THE FRANKENCAMERA:
BUILDING A PROGRAMMABLE CAMERA FOR COMPUTATIONAL PHOTOGRAPHY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Eino-Ville Aleksi Talvala

February 2011

2011 by Eino-Ville Aleksi Talvala. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: http://purl.stanford.edu/yf234dx5834

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Marc Levoy

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Digital cameras, both in traditional form factors and as parts of cell phones, have become ubiquitous over the last decade. But even though many cameras have as much processing power as a desktop computer from a decade ago, the user interfaces and photographic capabilities of digital cameras have changed very little from those of their film predecessors. And while every digital camera today is a complete embedded computer, for the most part they remain black boxes to the end-user — they cannot be reprogrammed or modified.

As a response to the rise of digital photography, the field of computational photography has emerged. Its aim is to improve photography by combining computing power with digital imaging, and create new images that no traditional camera could capture. While many interesting and powerful techniques have been developed by researchers in computational photography, the field has been hampered by the black box nature of digital cameras. Most techniques can only be tested in a controlled studio environments, on still scenes, and require elaborate experimental setups to work around the restrictions imposed by limited camera control.

This dissertation presents the Frankencamera platform, a digital camera system designed for researchers in computational photography. The Frankencamera is a fully open, fully programmable digital camera — one that can be easily modified to test out new methods and algorithms in real-world scenarios. Underlying the platform is the Frankencamera architecture, which provides a new way to view camera hardware. It allows for

precise per-frame control of capture parameters, as well as accurate synchronization of all the assorted hardware devices that need to cooperate for a successful photograph.

Based on this architecture, this dissertation details two hardware platforms: the F2, a custom-built camera that can easily be modified for novel uses; and the Nokia N900, a commercial smartphone that can be software-modified to become a Frankencamera. Both platforms can be easily programmed using the FCam API, which embodies the Frankencamera architecture, making it easy for researchers and developers to precisely control the underlying camera hardware, without needing low-level knowledge of the camera internals.

Finally, this dissertation presents several sample applications to demonstrate the usefulness of the Frankencamera platform. The applications are developed for both the F2 and the N900 using the FCam API, and include projects by researchers in the field, as well as by students in a classroom environment. Several of these applications could not have been developed for any existing camera platform, and the ease and speed at which they were written show that the Frankencamera platform is a compelling tool for computational photography.

Acknowledgements

Over my seven years at Stanford, I worked with and was helped by a great many people, without whom this dissertation would not have seen the light of day.

I was lucky to have two great advisers, with complementary strengths. Prof. Mark Horowitz was an endless fount of knowledge on systems and hardware design, and was always able to identify the key issues in any problem or plan and lay them out plainly. Prof. Marc Levoy shared his great expertise in computer graphics and cameras, as well as his ability to identify promising new directions for research. Both shared a great enthusiasm and sense of excitement for their research and for their students. They could always be counted on to get me to look at the big picture when I was deep in the guts of yet another frustrating problem, and to remember that the problem I was working on was in fact part of a larger, exciting project. Thanks to you both for making it easy to straddle between two research groups.

I'm also grateful to Prof. Pat Hanrahan, who stepped in at the last minute to be my third reader and also provided excellent advice on writing up a systems dissertation. I also want to thank Prof. Dwight Nishimura, who enthusiastically agreed to chair my oral examination.

Kari Pulli's advice and feedback improved the Frankencamera design greatly, and he and his team from Nokia Research Center Palo Alto were valuable collaborators throughout the Frankencamera project. Thanks Natasha, Marius, Daniel, and Timo! Our research partners across the Atlantic — Hendrik Lensch, Wojcieck Matusik, and Boris Ajdin — also

helped shape the Frankencamera to its final form.

Andrew Adams co-designed and built the Frankencamera platform with me, and shared an office for many years with only a few complaints about my sense of humor. Thanks for everything, Andrew! Many thanks also go out to the other students in the Graphics Lab and the VLSI Group: Bennett, Amin, Alex, Vaibhav, Gaurav, Augusto, and Billy gave me great opportunities to work with them when I was just starting out. Zhang, Sung Hee, Dave, Jongmin, Jen, and Abe were both great company and great collaborators on several projects, including the Frankencamera project. Jeremy and Björn also shared the office with me, offering viewpoints from outside my own narrow corner of research. The lab administrators Heather, Ada, Teresa, Melissa, and Monica kept everything from falling apart, and John Gerth both kept the computers up and running and provided many useful insights on practical computing systems as a whole.

At the start of my career at Stanford, I was awarded a National Science Foundation Graduate Research Fellowship, which gave me the ability to explore the research groups and projects at Stanford without having to immediately scramble for funding. Later, I was funded by a Kodak Graduate Fellowship, which allowed me to complete my studies with no worries about money.

My parents, Outi and Harri, somehow managed to keep me in one piece through my early years of scientific exploration (going down the wrong side of the sledding hill with one's eyes closed, in hindsight, is not the best idea), and never stopped prodding me to try new things. My brother Jussi was always happy to listen to me ramble about my research, a rare talent. Kiitos kaikesta!

And finally, thank you Kirstin. You never seemed to doubt for a moment that I'd get this dissertation done, even when I was very unsure of it myself. Now it's your turn![*]

[*] No rush...

Contents

List of Figures

List of Code Examples

Chapter 1

Introduction - *camera obscura*

The word ‘camera’ comes from the Latin *camera obscura*, “dark room.” And coincidentally, much of the modern digital camera is still a black box. Its core cannot be modified, extended or improved without substantial feats of reverse engineering. The user interfaces and capabilities of cameras have changed little since the world transitioned from film to digital cameras, while the computational power hidden inside the black box has increased tremendously. This dissertation presents a new camera system that aims to open the black box for researchers and photographers.

