The Frankencamera: An Experimental Platform for Computational Photography

Andrew Adams[1] Eino-Ville Talvala[1] Sung Hee Park[1]
David E. Jacobs[1] Boris Ajdin[2] Natasha Gelfand[3]
Jennifer Dolson[1] Daniel Vaquero[3,4] Jongmin Baek[1]
Marius Tico[3] Hendrik P. A. Lensch[2] Wojciech Matusik[5]
Kari Pulli[3] Mark Horowitz[1] Marc Levoy[1]

[1]Stanford University
[2]Ulm University
[3]Nokia Research Center Palo Alto
[4]University of California, Santa Barbara
[5]Disney Research, Zürich

(a)

(b)

Figure 1: *Two implementations of the Frankencamera architecture:* ***(a)*** *The custom-built F2 – portable and self-powered, best for projects requiring flexible hardware.* ***(b)*** *A Nokia N900 with a modified software stack – a compact commodity platform best for rapid development and deployment of applications to a large audience.*

Abstract

Although there has been much interest in computational photography within the research and photography communities, progress has been hampered by the lack of a portable, programmable camera with sufficient image quality and computing power. To address this problem, we have designed and implemented an open architecture and API for such cameras: the Frankencamera. It consists of a base hardware specification, a software stack based on Linux, and an API for C++. Our architecture permits control and synchronization of the sensor and image processing pipeline at the microsecond time scale, as well as the ability to incorporate and synchronize external hardware like lenses and flashes. This paper specifies our architecture and API, and it describes two reference implementations we have built. Using these implementations we demonstrate six computational photography applications: HDR viewfinding and capture, low-light viewfinding and capture, automated acquisition of extended dynamic range panoramas, foveal imaging, IMU-based hand shake detection, and rephotography. Our goal is to standardize the architecture and distribute Frankencameras to researchers and students, as a step towards creating a community of photographer-programmers who develop algorithms, applications, and hardware for computational cameras.

CR Categories: I.4.1 [Image Processing and Computer Vision]: Digitization and Image Capture—Digital Cameras

Keywords: computational photography, programmable cameras

1 Introduction

Computational photography refers broadly to sensing strategies and algorithmic techniques that enhance or extend the capabilities of digital photography. Representative techniques include high dynamic range (HDR) imaging, flash-noflash imaging, coded aperture and coded exposure imaging, panoramic stitching, digital photomontage, and light field imaging [Raskar and Tumblin 2010].

Although interest in computational photography has steadily increased among graphics and vision researchers, few of these techniques have found their way into commercial cameras. One reason is that cameras are closed platforms. This makes it hard to incrementally deploy these techniques, or for researchers to test them in the field. Ensuring that these algorithms work robustly is therefore difficult, and so camera manufacturers are reluctant to add them to their products. For example, although high dynamic range (HDR) imaging has a long history [Mann and Picard 1995; Debevec and Malik 1997], the literature has not addressed the question of automatically deciding which exposures to capture, i.e., metering for HDR. As another example, while many of the drawbacks of flash photography can be ameliorated using flash-noflash imaging [Petschnigg et al. 2004; Eisemann and Durand 2004], these techniques produce visible artifacts in many photographic situations [Durand 2009]. Since these features do not exist in actual cameras, there is no strong incentive to address their artifacts.

Particularly frustrating is that even in platforms like smartphones, which encourage applet creation and have increasingly capable imaging hardware, the programming interface to the imaging system is highly simplified, mimicking the physical interface of a point-and-shoot camera. This is a logical interface for the manufacturer to include, since it is complete for the purposes of basic camera operations and stable over many device generations. Unfortunately, it means that in these systems it is not possible to create imaging applications that experiment with most areas of computational photography.

To address this problem, we describe a camera architecture and API flexible enough to implement most of the techniques proposed in the computational photography literature. We believe the architec-

ture is precise enough that implementations can be built and verified for it, yet high-level enough to allow for evolution of the underlying hardware and portability across camera platforms. Most importantly, we have found it easy to program for.

In the following section, we review previous work in this area, which motivates an enumeration of our design goals at the beginning of Section 3. We then describe our camera architecture in more detail, and our two reference implementations. The first platform, the F2 (Figure 1a), is composed of off-the-shelf components mounted in a laser-cut acrylic case. It is designed for extensibility. Our second platform (Figure 1b) is a Nokia N900 with a custom software stack. While less customizable than the F2, it is smaller, lighter, and readily available in large quantities. It demonstrates that current smartphones often have hardware components with more capabilities than their APIs expose. With these implementations in mind, we describe how to program for our architecture in Section 4. To demonstrate the capabilities of the architecture and API, we show six computational photography applications that cannot easily be implemented on current cameras (Section 5).

2 Prior Work

A digital camera is a complex embedded system, spanning many fields of research. We limit our review of prior work to camera platforms rather than their constituent algorithms, to highlight why we believe a new architecture is needed to advance the field of computational photography.

Consumer cameras. Although improvements in the features of digital SLRs have been largely incremental, point-and-shoot camera manufacturers are steadily expanding the range of features available on their cameras. Among these, the Casio EX-F1 stands out in terms of its computational features. This camera can capture bursts of images at 60 fps at a 6-megapixel resolution. These bursts can be computationally combined into a new image directly on the camera in a variety of ways. Unfortunately, the camera software cannot be modified, and thus no additional features can be explored by the research community.

In general, DSLR and point-and-shoot cameras use vendor-supplied firmware to control their operation. Some manufacturers such as Canon and Nikon have released software development kits (SDKs) that allow one to control their cameras using an external PC. While these SDKs can be useful for some computational photography applications, they provide a programming interface equivalent to the physical interface on the camera, with no access to lower layers such as metering or auto-focus algorithms. Furthermore, using these SDKs requires tethering the camera to a PC, and they add significant latency to the capture process.

Though the firmware in these cameras is always proprietary, several groups have successfully reverse-engineered the firmware for some Canon cameras. In particular, the Canon Hack Development Kit [CHD 2010] non-destructively replaces the original firmware on a wide range of Canon point-and-shoot cameras. Photographers can then script the camera, adding features such as custom burst modes, motion-triggered photography, and time-lapse photography. Similarly, the Magic Lantern project [mag 2010] provides enhanced firmware for Canon 5D Mark II DSLRs. While these projects remove both the need to attach a PC to the camera and the problem of latency, they yield roughly the same level of control as the official SDK: the lower levels of the camera are still a black box.

Smartphones are programmable cell phones that allow and even encourage third-party applications. The newest smartphones are capable of capturing still photographs and videos with quality comparable to point-and-shoot cameras. These models contain numerous input and output devices (e.g., touch screen, audio, buttons, GPS, compass, accelerometers), and are compact and portable. While these systems seem like an ideal platform for a computational camera, they provide limited interfaces to their camera subsystems. For example, the Apple iPhone 3GS, the Google Nexus One, and the Nokia N95 all have variable-focus lenses and high-megapixel image sensors, but none allow application control over absolute exposure time, or retrieval of raw sensor data – much less the ability to stream full-resolution images at the maximum rate permitted by the sensor. In fact, they typically provide less control of the camera than the DSLR camera SDKs discussed earlier. This lack of control, combined with the fixed sensor and optics, make these devices useful for only a narrow range of computational photography applications. Despite these limitations, the iPhone app store has several hundred third-party applications that use the camera. This confirms our belief that there is great interest in extending the capabilities of traditional cameras; an interest we hope to support and encourage with our architecture.

Smart cameras are image sensors combined with local processing, storage, or networking, and are generally used as embedded computer vision systems [Wolf et al. 2002; Bramberger et al. 2006]. These cameras provide fairly complete control over the imaging system, with the software stack, often built atop Linux, implementing frame capture, low-level image processing, and vision algorithms such as background subtraction, object detection, or object recognition. Example research systems are Cyclops [Rahimi et al. 2005], MeshEye [Hengstler et al. 2007], and the Philips wireless smart camera motes [Kleihorst et al. 2006]. Commercial systems include the National Instruments 17XX, Sony XCI-100, and the Basler eXcite series.

