# Uniform Image and Camera Access

Gregor Miller and Sidney Fels
Human Communication Technologies Laboratory
University of British Columbia
2366 Main Mall, Vancouver, BC, Canada V6T 1Z4
{gregor,ssfels}@ece.ubc.ca

## Abstract

*We introduce a work-in-progress camera access scheme we call the* Unified Camera Framework. *Attempts have been made in the past to provide simple access to cameras, however these are generally OS specific or lacking in functionality. We present a novel interface which works across operating systems, and provides access to native images through a descriptor. 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.

## 1. Introduction

One of the most commonly performed tasks in computer vision is also one that is often overlooked: access to image data produced by cameras. There are various camera access solutions available, but they are either specific to a package [9], a particular camera type (IIDC or Point Grey Research) or operating system (Quicktime™ 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 central motivation of our work comes from the universal need for access to image data for the vision community and hobbyists, and the lack of a sufficient abstraction over existing technologies. We did not want to create another standard since it is difficult to establish support and unlikely to improve the current situation. Instead we have created an abstraction layer over image and camera access which supports existing standards, and can support new cameras through its extensible framework. In order to provide an access system with the necessary level of abstraction, the contributions of this work are: a generic image description; uniform access to cameras across operating systems and networks; and configuration of all cameras up to their capabilities.

Individually these contribute to the field of computer vision and to camera technology in general, however when combined they form a contribution to those not familiar with camera access. 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, 12, 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. |
| **Analyse** | : | 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™ 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 specialised 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 supply a mechanism to incorporate networking schemes into UCF to allow access to any camera on the network which is attached to a UCF server. This is designed to accommodate existing network technologies, so that cameras may be accessed on existing networks. 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.

## 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[11]. 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[10]. 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

| 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 synchronised) |
| 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.

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 provides 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

Through these three 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 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 minimised; 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-synchronised 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.

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

| 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.

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 for access to a camera, summarised 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 `Class` method in the definition of `Camera` returns the name of the inherited class. Using this (or through prior knowledge from the UCF URL - explained in the next section) 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.

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 class definition to access images from the requested camera. More advanced users can use the specific class definition to configure the camera or utilise some specific functionality. For example, Point Grey Research (PGR) cameras can be configured extensively while a basic webcam 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 URL, 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.

## 4. Example Implementation

We have developed an example implementation of the Unified Camera Framework which we call the *All Seeing Eye* (ASE). We have wrapped 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)
- Logitech cameras under Linux using Quickcam drivers
- Point Grey Research cameras using FlyCapture drivers
- Cameras on Nokia phones (currently on N80 and N95)

Various issues have arisen through the implementation of UCF. An additional method (Initialize) was needed 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 from the Connect method.

Supporting devices on different platforms is not completely trivial: we took advantage of other cross-platform libraries (such as boost) for getting access to network cameras, but a static / dynamic library plug-in system was required to enable different cameras. 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 we will move to a dynamic plug-in architecture for future development, so that only a single library is required.

We have tested frame rates on Axis 206/207 cameras, iSight on Mac OS X and Logitech Quickcams attached to Windows machines, at 640×480 and maintained 30Hz operation on all cameras. 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 natively delivers YUV422 RAW

```
// Pseudo-code definition of base class for Cameras
base class Camera
  - String Name()
  - String ID() // For network access
  - String Class() // inherited class (e.g. Axis, Windows)
  - Error Connect(ID &) // Allows simpler network integration
  - Error Disconnect()
  - Error Initialize() // Required for synchronization issues
  - Error Get(Image &, bool) // boolean to signal blocking call or not
  - Error Query(Capabilities &) // Discover capabilities of camera
  - Parameters Config()
  - Error Config(Parameters &)
```

Figure 2. Definition of the Camera class used to encapsulate the functionality of a basic camera.

images; All of these native formats are represented by the image description, and on reception of the images our viewer (or writer) converts these to 8-bit per channel RGB RAW and display them on the screen (or write to disk). Higher resolutions are possible with the iSight camera (up to $1280\times1024$), still at 30Hz, and this is configurable with the UCF driver. The Nokia camera driver produces RAW RGB files, however due to the limitations of the platform videos are not retrievable; instead a user must point-and-click to provide an image which is either stored locally on the camera or sent across the network.

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. Currently we can use the interface to access cameras on other machines using a hostname and port combination, and each camera on every machine has its own small server. 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.

## 5. Conclusion

We have presented our novel camera access scheme, the Unified Camera Framework. This brings together three levels of access required for computer vision and camera systems: image access, which provides a mechanism to describe many different formats of images within one framework; uniform access, where the same interface is used on every platform to access a camera either locally or on the network; and finally an extensible mechanism for configuring cameras up to their capability.

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. We also intend to develop an addressing scheme for cameras with an auto-discovery mechanism to enhance usability. 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.

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

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

[11] 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.

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