Over the last decade, as digital cameras became commonplace, researchers began to explore how to improve photography through computation. Combining techniques from computer graphics, computer vision, optics, and medical imaging, researchers formed the field of computational photography. Common approaches in computational photography involve modifying the optics of cameras in clever ways, fusing together multiple images to create a better one, and various other alterations to the post-processing of digital images. But without access to the core of the camera, many interesting avenues of research cannot be explored. Thus, the black box has become an impediment to the growth of computational photography.

1.1 Problems with the black box

As an example, one can look at the technique of high dynamic range (HDR) imaging [1]. One of the oldest ideas in computational photography, HDR techniques seek to overcome a common problem in photography: A real-world scene contains a range of luminances that exceed the range the camera sensor can capture. Either the bright sunlit wall saturates to pure white, or the shadows under the bushes become a pure black — no camera setting can record both at once. The most common HDR technique simply involves taking multiple images of a scene, each with a different exposure setting, so that each part of the scene is well-exposed in at least one photograph. With knowledge of sensor parameters and the exposure times used, these multiple digital images can later be combined into a single image [2, 3]. A great deal of research has been done on how to best combine the captured images, and then on how to best display the HDR image on a computer screen with far less dynamic range than the image contains, a process known as tonemapping [1].

However, very little work has been done on the problem of metering for HDR — how to select the number of images to capture and the best camera settings for each picture. Only some recent theoretical work [4] exists, none of which has been tested on real-world devices. While every digital camera includes sophisticated algorithms to automatically meter single photographs, HDR captures still must be done manually. The metering must be done by hand, either with a light meter (hand-held or incorporated in the camera), or by taking several test images to establish the dynamic range of the scene.

This makes the process of recording HDR image stacks take minutes at best, from setting up the tripod to recording the last photograph. And even after the photographer determines the number of captures and their settings, manually capturing the needed images will take several seconds per photograph even for proficient operators. This is a substantial drawback, since noticeable motion in the scene during the capture process

leads to artifacts in the final result. Therefore, the practical uses of HDR imaging have largely been restricted to capturing nearly static scenes such as large-scale landscapes, architecture, and still lifes. And through all this, all the sophistication of the camera's control routines, its complex metering circuitry, and high burst capture rate go unused.

Without access to the internals of the digital camera, without the ability to rewrite the metering and burst modes of the camera, there is no way to improve this process. Researchers cannot easily test an idea or an implementation on a real, portable device, and thus cannot verify if their solution is truly feasible on a real camera in real photographic conditions. The same pattern repeats for many other methods and ideas in computational photography.

If computational photography is to keep progressing, it cannot solely focus on simply post-processing images captured by existing cameras, or on modifying the periphery. To truly enable new ways and capabilities for photography, researchers must be able to implement their ideas on portable platforms, not just in a studio setup that cannot be moved or easily reproduced.

1.2 The causes of the black box

In 2000, digital cameras accounted for only 14% of worldwide camera sales. Since then, they have rapidly supplanted film cameras, as shown in Figure 1.1. By 2009, digital cameras accounted for essentially the entire consumer market with over 100 million digital cameras produced a year [5]. But while cameras have rapidly transitioned from film to digital, in many ways they have changed very little. While digital image capture and storage has substantial benefits in terms of storage capacity and convenience, the conventions of the user interfaces and the kinds of capabilities cameras include are mostly unchanged. The same parameters are shown to the user, and the same types of controls exist to control the camera. Higher-end cameras like digital single-lens-reflex

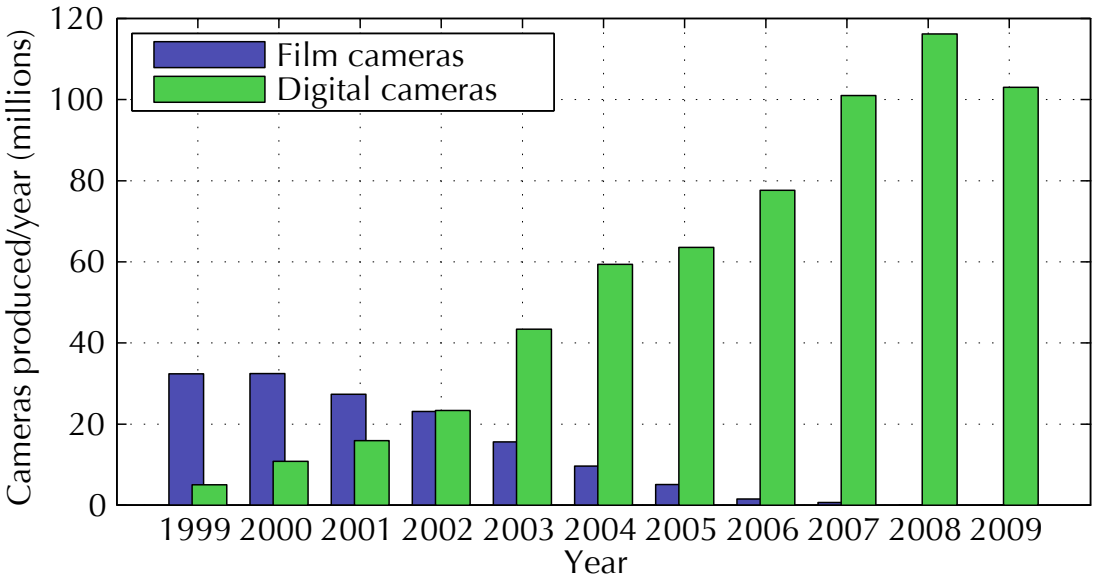


Figure 1.1: Film vs. digital camera production since 1999. Film camera production statistics past 2007 are not available. Data based on CIPA production numbers [5, 6]

(dSLR) cameras allow the user to disable or guide the automation for metering or focusing, but they do not allow for the replacement or rewriting of these algorithms. Low-end cameras like digital point-and-shoot cameras, intended for simple operation, offer very little if any customization.