The smart cameras closest in spirit to our project are the CMUcam [Rowe et al. 2007] open-source embedded vision platform and the network cameras built by Elphel, Inc. [Filippov 2003]. The latter run Linux, have several sensor options (Aptina and Kodak), and are fully open-source. In fact, our earliest Frankencamera prototype was built around an Elphel 353 network camera. The main limitation of these systems is that they are not complete cameras. Most are tethered; few support synchronization with other I/O devices; and none contain a viewfinder or shutter button. Our first prototype streamed image data from the Elphel 353 over Ethernet to a Nokia N800 Internet tablet, which served as the viewfinder and user interface. We found the network latency between these two devices problematic, prompting us to seek a more integrated solution.

Our Frankencamera platforms attempt to provide everything needed for a practical computational camera: full access to the imaging system like a smart camera, a full user interface with viewfinder and I/O interfaces like a smartphone, and the ability to be taken outdoors, untethered, like a consumer camera.

3 The Frankencamera Architecture

Informed by our experiences programming for (and teaching with) smartphones, point-and-shoots, and DSLRs, we propose the following set of requirements for a Frankencamera:

1. Is handheld, self-powered, and untethered. This lets researchers take the camera outdoors and face real-world photographic problems.
2. Has a large viewfinder with a high-quality touchscreen to enable experimentation with camera user interfaces.
3. Is easy to program. To that end, it should run a standard operating system, and be programmable using standard languages,

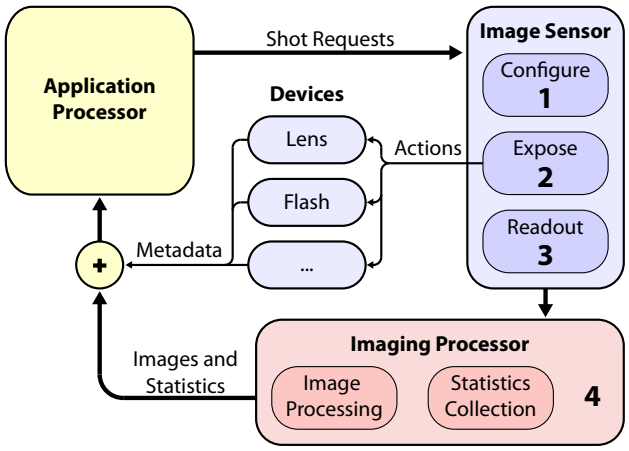

Figure 2: ***The Frankencamera Abstract Architecture.*** *The architecture consists of an application processor, a set of photographic devices such as flashes or lenses, and one or more image sensors, each with a specialized image processor. A key aspect of this system is that image sensors are pipelined. While the architecture can handle different levels of pipelining, most imaging systems have at least 4 pipeline stages, allowing for 4 frames in flight at a time: When the application is preparing to request frame n, the sensor is simultaneously configuring itself to capture frame $n-1$, exposing frame $n-2$, and reading out frame $n-3$. At the same time the fixed-function processing units are processing frame $n-4$. Devices such as the lens and flash perform actions scheduled for the frame currently exposing, and tag the frame leaving the pipeline with the appropriate metadata.*

libraries, compilers, and debugging tools.

4. Has the ability to manipulate sensor, lens, and camera settings on a per-frame basis at video rate, so we can request bursts of images with unique capture parameters for each image.

5. Labels each returned frame with the camera settings used for that frame, to allow for proper handling of the data produced by requirement 4.

6. Allows access to raw pixel values at the maximum speed permitted by the sensor interface. This means uncompressed, undemosaicked pixels.

7. Provides enough processing power in excess of what is required for basic camera operation to allow for the implementation of nearly any computational photography algorithm from the recent literature, and enough memory to store the inputs and outputs (often a burst of full-resolution images).

8. Allows standard camera accessories to be used, such as external flash or remote triggers, or more novel devices, such as GPS, inertial measurement units (IMUs), or experimental hardware. It should make synchronizing these devices to image capture straightforward.

Figure 2 illustrates our model of the imaging hardware in the Frankencamera architecture. It is general enough to cover most platforms, so that it provides a stable interface to the application designer, yet precise enough to allow for the low-level control needed to achieve our requirements. It encompasses the image sensor, the fixed-function imaging pipeline that deals with the resulting image data, and other photographic devices such as the lens and flash.

The Image Sensor. One important characteristic of our architecture is that the image sensor is treated as stateless. Instead, it is a pipeline that transforms requests into frames. The requests specify the configuration of the hardware necessary to produce the desired frame. This includes sensor configuration like exposure and gain, imaging processor configuration like output resolution and format, and a list of device actions that should be synchronized to exposure, such as if and when the flash should fire.

The frames produced by the sensor are queued and retrieved asynchronously by the application. Each one includes both the actual configuration used in its capture, and also the request used to generate it. The two may differ when a request could not be achieved by the underlying hardware. Accurate labeling of returned frames (requirement 5) is essential for algorithms that use feedback loops like autofocus and metering.

As the manager of the imaging pipeline, a sensor has a somewhat privileged role in our architecture compared to other devices. Nevertheless, it is straightforward to express multiple-sensor systems. Each sensor has its own internal pipeline and abstract imaging processor (which may be implemented as separate hardware units, or a single time-shared unit). The pipelines can be synchronized or allowed to run independently. Simpler secondary sensors can alternatively be encapsulated as devices (described later), with their triggering encoded as an action slaved to the exposure of the main sensor.

The Imaging Processor. The imaging processor sits between the raw output of the sensor and the application processor, and has two roles. First, it generates useful statistics from the raw image data, including a small number of histograms over programmable regions of the image, and a low-resolution sharpness map to assist with autofocus. These statistics are attached to the corresponding returned frame.

Second, the imaging processor transforms image data into the format requested by the application, by demosaicking, white-balancing, resizing, and gamma correcting as needed. As a minimum we only require two formats; the raw sensor data (requirement 6), and a demosaicked format of the implementation's choosing. The demosaicked format must be suitable for streaming directly to the platform's display for use as a viewfinder.

The imaging processor performs both these roles in order to relieve the application processor of essential image processing tasks, allowing application processor time to be spent in the service of more interesting applications (requirement 7). Dedicated imaging processors are able to perform these roles at a fraction of the compute and energy cost of a more general application processor.

Indeed, imaging processors tend to be fixed-functionality for reasons of power efficiency, and so these two statistics and two output formats are the only ones we require in our current architecture. We anticipate that in the longer term image processors will become more programmable, and we look forward to being able to replace these requirements with a programmable set of transformation and reduction stages. On such a platform, for example, one could write a "camera shader" to automatically extract and return feature points and descriptors with each frame to use for alignment or structure from motion applications.

Devices. Cameras are much more than an image sensor. They also include a lens, a flash, and other assorted devices. In order to facilitate use of novel or experimental hardware, the requirements the architecture places on devices are minimal.

Devices are controllable independently of a sensor pipeline by

whatever means are appropriate to the device. However, in many applications the timing of device actions must be precisely coordinated with the image sensor to create a successful photograph. The timing of a flash firing in second-curtain sync mode must be accurate to within a millisecond. More demanding computational photography applications, such as coded exposure photography [Raskar et al. 2006], require even tighter timing precision.

To this end, devices may also declare one or more actions they can take synchronized to exposure. Programmers can then schedule these actions to occur at a given time within an exposure by attaching the action to a frame request. Devices declare the latency of each of their actions, and receive a callback at the scheduled time minus the latency. In this way, any event with a known latency can be accurately scheduled.

Devices may also tag returned frames with metadata describing their state during that frame's exposure (requirement 5). Tagging is done after frames leave the imaging processor, so this requires devices to keep a log of their recent state.

Some devices generate asynchronous events, such as when a photographer manually zooms a lens, or presses a shutter button. These are time-stamped and placed in an event queue, to be retrieved by the application at its convenience.

