attached to a camera. We are able to stream images from cameras attached to remote machines at full frame rate (30Hz) using Hive. Hive also deals with byte ordering to ensure the data arrives in the correct format. Hive operates on Windows, Mac OS X, Linux, Symbian and iPhone OS, therefore we can address cameras on any of these operating systems and transfer images to any other platform.

5. CONCLUSION

We have presented our novel camera access scheme, the Unified Camera Framework. This brings together four levels of access required for computer vision camera systems:

1. Image access through a descriptor which provides a mechanism to describe many different types of images, allowing for native high quality images to be passed on without resampling, and automated conversion of formats
2. Uniform access to cameras through a single interface on all platforms to devices either locally or on the network
3. Configuration through an extensible mechanism for tuning camera parameters up to each device's capabilities
4. The Camera Access Protocol, a novel scheme for addressing cameras and providing access to the cameraverse

The introduction of a uniform interface to camera access provides ease-of-use, portability and extensibility, and with the addition of the *Camera Access Protocol* (from which we can define the *cameraverse*) we also remove the restriction of a particular programming language or system architecture.

We intend to extend this framework to support many more cameras, and to provide an extended synchronisation layer to provide more support for multiple cameras. This implementation of UCF will be made publicly available (along with drivers to support ASE on the Hive transport system). This is to benefit the wider vision community and also in the hope that other researchers will help us add support for more cameras and systems.

6. REFERENCES

[1] 1394 Trade Association. IIDC 1394-based Digital Camera Specification. Technical Report 1.3, 1394 Trade Association, 2000.

[2] A. Afrah, G. Miller, D. Parks, M. Finke, and S. Fels. Hive: A distributed system for vision processing. In *Proc. 2nd International Conference on Distributed Smart Cameras*, September 2008.

[3] Axis Corporation. VAPIX API: http://www.axis.com/files/ manuals/VAPIX_3_HTTP_API_3_00.pdf, 2008.

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

[5] Camellia. http://camellia.sourceforge.net/.

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

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

[8] A. Makarenko, A. Brooks, , and T. Kaupp. On the benefits of making robotic software frameworks thin. In *International Conference on Intelligent Robots and Systems*, 2007.

[9] G. Miller and S. Fels. Uniform image and camera access. In *Workshop on the Applications of Computer Vision*. IEEE, December 2009.

[10] National Instruments LabView: http://www.ni.com/labview/.

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

[12] M. H. Schimek, B. Dirks, H. Verkuil, and M. Rubli. Video For Linux Two API Specification: http://v4l2spec.bytesex.org/v4l2spec/v4l2.pdf. Technical Report 0.24, Linux, 2008.

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

[14] Zeroconf. http://www.zeroconf.org/.