But all modern cameras contain a great deal of processing power, in order to maximize the rate of image capture and storage. Internally, they are all complete embedded computer systems, running on a variety of operating systems and processors. A great deal of the camera's functionality is expressed in software, not in hardware, and much of what is in hardware is configurable. Yet only the camera manufacturer has access to those levels of the camera — an end-user can only select from the prewritten options.

One reason for such limits may be that traditional camera manufacturers see themselves as hardware companies, making a complete, discrete product — a high-quality camera [7]. While a typical camera contains a great deal of software, it likely only does so

because it is the simplest way to build a camera, not because the camera manufacturers have any interest in software and software infrastructures. This is consistent with the fact that the camera industry tends to be very secretive, unwilling to discuss in any detail how their black boxes work. Therefore, giving anyone outside their company access to the core bits of a camera would go against their beliefs in many ways. It makes their hardware product into a software platform that requires support. It opens the manufacturer to patent claims if the opened camera sections reveal intellectual property registered by others. Finally, some manufacturers have expressed worry that allowing third-party code to run on their cameras could diminish their reputation if that code is of poor quality. As a result, camera makers at best allow for remote control of their cameras, with access roughly equal to that of a human operator using the camera manually.

Of course, the cell phone camera has recently been making substantial inroads on the low-end camera market, and as the quality of cell phone cameras increases, they are likely to completely replace the basic point-and-shoot camera. And cell phones, unlike cameras, have been programmable for several years now, with official support from phone manufacturers. High-end smartphones such as the Nokia N-series, the Apple iPhone line, and Google's Android phones have a great deal of computational power and reasonable camera subsystems. And indeed, each manufacturer includes application programming interfaces (APIs) for camera control, available to all developers. These APIs, however, are typically just sufficient for implementing a basic point-and-shoot camera application, and little more. This is understandable, since the rapid development cycles of cell phone products leave no time for adding features not perceived to be in demand. Instead, the manufacturers build in just enough functionality to write their own camera application.

In part, the work presented here aims to show the value of additional camera control, so that in the future, manufacturers might be persuaded to include such features by default.

1.3 Opening up the box

The Frankencamera architecture presented in this dissertation is designed to make internals of cameras accessible to researchers. It is an open architecture, where no part of the processing pipeline is obscured, and no hidden algorithms control essential parts of the capture process.

Chapter 2 explores the field of computational photography in more detail, to demonstrate the need for a programmable camera to advance the field. Then, the state of the camera industry is discussed, to make the case that none of the existing platforms are sufficient for computational photography.

The design of the Frankencamera architecture is based on the key requirements for a programmable camera, which will be explored in Chapter 3. As an abstract architecture, the Frankencamera is intended to be implementable on many platforms, from cell phones to high-end digital cameras. Unlike typical mainstream camera APIs, the Frankencamera architecture simultaneously provides both high throughput and precision control, by modeling the camera imaging system explicitly as a pipeline through which both requests for images and the images themselves flow. Chapter 4 covers the Frankencamera architecture in detail.

To demonstrate that the architecture is useful and practical, it must be implemented. In Chapter 5, the two current Frankencamera hardware platforms are described: the custom-build F2 Frankencamera, and the Nokia N900 smartphone. And in Chapter 6, the FCam camera control API is laid out, along with the details of the software implementation of the API.

Finally, Chapter 7 concludes the dissertation. It first covers several applications written for the Frankencamera platforms in order to validate the platform design and implementation, and then discusses further extensions to the Frankencamera platform and other future work.

1.4 Contributions

The Frankencamera project is large, and several people have contributed valuable work to it. While for completeness the whole project is described, the specific contributions of this dissertation are:

- A list of the key requirements for a programmable camera to be useful for computational photography research.
- An abstract architecture for programmable cameras, called the Frankencamera architecture.
- The design of a custom digital camera implementing the architecture, the F2 Frankencamera.
- The design of an application-level C++ API, called FCam, for writing computational photography applications on Frankencameras.
- The implementation of FCam for both the F2 and the Nokia N900 smartphone.

The FCam API and the software implementations were designed and implemented in close collaboration with Andrew Adams, who was also the co-architect of the Frankencamera architecture as a whole. The Frankencamera is also discussed in a paper published in the 2010 SIGGRAPH conference [8]. The FCam API implementation and specifications for the Nokia N900 are also publicly available [9], and applications using the FCam API can be installed and run on any N900 smartphone.

Chapter 2

Today's cameras and computational photography

Before embarking on the task of building a whole new programmable platform, it is valuable to examine exactly why building one is necessary. This chapter is divided into two sections, to first explain why researchers in computational photography need a programmable camera platform to push the boundaries of the field further, and then to show that existing platforms are not up to the task.

The first section outlines the development of computational photography over the last several years, to demonstrate that the lack of a programmable camera has resulted in a strong trend of working only on post-processing algorithms. Further, the few researchers that have considered capture-time issues have been handicapped by having no way to test their ideas. In addition, the lack of a demonstration platform on which to run novel applications reduces the visibility and the impact of the field as a whole.

The next section covers the state of the art in camera systems and their programming interfaces, and argues that existing platforms can be split into two general categories. The first set of cameras do not provide enough (or any) control over their imaging pipelines to be usable for most computational photography research. This category includes all

consumer digital cameras and camera phones. The second set of cameras are not self-contained devices that one can take outdoors and use like a typical consumer camera. In this category are the various machine vision, network, and security camera systems. Neither set of cameras provides all of the functionality that a programmable camera needs to provide.

This dissertation assumes basic familiarity with photographic terms and technologies. For further information about them, Goldberg's *Dark Side of the Lens* [10] is a useful reference for camera technology (although it pre-dates digital image sensors).