Discussion. While this pipelined architecture is simple, it expresses the key constraints of real camera systems, and it provides fairly complete access to the underlying hardware. Current camera APIs model the hardware in a way that mimics the physical camera interface: the camera is a stateful object, which makes blocking capture requests. This view only allows one active request at a time and reduces the throughput of a camera system to the reciprocal of its latency – a fraction of its peak throughput. Streaming modes, such as those used for electronic viewfinders, typically use a separate interface, and are mutually exclusive with precise frame level control of sensor settings, as camera state becomes ill-defined in a pipelined system. Using our pipelined model of a camera, we can implement our key architecture goals with a straightforward API. Before we discuss the API, however, we will describe our two implementations of the Frankencamera architecture.

3.1 The F2

Our first Frankencamera implementation is constructed from an agglomeration of off-the-shelf components (thus 'Frankencamera'). This makes duplicating the design easy, reduces the time to construct prototypes, and simplifies repair and maintenance. It is the second such major prototype (thus 'F2'). The F2 is designed to closely match existing consumer hardware, making it easy to move our applications to mass-market platforms whenever possible. To this end, it is built around the Texas Instruments OMAP3430 System-on-a-Chip (SoC), which is a widely used processor for smartphones. See Figure 3 for an illustration of the parts that make up the F2.

The F2 is designed for extensibility along three major axes. First, the body is made of laser-cut acrylic and is easy to manufacture and modify for particular applications. Second, the optics use a standard Canon EOS lens mount, making it possible to insert filters, masks, or microlens arrays in the optical path of the camera. Third, the F2 incorporates a Phidgets [Greenberg and Fitchett 2001] controller, making it extendable with buttons, switches, sliders, joysticks, camera flashes, and other electronics.

The F2 uses Canon lenses attached to a programmable lens controller. The lenses have manual zoom only, but have programmable

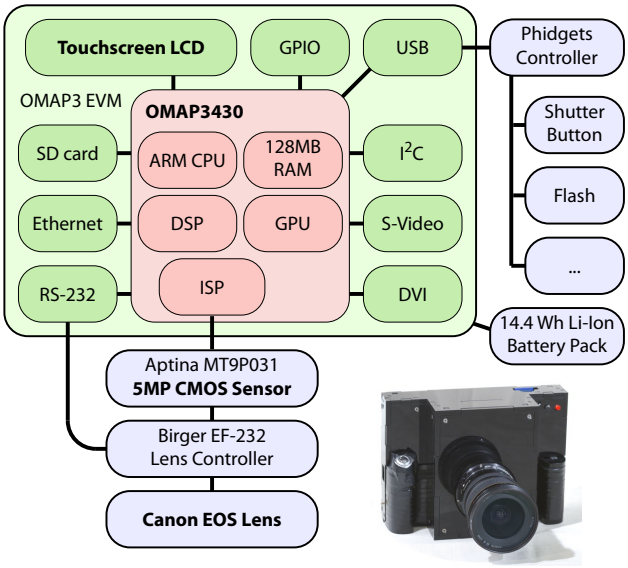

Figure 3: ***The F2.*** *The F2 implementation of the Frankencamera architecture is built around an OMAP3 EVM board, which includes the Texas Instruments OMAP3430 SoC, a* 640 × 480 *touchscreen LCD, and numerous I/O connections. The OMAP3430 includes a fixed-function imaging processor, an ARM Cortex-A8 CPU, a DSP, a PowerVR GPU supporting OpenGL ES 2.0, and 128MB of RAM. To the EVM we attach: a lithium polymer battery pack and power circuitry; a Phidgets board for controlling external devices; a five-megapixel CMOS sensor; and a Birger EF-232 lens controller that accepts Canon EOS lenses. The key strengths of the F2 are the extensibility of its optics, electronics, and physical form factor.*

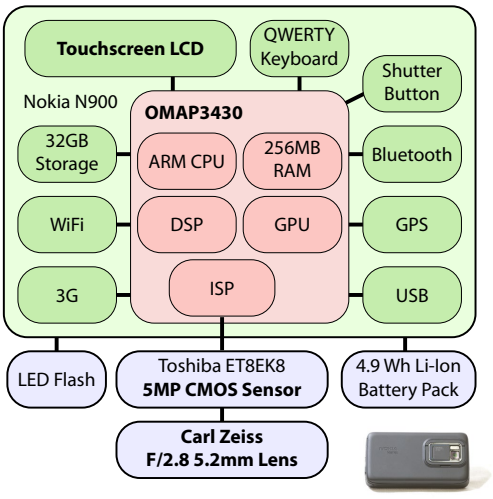

Figure 4: ***The Nokia N900.*** *The Nokia N900 incorporates similar electronics to the F2, in a much smaller form factor. It uses the same OMAP3430 SoC, an* 800 × 480 *touchscreen LCD, and numerous wireless connectivity options. The key strengths of the N900 are its small size and wide availability.*

aperture and focus. It uses a five-megapixel Aptina MT9P031 image sensor, which, in addition to the standard settings offers programmable region-of-interest, subsampling, and binning modes. It can capture full-resolution image data at 11 frames per second, or VGA resolution at up to 90 frames per second. The F2 can mount one or more Canon or Nikon flash units, which are plugged in over the Phidgets controller. As we have not reverse-engineered any flash communication protocols, these flashes can merely be triggered at the present time.

In the F2, the role of abstract imaging processor is fulfilled by the ISP within the OMAP3430. It is capable of producing raw or YUV 4:2:2 output. For each frame, it also generates up to four image histograms over programmable regions, and produces a 16×12 sharpness map using the absolute responses of a high-pass IIR filter summed over each image region. The application processor in the F2 runs the Ångström Linux distribution [Ang 2010]. It uses high-priority real-time threads to schedule device actions with a typical accuracy of ± 20 microseconds.

The major current limitation of the F2 is the sensor size. The Aptina sensor is 5.6mm wide, which is a poor match for Canon lenses intended for sensors 23-36mm wide. This restricts us to the widest-angle lenses available. Fortunately, the F2 is designed to be easy to modify and upgrade, and we are currently engineering a DSLR-quality full-frame sensor board for the F2 using the Cypress Semiconductor LUPA 4000 image sensor, which has non-destructive readout of arbitrary regions of interest and extended dynamic range.

Another limitation of the F2 is that while the architecture permits a rapidly alternating output resolution, on the OMAP3430 this violates assumptions deeply encoded in the Linux kernel's memory management for video devices. This forces us to do a full pipeline flush and reset on a change of output resolution, incurring a delay of roughly 700ms. This part of the Linux kernel is under heavy development by the OMAP community, and we are optimistic that this delay can be substantially reduced in the future.

3.2 The Nokia N900

Our second hardware realization of the Frankencamera architecture is a Nokia N900 with a custom software stack. It is built around the same OMAP3430 as the F2, and it runs the Maemo Linux distribution [Mae 2010]. In order to meet the architecture requirements, we have replaced key camera drivers and user-space daemons. See Figure 4 for a description of the camera-related components of the Nokia N900.

While the N900 is less flexible and extensible than the F2, it has several advantages that make it the more attractive option for many applications. It is smaller, lighter, and readily available in large quantities. The N900 uses the Toshiba ET8EK8 image sensor, which is a five-megapixel image sensor similar to the Aptina sensor used in the F2. It can capture full-resolution images at 12 frames per second, and VGA resolution at 30 frames per second. While the lens quality is lower than the Canon lenses we use on the F2, and the aperture size is fixed at f/2.8, the low mass of the lens components means they can be moved very quickly with a programmable speed. This is not possible with Canon lenses. The flash is an ultra-bright LED, which, while considerably weaker than a xenon flash, can be fired for a programmable duration with programmable power.

The N900 uses the same processor as the F2, and a substantially similar Linux kernel. Its imaging processor therefore has the same capabilities, and actions can be scheduled with equivalent accuracy. Unfortunately, this also means the N900 has the same resolution switching cost as the F2. Nonetheless, this cost is significantly less than the resolution switching cost for the built-in imaging API (GStreamer), and this fact is exploited by several of our applications.

On both platforms, roughly 80MB of free memory is available to the application programmer. Used purely as image buffer, this represents eight 5-MP images, or 130 VGA frames. Data can be written to permanent storage at roughly 20 MB/sec.

4 Programming the Frankencamera

Developing for either Frankencamera is similar to developing for any Linux device. One writes standard C++ code, compiles it with a cross-compiler, then copies the resulting binary to the device. Programs can then be run over ssh, or launched directly on the device's screen. Standard debugging tools such as gdb and strace are available. To create a user interface, one can use any Linux UI toolkit. We typically use Qt, and provide code examples written for Qt. OpenGL ES 2.0 is available for hardware-accelerated graphics, and regular POSIX calls can be used for networking, file I/O, synchronization primitives, and so on. If all this seems unsurprising, then that is precisely our aim.

Programmers and photographers interact with our architecture using the "FCam" API. We now describe the API's basic concepts illustrated by example code. For more details, please see the API documentation and example programs included as supplemental material.

4.1 Shots

The four basic concepts of the FCam API are *shots*, *sensors*, *frames*, and *devices*. We begin with the shot. A *shot* is a bundle of parameters that completely describes the capture and post-processing of a single output image. A shot specifies sensor parameters such as gain and exposure time (in microseconds). It specifies the desired output resolution, format (raw or demosaicked), and memory location into which to place the image data. It also specifies the configuration of the fixed-function statistics generators by specifying over which regions histograms should be computed, and at what resolution a sharpness map should be generated. A shot also specifies the total time between this frame and the next. This must be at least as long as the exposure time, and is used to specify frame rate independently of exposure time. Shots specify the set of actions to be taken by devices during their exposure (as a standard STL set). Finally, shots have unique ids auto-generated on construction, which assist in identifying returned frames.

The example code below configures a shot representing a VGA resolution frame, with a 10ms exposure time, a frame time suitable for running at 30 frames per second, and a single histogram computed over the entire frame:

```
Shot shot;
shot.gain = 1.0;
shot.exposure  = 10000;
shot.frameTime = 33333;
shot.image = Image(640, 480, UYVY);
shot.histogram.regions = 1;
shot.histogram.region[0] = Rect(0, 0, 640, 480);
```

4.2 Sensors

After creation, a *shot* can be passed to a *sensor* in one of two ways – by *capturing* it or by *streaming* it. If a sensor is told to *capture* a configured shot, it pushes that shot into a request queue at the top of the imaging pipeline (Figure 2) and returns immediately:

```
Sensor sensor;
sensor.capture(shot);
```

The sensor manages the entire pipeline in the background. The shot is issued into the pipeline when it reaches the head of the request queue, and the sensor is ready to begin configuring itself for the next frame. If the sensor is ready, but the request queue is empty, then a bubble necessarily enters the pipeline. The sensor cannot simply pause until a shot is available, because it has several other pipeline stages; there may be a frame currently exposing, and another currently being read out. Bubbles configure the sensor to use the minimum frame time and exposure time, and the unwanted image data produced by bubbles is silently discarded.

Bubbles in the imaging pipeline represent wasted time, and make it difficult to guarantee a constant frame rate for video applications. In these applications, the imaging pipeline must be kept full. To prevent this responsibility from falling on the API user, the sensor can also be told to *stream* a shot. A shot to be streamed is copied into a holding slot alongside the request queue. Then whenever the request queue is empty, and the sensor is ready for configuration, a copy of the contents of the holding slot enters the pipeline instead of a bubble. Streaming a shot is done using: `sensor.stream(shot)`.

Sensors may also capture or stream vectors of shots, or *bursts*, in the same way that they capture or stream shots. Capturing a burst enqueues those shots at the top of the pipeline in the order given, and is useful, for example, to capture a full high-dynamic-range stack in the minimum amount of time. As with a shot, streaming a burst causes the sensor to make an internal copy of that burst, and atomically enqueue all of its constituent shots at the top of the pipeline whenever the sensor is about to become idle. Thus, bursts are atomic – the API will never produce a partial or interrupted burst. The following code makes a burst from two copies of our shot, doubles the exposure of one of them, and then uses the sensor's stream method to create frames that alternate exposure on a per-frame basis at 30 frames per second. The ability to stream shots with varying parameters at video rate is vital for many computational photography applications, and hence was one of the key requirements of our architecture. It will be heavily exploited by our applications in Section 5.

```
std::vector<Shot> burst(2);
burst[0] = shot;
burst[1] = shot;
burst[1].exposure = burst[0].exposure*2;
sensor.stream(burst);
```

To update the parameters of a shot or burst that is currently streaming (for example, to modify the exposure as the result of a metering algorithm), one merely modifies the shot or burst and calls *stream* again. Since the shot or burst in the internal holding slot is atomically replaced by the new call to *stream*, no partially updated burst or shot is ever issued into the imaging pipeline.

4.3 Frames

On the output side, the sensor produces *frames*, retrieved from a queue of pending frames via the *getFrame* method. This method is the only blocking call in the core API. A frame contains image data, the output of the statistics generators, the precise time the exposure began and ended, the actual parameters used in its capture, and the requested parameters in the form of a copy of the shot used to generate it. If the sensor was unable to achieve the requested parameters (for example, if the requested frame time was shorter than the requested exposure time), then the actual parameters will reflect the modification made by the system.

Frames can be identified by the `id` field of their shot. Being able to reliably identify frames is another of the key requirements for our architecture. The following code displays the longer exposure of the two frames specified in the burst above, but uses the shorter of the two to perform metering. The functions `displayImage` and `metering` are hypothetical functions that are not part of the API.

```
while (1) {
  Frame::Ptr frame = sensor.getFrame();
  if (frame.shot().id == burst[1].id) {
    displayImage(frame.image);
  } else if (frame.shot().id == burst[0].id) {
    unsigned newExposure = metering(frame);
    burst[0].exposure = newExposure;
    burst[1].exposure = newExposure*2;
    sensor.stream(burst);
  }
}
```

In simple programs it is typically not necessary to check the ids of returned frames, because our API guarantees that exactly one frame comes out per shot requested, in the same order. Frames are never duplicated or dropped entirely. If image data is lost or corrupted due to hardware error, a frame is still returned (possibly with statistics intact), with its image data marked as invalid.

4.4 Devices

In our API, each device is represented by an object with methods for performing its various functions. Each device may additionally define a set of *actions* which are used to synchronize these functions to exposure, and a set of *tags* representing the metadata attached to returned frames. While the exact list of devices is platform-specific, the API includes abstract base classes that specify the interfaces to the lens and the flash.

The lens. The lens can be directly asked to initiate a change to any of its three parameters: focus (measured in diopters), focal length, and aperture, with the methods `setFocus`, `setZoom`, and `setAperture`. These calls return immediately, and the lens starts moving in the background. For cases in which lens movement should be synchronized to exposure, the lens defines three actions to do the same. Each call has an optional second argument that specifies the speed with which the change should occur. Additionally, each parameter can be queried to see if it is currently changing, what its bounds are, and its current value. The following code moves the lens from its current position to infinity focus over the course of two seconds.

```
Lens lens;
float speed = (lens.getFocus()-lens.farFocus())/2;
lens.setFocus(lens.farFocus(), speed);
```

A lens tags each returned frame with the state of each of its three parameters during that frame. Tags can be retrieved from a frame like so:

```
Frame::Ptr frame = sensor.getFrame();
Lens::Tags *tags = frame->tags(&lens);
cout << "The lens was at: " << tags->focus;
```

The flash. The flash has a single method that tells it to fire with a specified brightness and duration, and a single action that does the same. It also has methods to query bounds on brightness and duration. Flashes with more capabilities (such as the strobing flash in Figure 5) can be implemented as subclasses of the base flash class. The flash tags each returned frame with its state, indicating whether it fired during that frame, and if so with what parameters.