2.1 A brief survey of computational photography

The young field of computational photography lies at the intersection of several older research areas. It was catalyzed into being by the transition from film to digital cameras, which created a rich source of digital photographs to mine and manipulate. Using techniques from computer graphics, computer vision, medical imaging, and optics, the goal of computational photography is to create novel images from digital photographs that are better than the source images were in some way. The image captured at the sensor is just the starting point, not the final product. Unlike computer vision, the target audience is people, not computers, and unlike medical imaging, the goal is to create a compelling image, not a tool for diagnosis.

Of course, standing at the crossroads puts one in a good position to transfer knowledge in both directions between disparate fields. Some computational photography techniques — for example, high-dimensional filtering and plenoptic imaging — are now being applied back to problems in computer vision [11] and medicine [12], respectively.

The classic thread of literature on high-dynamic-range imaging [1, 2, 3] already mentioned in Chapter 1, is a good example of the methods that simply improve the apparent capabilities of a camera. Other techniques that capture a pair or a small set of images of a

scene and meld them together include flash/no flash imaging [13], noisy/blurry image fusion [14], the whole field of panoramic stitching [15], and many others. These techniques primarily involve post-processing sets of images captured by a standard camera on desktop computers.

The field is certainly not restricted to creating just single photographs - the output can be a representation of an entire scene's light transport [16, 17, 18, 19] or a reconstruction of an entire city block or a street [20, 21], from which images can then be rendered on demand. And the inputs are not exclusively digital photographs; many very difficult image processing problems become much easier if additional information can be integrated from other sensors. For example, images blurred due to handshake can be deblurred with the aid of inertial sensors [22]. Again here, the computation for the final image is currently done off-camera, on desktop systems or large computer clusters.

Another thread of research concentrates on modifying the optics of the camera to record more or different information about the scene than a traditional camera would. The data captured with such modified optics is rarely directly viewable, and typically requires extensive post-processing. Methods in this category include plenoptic cameras [23, 24], coded aperture techniques [25, 26], and many other clever modifications to the basic camera optical path [27, 28, 29].

2.2 The missing research

Yet with all this varied research in the field, there is not much of it visible on the camera itself. Researchers have been working on the periphery of cameras: using it as-is to record data for later processing; modifying the optics leading into it; and bolting on additional hardware components. The powerful processing pipelines and complex computational engines inside every modern camera, and the algorithms that run on them, have been left untouched. Some published research can be found on improving the three standard

areas of auto-exposure, auto-white balance, and autofocus algorithms [30, 31, 32] for straightforward photography, but most work on the heart of cameras is done behind (tightly) closed doors by camera manufacturers. And even the little public work that can be found often requires a great deal of engineering to get anywhere. For example, Lee et al. [33], in wishing to experiment with a touchscreen to drive the selection of the autofocus target region, had to construct an entire external control system with an FPGA board and a pulley-drive actuator, in order to control a manual-focus lens attached to a black-box camera.

It can of course be argued that this is no great loss — let the camera industry focus on their expertise, making high-quality devices for traditional photography — and have the computational photography research community build their work on top of that foundation, and not concern themselves with trying to improve systems that the camera industry is continually working on.

But to truly make an impact, researchers in the field cannot afford to do that, for several reasons. First, even methods that rely purely on post-processing a set of photographs still need to capture those photographs. Unfortunately, the access provided by camera manufacturers barely suffices for real-world use and testing in the best cases, and is much too restrictive for most uses. Because of the limitations in access, most methods are tested only in the lab and for still scenes. The reality of hand-held photography in a dynamic world is left unexplored, with the ultimate result that nobody knows for sure if many of the proposed methods are truly robust and usable for everyday photography.

Second, beyond simply being able to better test already-developed algorithms, there is much unexplored terrain to cover in the fundamental operation of digital cameras. If plenoptic cameras, which can digitally refocus their images after capture, become popular, how should their metering and user interface work? What if a scene is being observed by a multitude of devices, like commonly might happen at a sporting event — can the cameras cooperate together, to perhaps create a single massive panorama of a

scene, or maybe allow small groups of cameras to all fire their flash units together? Or could a camera look at the photos its owner has kept or adjusted in a photo editor, and learn how to meter and focus better on the subjects of interest to the photographer? All of these ideas require access and changes to the low-level feedback loops inside cameras, and also to the user interfaces presented to the photographer.

Third, there is a great deal of value in moving all the processing onto the camera. If the final image can be produced on-camera, then the photographer can immediately determine whether the result is what they were looking for, and try again if needed. Otherwise, photography regresses back to the days of film, where the final result can't be seen without a trip to the (digital) darkroom, perhaps days later.

And finally, it is simply frustrating to watch the camera industry very slowly integrate results from computational photography. It took over a decade from the first large-scale dissemination of the idea of HDR in the research community [3], before the camera industry released the first products that incorporated HDR features of any sort, such as the Pentax K-7 released in May 2009 [34]. With a programmable camera, researchers could test and show off their results now, instead of in a decade or so. This would be especially powerful if consumer cameras become sufficiently programmable — then great numbers of people could install an application embodying a new research result and immediately benefit from it, instead of having to buy a new device years down the line that has the feature finally incorporated in it.

For all these reasons, computational photography needs a programmable platform, a portable computational camera. The next section discusses why none of the many existing camera devices are up to the task.

2.3 A tour of the camera industry

It is perhaps inaccurate to claim that there is a single "camera industry." Imaging devices are used far and wide, from mass consumer platforms to very expensive, very narrow niche markets. However, none of the products out there today are really suitable for a test platform for computational photography.

2.3.1 Traditional digital cameras

In terms of user interface and photographic functionality, nothing comes close to the traditional digital cameras in quality, especially when considering the digital single-lens reflex (dSLR) cameras. They are designed solely for photography, with numerous buttons and dials to allow for quick operation without moving one's eye off the high-quality optical viewfinder. With the conversion to digital, modern dSLRs also feature high-resolution LCD screens for reviewing captured images as well as for displaying a live sensor feed for high-definition video recording and as an alternate viewfinder. Their sensors and imaging pipelines can churn out images at 8-10 fps [35] at full resolution. The high-quality lenses available to dSLRs cover most every photographic niche, and the batteries in a typical dSLR allow for the capture of hundreds of photographs before requiring recharging. All in all, they boast impressive specifications and capabilities, and in fact much computational photography research has been done using dSLRs as the source of camera data [36, 17, 27, 28].

Nevertheless, dSLRs (and their smaller cousins, point-and-shoot cameras) have several limitations that make their use for computational photography cumbersome at best, and impossible for many tasks. First, there are no programmable dSLRs — at best, manufacturers like Canon offer software development kits (SDKs) for remote control of their cameras from a PC connected over USB [37]. Second, these SDKs do not offer any more control than can be had through the buttons and menus of the camera itself. The

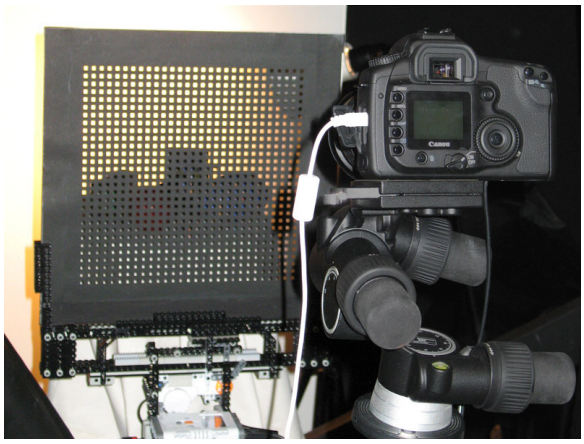


Figure 2.1: An example of an experimental setup for computational photography. The Canon 20D, a high-end digital SLR camera, is coupled with an experimental apparatus [36], here an optical mask mounted on a 2-axis gantry, with the whole system controlled by an off-screen laptop. Capturing 288 photographs for a single data set took between 30 to 60 minutes, a rate of 0.15 to 0.30 frames per second. By comparison, the Canon 20D has a burst capture rate of 5 fps.

SDKs can be used to automate typical photographic tasks, but they cannot be used to rewrite the autofocus routine, or to create a new metering algorithm for HDR imaging. And finally, using these SDKs tethers the camera to a computer, on which the program is actually running. While a laptop can be quite lightweight, being forced to carry a laptop together with the camera can be quite cumbersome, and if a novel user interface is required for an application, it must be displayed on the laptop, not on the camera that the photographer actually wants to use. Examples of the kind of awkward setups this results in can be seen in Figure 2.1 for removing glare from photographs, and in Figure 2.2 for computational re-photography.

The SDKs also focus on traditional studio photography, which can lead to problems when they are used for computational photography. For the digital glare removal project [36], it was necessary to capture dozens of HDR exposure sequences for each dataset. Using the Canon SDK with the Canon EOS 20D dSLR, it turned out that changes

Figure 2.2: Rephotography with a tethered dSLR. In order to assist a photographer with the task of rephotography, where a historical scene is photographed again some time later from the exact same viewpoint, Bae et al. [38] were forced to run all computation and the user interface on a laptop connected to a dSLR, instead of on the camera itself. *Figure courtesy of Soonmin Bae.*

to camera parameters were not synchronized with capture requests — that is, if the exposure setting was changed through an SDK function call, and then immediately a capture request was sent in through another function call, the captured image often ended up using the previous exposure value, not the one just set. Manually adding in one-second delays took care of this problem, at the cost of substantial increases in total capture time. This was acceptable for the still-life scenes used for testing indoors, but failed utterly when attempting to capture an outdoor scene with a very light breeze blowing.

There are a few collaborative projects that are attempting to hack the firmware on digital cameras to enable new functionality, notably the CHDK project [39] for Canon Powershot point-and-shoots, and the Magic Lantern project [40] for the Canon EOS 5D dSLR. However, these projects have mostly been limited to scripting the user interface level of the system, and due to active efforts from Canon, possibly in response to these projects, many of Canon's newer cameras have been locked down and cannot be modified [41]. Such uncertainty makes these projects unattractive as the base for a widespread research platform, which is desirable for easily sharing results with others.

2.3.2 Machine vision cameras

Computer vision and robotics researchers have of course needed digital camera and video systems for a long time, and many industrial uses can be found for digital cameras on the production and quality control lines. Cameras for microscopes and other medical instruments make up another significant market segment.

Dozens of companies produce camera systems for these markets, although the systems share many characteristics. First, the cameras are far more basic than dSLRs are — there is no user interface on the cameras themselves (beyond perhaps a power switch), and often the lenses are completely manual, or only partially controllable (aperture control without focusing, for example). Many cameras have low-resolution sensors compared to consumer cameras, and are often monochrome-only.

Second, they invariably need to be connected to a computer to operate, often over high-bandwidth industry-specific connections such as CameraLink [42]. Their APIs are again typically not tailored toward precise control of the camera settings, normally simply exposing the sensor's (or the controlling FPGA's) register map to user applications, although some exceptions exist. For example, the SDK for the Retiga 4000R CCD camera from QImaging [43] incorporates a input queue of sensor settings to be used for future frames, allowing precise per-frame control similar to the FCam API described in Chapter 6.