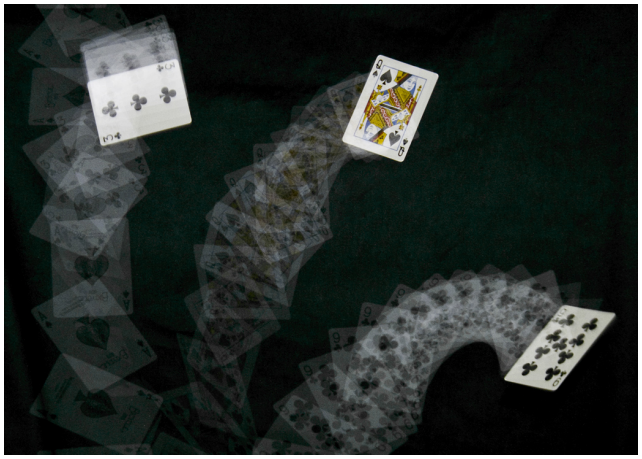

Figure 5: *The Frankencamera API provides precise timing control of secondary devices like the flash. To produce the image above, two Canon flash units were mounted on an F2. The weaker of the two was strobed for the entire one-second exposure, producing the card trails. The brighter of the two was fired once at the end of the exposure, producing the crisp images of the three cards.*

The following code example adds an action to our shot to fire the flash briefly at the end of the exposure (second-curtain sync). The results of a similar code snippet run on the F2 can be seen in Figure 5.

```
Flash flash;
Flash::FireAction fire(&flash);
fire.brightness = flash.maxBrightness();
fire.duration = 5000;
fire.time = shot.exposure - fire.duration;
shot.actions.insert(&fire);
```

Other devices. Incorporating external devices and having our API manage the timing of their actions is straightforward. One need merely inherit from the Device base class, add methods to control the device in question, and then define any appropriate actions, tags, and events. This flexibility is critical for computational photography, in which it is common to experiment with novel hardware that affects image capture.

4.5 Included Algorithms

There are occasions when a programmer will want to implement custom metering and autofocus algorithms, and the API supports this. For example, when taking a panorama, it is wise to not vary exposure by too much between adjacent frames, and the focus should usually be locked at infinity. In the common case, however, generic metering and autofocus algorithms are helpful, and so we include them as convenience functions in our API.

Metering. Our metering algorithm operates on the image histogram, and attempts to maximize overall brightness while minimizing the number of oversaturated pixels. It takes a pointer to a shot and a frame, and modifies the shot with suggested new parameters.

Autofocus. Our autofocus algorithm consists of an autofocus helper object, which is passed a reference to the lens and told to initiate autofocus. It then begins sweeping the lens from far focus to near focus. Subsequent frames should be fed to it, and it inspects their sharpness map and the focus position tag the lens has placed on them. Once the sweep is complete, or if sharpness degrades for several frames in a row, the lens is moved to the sharpest position found. While this algorithm is more straightforward than an iterative algorithm, it terminates in at most half a second, and is quite robust.

Image Processing. Once the images are returned, programmers are free to use any image processing library they like for analysis and transformation beyond that done by the image processor. Being able to leverage existing libraries is a major advantage of writing a camera architecture under Linux. For convenience, we provide methods to synchronously or asynchronously save raw files to storage (in the DNG format [Adobe, Inc. 2010]), and methods to demosaic, gamma correct, and similarly store JPEG images.

4.6 Implementation

In our current API implementations, apart from fixed-function image processing, FCam runs entirely on the ARM CPU in the OMAP3430, using a small collection of user-space threads and modified Linux kernel modules (See Figure 6 for the overall software stack). Our system is built on top of Video for Linux 2 (V4L2) – the standard Linux kernel video API. V4L2 treats the sensor as stateful with no guarantees about timing of parameter changes. To provide the illusion of a stateless sensor processing stateful *shots*, we use three real-time-priority threads to manage updates to image sensor parameters, readback of image data and metadata, and device actions synchronized to exposure.

Setting Sensor Parameters. The "Setter" thread is responsible for sensor parameter updates. The timing of parameter changes is specific to the image sensor in question: On the F2, this thread sets all the parameters for frame $n+2$ just after the readout of frame n begins. On the N900, parameters must be set in two stages. When readout of frame n begins, exposure and frame time are set for frame $n+2$, and parameters affecting readout and post-processing are set for frame $n+1$. Once all of a shot's parameters are set, the Setter predicts when the resulting V4L2 buffer will return from the imaging pipeline, and pushes the annotated shot onto an internal "in-flight" queue. To synchronize the Setter thread with frame readout, we add a call to the imaging pipeline driver which sleeps the calling thread until a hardware interrupt for the start of the next frame readout arrives.

Our image sensor drivers are standard V4L2 sensor drivers with one important addition. We add controls to specify the time taken by each individual frame, which are implemented by adjusting the amount of extra vertical blanking in sensor readout.

Handling Image Data. The "Handler" thread runs at a slightly lower priority. It receives the V4L2 image buffers produced by the imaging pipeline, which consist of timestamped image data. This timestamp is correlated with the predicted return times for the shots in flight, in order to match each image with the shot that produced it. The Handler then queries the imaging processor driver for any requested statistics. These are also timestamped, and so can similarly be matched to the appropriate shot. The image data from the buffer is then copied into the frame's desired memory target (or discarded), and the completed FCam frame is placed in the frame queue, ready to be retrieved by the application.

Scheduling Device Actions. The "Action" thread runs at the highest priority level and manages the timing of scheduled actions. Actions are scheduled by the Setter when it sets a frame's exposure

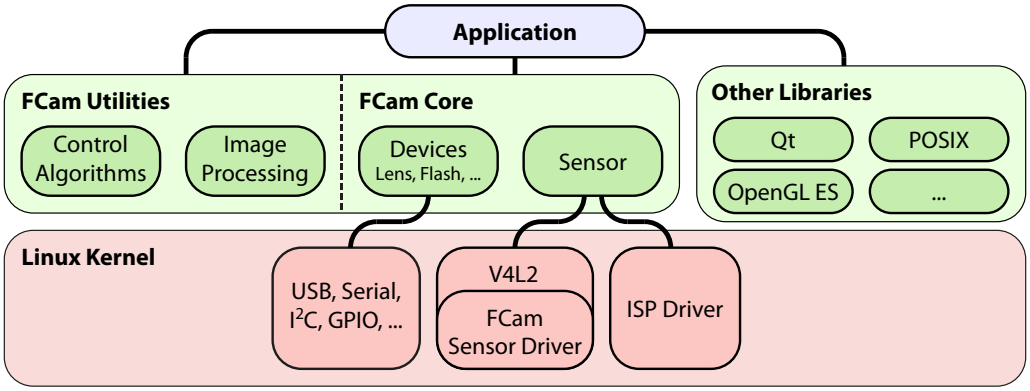

Figure 6: ***The Frankencamera Software Stack.*** *The core of the Frankencamera API (FCam) consists of the sensor object, and various device objects for lenses, flashes, buttons, etc. The sensor object has three tasks: controlling a custom video for Linux (V4L2) sensor driver, which manages and synchronizes sensor state updates and frame timing; managing the imaging processor (ISP), configuring fixed-function image processing units and gathering the resulting statistics and image data; and precise timing of device actions slaved to exposures (e.g., firing the flash). Each task is performed in its own real-time priority thread. The API also includes utility functions to save images in processed or raw formats, and helper functions for autofocus and metering. For other functionality, such as file I/O, user interfaces, and rendering, the programmer may use any library available for Linux.*

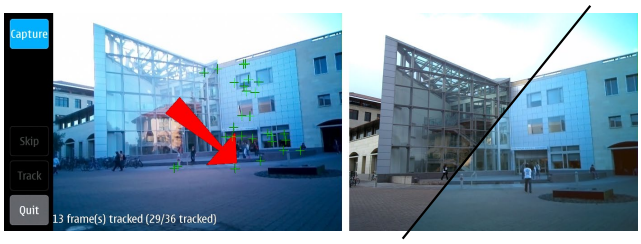

Figure 7: ***Rephotography****. A Frankencamera platform lets us experiment with novel capture interfaces directly on the camera. Left: The rephotography application directs the user towards the location from which a reference photograph was taken (by displaying a red arrow on the viewfinder). Right: The reference photograph (above and left), which was taken during the morning, overlaid on the image captured by the rephotography application several days later at dusk.*

time, as this is the earliest time at which the actions' absolute trigger time is known. The Action thread sleeps until several hundred microseconds before the trigger time of the next scheduled action, busy waits until the trigger time, then fires the action. We find that simply sleeping until the trigger time is not sufficiently accurate. By briefly busy-waiting we sacrifice a small amount of CPU time, but are able to schedule actions with an accuracy of ± 20 microseconds.