Many camera companies in this space are scarcely more open about their designs than the traditional camera manufacturers, so typically it is impossible to reprogram the FPGAs or other programmable units on the cameras themselves without substantial manufacturer assistance. This is true of cameras such as the Point Grey Grasshopper [44] compact high-resolution camera, or of the Basler A400k [45] high-speed camera. The former, for example, has an HDR mode with exactly 4 different exposures to cycle through, but no way to change that number by updating the internal FPGA code. The latter runs

at high speed, but with no access to its internal programming, provides no way to add synchronization between capture settings and output frames.

However, some open platforms exist, notably the Elphel, Inc. line of network cameras [46, 47]. Built into a security camera form factor, the Elphel cameras use wired Ethernet for power and image transfer, running a web server as a user interface. The cameras run Linux, and their FPGA code is publicly available and the cameras themselves are fully open to modification. That said, like the other cameras listed in this section, the Elphel cameras have no physical user interfaces, and cannot run without external power. Additionally, nearly all their processing power is contained within the FPGA, raising the bar for implementing new image processing algorithms significantly, as discussed in Chapter 5.

Academic research projects in this space include the CMUcam [48], intended to be a small, open embedded vision platform. But again, the CMUcam has no user interface and no facilities to work without a connection to a host system.

In spite of these limitations, open machine vision cameras do offer a good starting point for building a full portable camera platform, by adding on user interface and power systems. In fact, the Elphel 10358 camera was a key part of the F1 Frankencamera prototype, and parts of it were used for the successor, the F2 Frankencamera. Both are described in Chapter 5.

2.3.3 Camera phones

Over the last few years, designers of high-end cell phones have been rapidly increasing the quality of their on-board cameras, while simultaneously ramping up the computing power and programmability of their application processors. Smartphones like the Apple iPhone, Google's Android family, and Nokia's N900 have camera systems nearly on par with traditional point-and-shoot cameras, and allow developers to create new applications that run on their platforms. The main processors on these devices are typically ARM

processors running at over 500 MHz, with acceleration for video encoding, 3D graphics, and other numerical processing.

At first glance, such platforms should be nearly ideal for computational photography. While the quality of the images isn't on par with dSLRs, they are good enough for experimenting in most conditions, and the openness and programmability of even the most closed-off smartphone is a breath of fresh air compared to the state of the traditional camera APIs. Additionally, smartphones tend to have high-quality touchscreens, high-speed Internet connectivity, and a broad user base, all valuable for testing and disseminating new ideas in computational photography.

Unfortunately, smartphones are brought down by the state of their camera APIs. Nearly all camera control APIs, including those by Google's Android, Apple's iOS, and Nokia's Symbian, separate the control of the camera hardware into two logical tasks [49, 50, 51]: viewfinding, and high-resolution still capture. The APIs are typically modal — the camera can either be running a viewfinder, with the application receiving a stream of uncompressed low-resolution frames. Or the camera can be set to capture a single high-resolution image, which will be returned to the application through a callback mechanism sometime later, typically in a post-processed form such as a JPEG-compressed buffer. An application cannot request a second high-resolution image until the callback for the first application has been called. For the APIs with more configuration options, the high-resolution capture can be configured with capture settings before the start of capture, and those will be properly applied to capture the requested photograph. For viewfinding, there is no synchronization between when capture settings are changed, and when they apply to the incoming stream of frames. And since the underlying hardware is inevitably a multi-stage pipeline with settings at multiple stages, the latency of changing any given setting is unknown and possibly different from all other settings. Therefore, updating multiple settings simultaneously does not guarantee that all the new settings will take effect on the same future frame.

Each mode is therefore crippled in complementary ways: Viewfinding runs at a high frame rate, which is excellent for capturing bursts of images for computational photography quickly, but it is not possible to know what settings were used to capture each frame, and the low resolution limits prevent the use of viewfinder frames for much more than initial experimentation. Conversely, still image capture is high-quality, and the settings used for capture are clear, and often image metadata is even returned alongside the image buffer. But the capture rate is very slow — since a new capture request can only be issued once the previous one has completed, the frame rate is equal to the reciprocal of the imaging pipeline length, and thus often on the order of seconds per frame. For example, the Nokia N95, one of the first phones with a high-quality camera, takes roughly 4 seconds to process a single capture. Figure 2.3 illustrates these problems.

Orthogonal to the above issues is the fact that most of these APIs have very limited camera control options in the first place — none allow for application-specified exposure times, white balance, or lens operation. Typically, all camera control loops run inside the operating system libraries, inaccessible to the user application, and they cannot be turned off. At most, applications may provide hints to these engines, such as a focus region to target, or an exposure compensation value to apply in metering.

Even for the camera interfaces which are open source, such as the Google Android camera API [49], the built-in access limits are a barrier. Since the APIs do not expose direct controls, hardware designers often take advantage of this and move much of the control to lower levels of the hardware, where it cannot be accessed even if one modifies the open API layers. In terms of access to the full camera subsystem, just having an open source software stack does not guarantee success.

To summarize the state of smartphone APIs for computational photography: They do not provide sufficient access to the core of the camera, and even if they did, the architectures of the existing camera APIs are not suitable for computational photography.

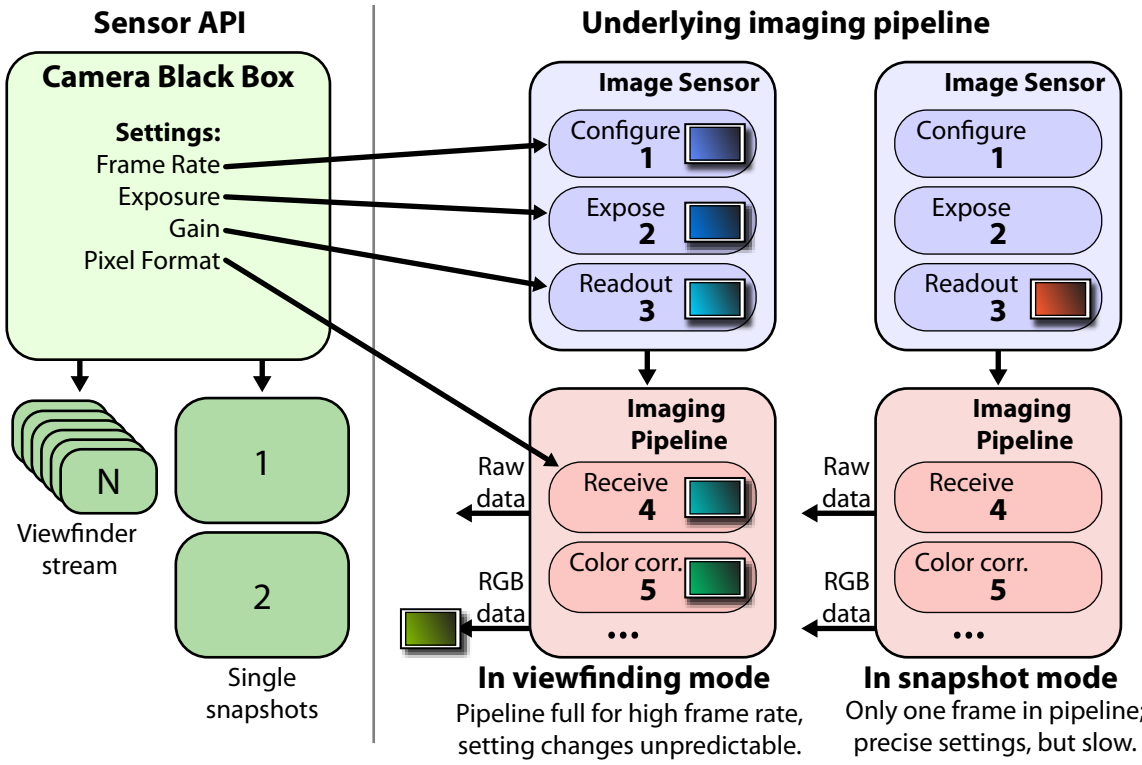


Figure 2.3: Camera API abstraction problems. Most camera APIs model the camera as a black box with settings, and two modes of operation, viewfinding mode and snapshot mode. In reality, the camera is a pipeline of processing stages, with settings for each stage. As a result, viewfinding modes do not allow precise control of the settings for each frame, or even knowledge of the settings actually applied. And in snapshot modes, APIs implement precise control of per-frame settings by allowing only one frame to be passing through the pipeline at a time, substantially cutting down frame rates.

Of course, there is an exception here as well. Nokia has been producing a line of internet tablets, small devices with large touchscreens and WiFi connectivity, for several years. The latest device in the line is the Nokia N900 [52], the first to be a full-fledged cell phone as well. These devices are notable for running Linux internally, with no virtual machine or sandbox sitting in between user applications and the phone hardware. While the native camera API for Linux [53] is roughly equal to the viewfinder mode in the APIs discussed above, due to well-documented hardware and open kernel drivers the existing APIs can be improved upon and circumvented as needed. As will be shown later, the

N900 can indeed be turned into a useful Frankencamera platform.

Clearly, building a programmable camera needs to encompass not just the hardware design and construction (Chapter 5), but also an API aimed for computational photography needs. The API described in Chapter 6 in effect merges the two modes of camera APIs, providing high frame rates, predictable control of sensor parameters on a per-frame basis, and full access to the camera imaging pipeline configuration.

But first, it is useful to clearly define the full set of requirements for a computational camera, and then from that build an overall mental model — an abstract architecture — for a Frankencamera, to guide the design of the concrete Frankencamera implementations. These are the the topics for the next two chapters.

Chapter 3

The requirements for a computational camera

As was made clear in the previous chapter, existing camera platforms come up short in various ways, if one wants to use them for computational photography. To create a better camera platform, it is important to first clearly define the requirements it needs to meet in order to be a truly useful research system. Such a list should also try to be minimal, containing only the fundamental requirements. All the rest of the desirable functionality can then be built from this minimal set.

The requirements summarized below in Section 3.4 were developed over time through the use of existing systems, the construction of successive camera system prototypes, and writing and testing computational photography applications for them. Recounting those experiences in full would be tedious and not particularly instructive. Instead, the requirements will be explored through two case studies of target camera applications, selected to cover a large range of behaviors and needs of computational photography applications.

The first case study is for a basic digital camera application. This is equivalent to the typical camera front-ends that ship with cell phone cameras, or the interfaces to

traditional point-and-shoot digital cameras. Being able to support such an application ensures that basic camera functionality will be implementable by the Frankencamera architecture. The second case study is an HDR panorama application, a more complex computational photography application. The application allows a user to simply pan their camera slowly across a scene, and the camera automatically captures images needed, and merges them into a single high-dynamic-range high-resolution panorama on the camera. For both case studies, it is important to first establish a baseline for the camera hardware on which the applications run.

3.1 Base hardware assumptions

Exactly how a camera application goes about fulfilling its goals depends substantially on the hardware available to it. In order to maximize the number of devices that could conceivably be used as a computational camera, it's important to keep the base hardware assumptions for the case studies as minimal as possible. To that end, the only pieces of expected hardware are:

1. A display of some sort for an electronic viewfinder.
2. An image sensor.
3. A lens.
4. A flash unit (included as a generic stand-in for any camera hardware needing synchronization with the sensor).
5. Some sort of user interface, either physical (buttons) or virtual (touchscreen).
6. A processor to run applications on, and storage for the captured images and code.

And the final addition, as the first key requirement for computational cameras: **A computational camera should be portable and battery-powered**, so that it can actually be

used in the real world outside the laboratory. Without this ability, it is impossible to properly test new ideas, or even to implement many of the existing ideas properly.