Performance. The FCam runtime, with its assortment of threads, uses 11% of the CPU time on the OMAP3430's ARM core when streaming 640×480 frames at 30 frames per second. If image data is discarded rather than copied, usage drops to 5%. Basic camera operations like displaying the viewfinder on screen, metering, and focusing do not measurably increase CPU usage.

Installation. Setting up a store-bought N900 for use with the FCam API involves installing a package of FCam kernel modules, and rebooting. It does not interfere with regular use of the device or its built-in camera application.

4.7 Discussion

Our goals for the API were to provide intuitive mechanisms to precisely manipulate camera hardware state over time, including control of the sensor, fixed-function processing, lens, flash, and any associated devices. We have accomplished this in a minimally surprising manner, which should be a key design goal of any API. The API is limited in scope to what it does well, so that programmers can continue to use their favorite image processing library, UI toolkit, file I/O, and so on. Nonetheless, we have taken a "batteries included" approach, and made available control algorithms for metering and focus, image processing functions to create raw and JPEG files, and example applications that demonstrate using our API with the Qt UI toolkit and OpenGL ES.

Implementing the API on our two platforms required a shadow pipeline of in-flight shots, managed by a collection of threads, to fulfill our architecture specification. This makes our implementation brittle in two respects. First, an accurate timing model of image sensor and imaging processor operation is required to correctly associate output frames with the shot that generated them. Second, deterministic guarantees from the image sensor about the latency of parameter changes are required, so we can configure the sensor correctly. In practice, there is a narrow time window in each frame during which sensor settings may be adjusted safely. To allow us to implement our API more robustly, future image sensors should provide a means to identify every frame they produce on both the input and output sides. Setting changes could then be requested to take effect for a named future frame. This would substantially reduce the timing requirements on sensor configuration. Image sensors could then return images tagged with their frame id (or even the entire sensor state), to make association of image data with sensor state trivial.

It would also be possible to make the API implementation more robust by using a real-time operating system such as RTLinux [Yodaiken 1999], which would allow us to specify hard deadlines for parameter changes and actions. However, this limits the range of devices on which the API can be deployed, and our applications to date have not required this level of control. In cases with a larger number of device actions that must be strictly synchronized, an implementation of the API on a real-time operating system might be preferable.

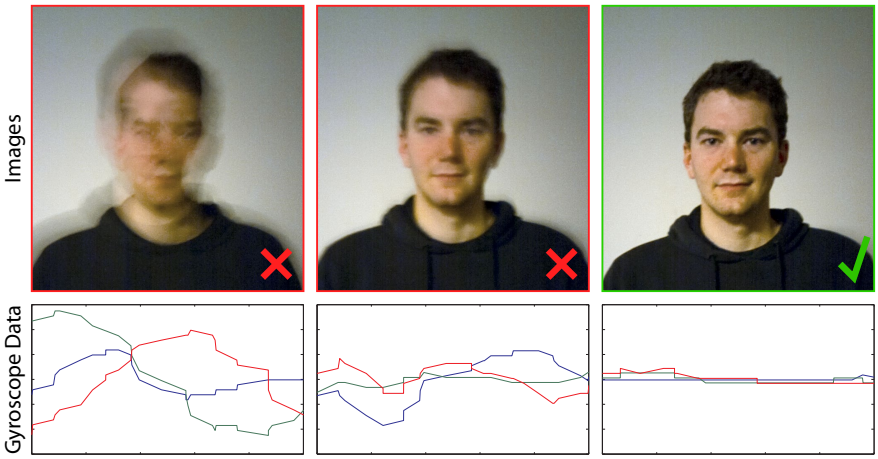

Figure 8: ***Lucky Imaging****. An image stream and 3-axis gyroscope data for a burst of three images with 0.5 second exposure times. The Frankencamera API makes it easy to tag image frames with the corresponding gyroscope data. For each returned frame, we analyze the gyroscope data to determine if the camera was moving during the exposure. In the presence of motion, the gyroscope values become non-linear. Only the frames determined to have low motion are saved to storage.*

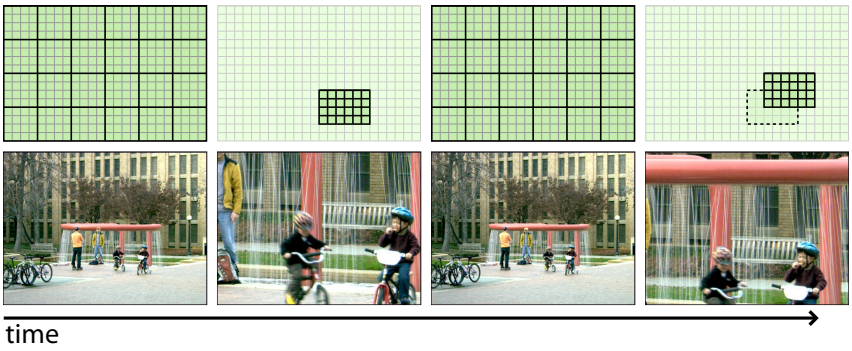

Figure 9: ***Foveal Imaging*** *records a video stream that alternates between a downsampled view of the whole scene and full-detail insets of a small region of interest. The inset can be used to record areas of high detail, track motion, or gather texture samples for synthesizing a high-resolution video. In this example, the inset is set to scan over the scene, the region of interest moving slightly between each pair of inset frames.*

5 Applications

We now describe a number of applications of the Frankencamera architecture and API to concrete problems in photography. Most run on either the N900 or the F2, though some require hardware specific to one platform or the other. These applications are representative of the types of in-camera computational photography our architecture enables, and several are also novel applications in their own right. They are all either difficult or impossible to implement on existing platforms, yet simple to implement under the Frankencamera architecture.

5.1 Rephotography

We reimplement the system of Bae et al. [2010], which guides a user to the viewpoint of a historic photograph for the purpose of recapturing the same shot. The user begins by taking two shots of the modern scene from different viewpoints, which creates a baseline for stereo reconstruction. SIFT keypoints are then extracted from each image. Once correspondences are found, the relative camera pose can be found by the 5-point algorithm, together with a consensus algorithm such as RANSAC, for rejecting outliers. Once the SIFT keypoints have been triangulated, the inliers are tracked through the streaming viewfinder frames using a KLT tracker, and the pose of the camera is estimated in real time from the updated positions of the keypoints. The pose of the historic photograph is pre-computed likewise, and the user is directed to move the camera towards it via a visual interface, seen in Figure 7 on the left. A sample result can be seen on the right.

In the original system by Bae et al., computations and user interactions take place on a laptop, with images provided by a tethered Canon DSLR, achieving an interactive rate of 10+ fps. In our implementation on the N900, we achieve a frame rate of 1.5 fps, handling user interaction more naturally through the touchscreen LCD of the N900. Most of the CPU time is spent detecting and tracking keypoints (whether KLT or SIFT). This application and applications like it would benefit immensely from the inclusion of a hardware-accelerated feature detector in the imaging pipe.

5.2 IMU-based Lucky Imaging

Long-exposure photos taken without use of a tripod are usually blurry, due to natural hand shake. However, hand shake varies over time, and a photographer can get "lucky" and record a sharp photo if the exposure occurs during a period of stillness (Figure 8). Our "Lucky Imaging" application uses an experimental Nokia 3-axis gyroscope affixed to the front of the N900 to detect hand shake. Utilizing a gyroscope to determine hand shake is computationally cheaper than analyzing full resolution image data, and will not confuse blur caused by object motion in the scene with blur caused by hand shake. We use an external gyroscope because the internal accelerometer in the N900 is not sufficiently accurate for this task.

To use the gyroscope with the FCam API, we created a device subclass representing a 3-axis gyroscope. The gyroscope object then tags frames with the IMU measurements recorded during the image exposure. The application streams full-resolution raw frames, saving them to storage only when their gyroscope tags indicate low motion during the frame in question. The ease with which this external device could be incorporated is one of the key strengths of our architecture.

This technique can be extended to longer exposure times where capturing a "lucky image" on its own becomes very unlikely. Indeed, Joshi et al. [2010] show how to deblur the captured images using the motion path recorded by the IMU as a prior.

Figure 10: ***HDR Imaging.*** *A programmable camera running the FCam API improves HDR acquisition in three ways. First, it lets us cycle the image sensor through three exposure times at video rate to meter for HDR and display an HDR viewfinder. Second, it lets us capture the burst of full-resolution images at maximum sensor rate to minimize motion artifacts (the three images on the left). Finally, the programmability of the platform lets us composite and produce the result on the camera for immediate review (on the far right).*

Figure 11: ***Low-Light Imaging.*** *We use the FCam API to create a low-light camera mode. For viewfinding, we implement the method of [Adams et al. 2008] which aligns and averages viewfinder frames. For capture, we implement the method of Tico and Pulli [2009], which fuses the crisp edges of a short-exposure high-gain frame (left), with the superior colors and low noise of a long-exposure low-gain frame (middle). The result is fused directly on the camera for immediate review.*

5.3 Foveal Imaging

CMOS image sensors are typically bandwidth-limited devices that can expose pixels faster than they can be read out into memory. Full-sensor-resolution images can only be read out at a limited frame rate: roughly 12 fps on our platforms. Low-resolution images, produced by downsampling or cropping on the sensor, can be read at a higher-rate: up to 90 fps on the F2. Given that we have a limited pixel budget, it makes sense to only capture those pixels that are useful measurements of the scene. In particular, image regions that are out-of-focus or oversaturated can safely be recorded at low spatial resolution, and image regions that do not change over time can safely be recorded at low temporal resolution.

Foveal imaging uses a streaming burst, containing shots that alternate between downsampling and cropping on the sensor. The downsampled view provides a 640×480 view of the entire scene, and the cropped view provides a 640×480 inset of one portion of the scene, analogously to the human fovea (Figure 9). The fovea can be placed on the center of the scene, moved around at random in order to capture texture samples, or programmed to preferentially sample sharp, moving, or well-exposed regions. For now, we have focused on acquiring the data, and present results produced by moving the fovea along a prescribed path. In the future, we intend to use this data to synthesize full-resolution high-framerate video, similar to the work of Bhat et al. [2007].

Downsampling and cropping on the sensor is a capability of the Aptina sensor in the F2 not exposed by the base API. To access this, we use derived versions of the `Sensor`, `Shot`, and `Frame` classes specific to the F2 API implementation. These extensions live in a sub-namespace of the FCam API. In general, this is how FCam handles platform-specific extensions.

5.4 HDR Viewfinding and Capture

HDR photography operates by taking several photographs and merging them into a single image that better captures the range of intensities of the scene [Reinhard et al. 2006]. While modern cameras include a "bracket mode" for taking a set of photos separated by a pre-set number of stops, they do not include a complete "HDR mode" that provides automatic metering, viewfinding, and compositing of HDR shots. We use the FCam API to implement such an application on the F2 and N900 platforms.

HDR metering and viewfinding is done by *streaming* a burst of three 640×480 shots, whose exposure times are adjusted based on the scene content, in a manner similar to Kang et al. [2003]. The HDR metering algorithm sets the long-exposure frame to capture the shadows, the short exposure to capture the highlights, and the middle exposure as the midpoint of the two. As the burst is streamed by the sensor, the three most recently captured images are merged into an HDR image, globally tone-mapped with a gamma curve, and displayed in the viewfinder in real time. This allows the photographer to view the full dynamic range that will be recorded in the final capture, assisting in composing the photograph.

Once it is composed, a high-quality HDR image is captured by creating a burst of three full-resolution shots, with exposure and gain parameters copied from the viewfinder burst. The shots are *captured* by the sensor, and the resulting frames are aligned and then merged into a final image using the *Exposure Fusion* algorithm [Mertens et al. 2007]. Figure 10 shows the captured images and results produced by our N900 implementation.

5.5 Low-Light Viewfinding and Capture

Taking high-quality photographs in low light is a challenging task. To achieve the desired image brightness, one must either increase gain, which increases noise, or increase exposure time, which introduces motion blur and lowers the frame rate of the viewfinder. In this application, we use the capabilities of the FCam API to implement a low-light camera mode, which augments viewfinding and image capture using the algorithms of Adams et al. [2008] and Tico and Pulli [2009], respectively.

The viewfinder of our low-light camera application *streams* short exposure shots at high gain. It aligns and averages a moving window of the resulting frames to reduce the resulting noise without sacrificing frame rate or introducing blur due to camera motion.

To acquire a full-resolution image, we *capture* a pair of shots: one using a high gain and short exposure, and one using a low gain and long exposure. The former has low motion blur, and the latter has low noise. We fuse the resulting frames using the algorithm of

Figure 12: ***Extended Dynamic Range Panorama Capture.*** *A Frankencamera platform allows for experimentation with novel capture interfaces and camera modes. Here we show a semi-automated panorama capture program. The image on the upper left shows the capture interface, with a map of the captured images and the relative location of the camera's current field of view. Images are taken by alternating between two different exposures, which are then combined in-camera to create an extended dynamic range panorama.*

Tico and Pulli, which combines the best features of each image to produce a crisp, low-noise photograph (Figure 11).

5.6 Panorama Capture

The field of view of a regular camera can be extended by capturing several overlapping images of a scene and stitching them into a single panoramic image. However, the process of capturing individual images is time-consuming and prone to errors, as the photographer needs to ensure that all areas of the scene are covered. This is difficult since panoramas are traditionally stitched off-camera, so no on-line preview of this capture process is available.

In order to address these issues, we implemented an application for capturing and generating panoramas using the FCam API on the N900. In the capture interface, the viewfinder alignment algorithm [Adams et al. 2008] tracks the position of the current viewfinder frame with respect to the previously captured images, and a new high-resolution image is automatically captured when the camera points to an area that contains enough new scene content. A map showing the relative positions of the previously captured images and the current camera pose guides the user in moving the camera (top left of Figure 12). Once the user has covered the desired field of view, the images are stitched into a panorama in-camera, and the result can be viewed for immediate assessment.

In addition to in-camera stitching, we can use the FCam API's ability to individually set the exposure time for each shot to create a panorama with extended dynamic range, in the manner of Wilburn et al. [2005]. In this mode, the exposure time of the captured frames alternates between short and long, and the amount of overlap between successive frames is increased so that each region of the scene is imaged by at least one short-exposure frame, and at least one long-exposure frame. In the stitching phase, the long and short exposure panoramas are generated separately, then combined [Mertens et al. 2007] to create an extended dynamic range result.

6 Conclusion

We have described the Frankencamera – a camera architecture suitable for experimentation in computational photography, and two implementations: our custom-built F2, and a Nokia N900 running the Frankencamera software stack. Our architecture includes an API that encapsulates camera state in the shots and frames that flow through the imaging pipeline, rather than in the photographic devices that make up the camera. By doing so, we unlock the underexploited potential of commonly available imaging hardware. The applications we have explored thus far are low-level photographic ones. With this platform, we now plan to explore applications in augmented reality, camera user interfaces, and augmenting photography using online services and photo galleries.

While implementing our architecture and API, we ran up against several limitations of the underlying hardware. We summarize them here both to express the corresponding limitations of our implementations, and also to provide imaging hardware designers with a wish-list of features for new platforms. Future imaging platforms should support the following:

1. Per-frame resolution switching at video-rate (without pipeline flush). This must be supported by the imaging hardware and the lowest levels of the software stack. In our implementations, resolution switches incur a ∼700ms delay.

2. Imaging processors that support streaming data from multiple image sensors at once. While our architecture supports multiple image sensors, neither of our implementations is capable of this.

3. A fast path from the imaging pipeline into the GPU. Ideally, the imaging pipeline must be able to output image data directly into an OpenGL ES texture target, without extra memory copies or data reshuffling. While image data can be routed to the GPU on our implementations, this introduces a latency of roughly a third of a second, which is enough to prevent us from using the GPU to transform viewfinder data.

4. A feature detector and descriptor generator among the statistics collection modules in the imaging processor. Many interesting imaging tasks require real-time image alignment, or more general feature tracking, which is computationally expensive on a CPU, and causes several of our applications to run more slowly than we would like.

5. More generally, we would like to see programmable execution stages replace the fixed-function transformation and statistics generation modules in the imaging path. Stages should be able to perform global maps (like gamma correction), global reductions (like histogram generation), and also reductions on local image patches (like demosaicking). We believe that many interesting image processing algorithms that are currently too computationally expensive for embedded devices (such as accelerated bilateral filters [Chen et al. 2007]) could be elegantly expressed in such a framework.

The central goal of this project is to enable research in computational photography. We are therefore distributing our platforms to students in computational photography courses, and are eager to see what will emerge. In the longer term, our hope is that consumer cameras and devices will become programmable along the lines of what we have described, enabling exciting new research and creating a vibrant community of programmer-photographers.

Acknowledgements

A. Adams is supported by a Reed-Hodgson Stanford Graduate Fellowship; E.-V. Talvala is supported by a Kodak Fellowship. S.H. Park and J. Baek acknowledge support by Nokia. D. Jacobs receives support from a Hewlett Packard Fellowship, and J. Dolson receives support from an NDSEG Graduate Fellowship from the United States Department of Defense. D. Vaquero was an intern at Nokia during this work. This work was partially done while W. Matusik was a Senior Research Scientist, and B. Ajdin was an intern at Adobe Systems, Inc., and we thank David Salesin and the Advanced Technology Labs for support and feedback. Finally, M. Levoy acknowledges support from the National Science Foundation under award 0540872.

References

ADAMS, A., GELFAND, N., AND PULLI, K. 2008. Viewfinder alignment. *Computer Graphics Forum (Proc. Eurographics) 27*, 2, 597–606.

ADOBE, INC., 2010. The Digital Negative Format. `http://www.adobe.com/products/dng/`.

2010. The Ångström Linux Distribution. `http://www.angstrom-distribution.org/`.

BAE, S., AGARWALA, A., AND DURAND, F. 2010. Computational re-photography. *ACM Transactions on Graphics (To appear)*.

BHAT, P., ZITNICK, C. L., SNAVELY, N., AGARWALA, A., AGRAWALA, M., COHEN, M., CURLESS, B., AND KANG, S. B. 2007. Using photographs to enhance videos of a static scene. In *Proc. Eurographics Symposium on Rendering*.

BRAMBERGER, M., DOBLANDER, A., MAIER, A., RINNER, B., AND SCHWABACH, H. 2006. Distributed embedded smart cameras for surveillance applications. *Computer 39*, 2, 68–75.

2010. The CHDK Project. `http://chdk.wikia.com`.

CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Transactions on Graphics 26*, 3 (July), 103:1–103:9.

DEBEVEC, P. E., AND MALIK, J. 1997. Recovering high dynamic range radiance maps from photographs. In *Proceedings of ACM SIGGRAPH 1997*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 369–378.

DURAND, F., 2009. private communication.

EISEMANN, E., AND DURAND, F. 2004. Flash photography enhancement via intrinsic relighting. *ACM Transactions on Graphics 23*, 3 (Aug.), 673–678.

FILIPPOV, A. 2003. Reconfigurable high resolution network camera. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, 276–277.

GREENBERG, S., AND FITCHETT, C. 2001. Phidgets: easy development of physical interfaces through physical widgets. In *UIST '01: Proc. of the 14th annual ACM symposium on user interface software and technology*, ACM, New York, NY, USA, 209–218.

HENGSTLER, S., PRASHANTH, D., FONG, S., AND AGHAJAN, H. 2007. Mesheye: a hybrid-resolution smart camera mote for applications in distributed intelligent surveillance. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, 360–369.

JOSHI, N., KANG, S. B., ZITNICK, C. L., AND SZELISKI, R. 2010. Image deblurring using inertial measurement sensors. *ACM Transactions on Graphics (Proc. SIGGRAPH) 29*, 3 (Aug.).

KANG, S. B., UYTTENDAELE, M., WINDER, S., AND SZELISKI, R. 2003. High dynamic range video. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, 319–325.

KLEIHORST, R., SCHUELER, B., DANILIN, A., AND HEIJLIGERS, M. 2006. Smart camera mote with high performance vision system. In *ACM SenSys 2006 Workshop on Distributed Smart Cameras (DSC 2006)*.

2010. The Maemo Linux Distribution. `http://maemo.org/`.

2010. The Magic Lantern project. `http://magiclantern.wikia.com/wiki/`.

MANN, S., AND PICARD, R. W. 1995. On being 'undigital' with digital cameras: Extending dynamic range by combining differently exposed pictures. In *Proceedings of IS&T*, 442–448.

MERTENS, T., KAUTZ, J., AND REETH, F. V. 2007. Exposure fusion. *Proceedings of Pacific Graphics*.

PETSCHNIGG, G., SZELISKI, R., AGRAWALA, M., COHEN, M., HOPPE, H., AND TOYAMA, K. 2004. Digital photography with flash and no-flash image pairs. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, 664–672.

RAHIMI, M., BAER, R., IROEZI, O., GARCIA, J. C., WARRIOR, J., ESTRIN, D., AND SRIVASTAVA, M. 2005. Cyclops: in situ image sensing and interpretation in wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, 192–204.

RASKAR, R., AND TUMBLIN, J. 2010. *Computational Photography: Mastering New Techniques for Lenses, Lighting, and Sensors*. In Press. A K Peters, Natick, MA, USA.

RASKAR, R., AGRAWAL, A., AND TUMBLIN, J. 2006. Coded exposure photography: motion deblurring using fluttered shutter. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, ACM, New York, NY, USA, 795–804.

REINHARD, E., WARD, G., PATTANAIK, S., AND DEBEVEC, P. 2006. *High Dynamic Range Imaging - Acquisition, Display and Image-based Lighting*. Morgan Kaufman Publishers, 500 Sansome Street, Suite 400, San Francisco, CA 94111.

ROWE, A., GOODE, A., GOEL, D., AND NOURBAKHSH, I. 2007. CMUcam3: An open programmable embedded vision sensor. Tech. Rep. RI-TR-07-13, Carnegie Mellon Robotics Institute, May.

TICO, M., AND PULLI, K. 2009. Image enhancement method via blur and noisy image fusion. In *ICIP '09: IEEE International Conference on Image Processing*, 1521–1525.

WILBURN, B., JOSHI, N., VAISH, V., TALVALA, E.-V., ANTUNEZ, E., BARTH, A., ADAMS, A., HOROWITZ, M., AND LEVOY, M. 2005. High performance imaging using large camera arrays. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 765–776.

WOLF, W., OZER, B., AND LV, T. 2002. Smart cameras as embedded systems. *Computer 35*, 48–53.

YODAIKEN, V. 1999. The RTLinux Manifesto. In *Proceedings of The 5th Linux Expo*.