Nearly any camera phone, point-and-shoot camera, or a recent dSLR meets these basic hardware assumptions. Imaging devices such as basic USB cameras, machine vision cameras, security cameras, tethered remote-controlled cameras, network cameras, or camera arrays and gantries, do not qualify. That is not to say that the architecture and API developed in this dissertation could not be deployed profitably on such systems, but they are not the focus of the effort. Instead, the goal is to create an open programmable platform as usable as any consumer camera.

3.2 Case study: A point-and-shoot camera application

Over the decades, camera systems have become more and more sophisticated, and even a low-end point-and-shoot camera application has to perform a great number of tasks quickly and accurately to be considered a reasonable camera.

Using the base camera hardware listed above, a basic camera application needs to be able to:

1. Show a smooth video stream on the electronic viewfinder for aiming the camera.
2. Control the sensor parameters for the viewfinder to keep the video properly exposed.
3. Be able to automatically focus the lens on demand, quickly.
4. Switch from displaying the viewfinder to capturing a high-resolution image with minimum delay when the user presses the shutter button.
5. Accurately synchronize firing the flash unit with the start of high-resolution image capture.

Each task is discussed in detail below.

3.2.1 Running a viewfinder

The first task requires an image sensor capable of streaming captured images at video rate, preferably around 30 frames per second. It isn't necessary to stream at full sensor resolution — typical viewfinders have resolutions on the order of 640x480 pixels, so many image sensors support various averaging and subsampling modes that reduce the image output resolution from the sensor. Additionally, the image processing inside the camera must be able to receive these raw sensor frames and convert them for display on a standard LCD panel. Since most color image sensors use Bayer color mosaics [54], the processing pipeline usually consists of demosaicking, noise suppression, color conversion, and possibly resampling to match the viewfinder resolution. This pipeline must be able to operate at 30 fps as well, and should not introduce noticeable lag. "Noticeable" is a subjective measure, but recent experiments in remote surgery [55] suggest that latencies of less than 300 ms are acceptable to surgeons. So conservatively, with non-expert users, if the processing pipeline takes less than 100 ms (3 frames at 30 fps) from image capture to display, the user is very unlikely to be troubled by the time delay between reality and what is displayed on the viewfinder. This all can be boiled down to a simple overall requirement: **A computational camera must be able to display a smooth low-resolution video stream on a viewfinder.**

3.2.2 Metering the viewfinder and autofocus

Task 2 requires the camera to adjust the sensor and lens parameters to keep the video from over- or under-exposing, while running the viewfinder. To do this, the camera needs to run a control loop that monitors the quality of the incoming video frames, and adjusts the sensor exposure, gain, and the lens aperture, to keep the video well-exposed. While

many possible control algorithms could be used to calculate the needed adjustments, they all share several key requirements.

When a new viewfinder frame is received, the control algorithm must also know the sensor and lens settings used to capture that frame. Without those values, all the control algorithm can determine is whether to increase or reduce the overall exposure. Actually calculating new exposure settings requires the old settings as a baseline.

The base hardware includes no secondary sensors to aid with autofocus, such as the phase-based detectors on dSLR cameras [10]. Therefore, Task 3 must be done using contrast-based autofocus, where the main sensor image stream is measured for sharpness while the camera lens focus distance is changed. By analyzing image sharpness over a range of focus distances, the distance at which the scene is the sharpest can be found, and the lens moved to that focus distance setting.

The algorithmic details can vary greatly, but like Task 2, any algorithm will require accurately knowing the lens focus distance at the time of capture for each image.

Beyond these two examples, the need to couple image data with its metadata can easily be generalized to apply to all camera components, especially in a research system where it isn't known a priori which system settings may become pivotal for a novel experiment. As such, the fundamental requirement can be summarized as: **A computational camera must know all the system settings used for each captured frame.**

3.2.3 High-resolution image capture

Once the user has pressed the shutter button, the camera must capture a full-resolution image as quickly as possible (Task 4). The delay between the button press and the actual start of image exposure is known as shutter lag, and should be minimized as much as possible. Lag of more than a few tenths of a second means that fast-moving scenes will have moved substantially before capture begins, possibly ruining the photograph entirely. In general, any sort of a mode switch in the camera should happen as fast as possible

(Here, the sensor must switch from recording the low-resolution viewfinder stream to recording a single high-resolution image, and back again). This point will be explored in more detail in the second case study.

Typically, a standard camera application will record just one high-resolution frame when the shutter button is pressed. This means the camera must be able to set the capture parameters for this one frame accurately. While this may seem like a trivial expectation at first, when combined with capturing bursts of high-resolution images as described in Chapter 2, this requirement breaks most existing camera APIs completely. Again, this point will be discussed further in the next case study.

Of course, the high-resolution images captured by the camera must be high-resolution enough to be useful. Useful is a vague term, but it is clear that a 1-megapixel camera would not be a credible tool for high-quality photography today, when even cell phone cameras are routinely between 3 and 5 megapixels in size. Large display devices (HDTVs, computer monitors, projectors) currently provide 1.5-2 megapixels of resolution for viewing photographs, and are growing in resolution rather slowly. As a conservative benchmark, one can estimate the needed resolution for a standard 8x10 inch print. When viewed at a 12-inch distance, in order to match standard human visual acuity (~ 0.5 line pairs resolvable/arcminute [56]) such a print needs roughly 6.5 megapixels of resolution. If viewed from 18 inches away, 3 megapixels becomes sufficient. Without setting a hard lower limit that's bound to become obsolete, **a computational camera sensor should have enough resolution to take credible photographs.** For most techniques, 4-6 megapixels is currently sufficient.

3.2.4 Firing the flash

A typical camera flash unit sends out an intense pulse of light of very short duration, typically on the order of a millisecond. Therefore, it is clearly essential that the firing of the flash (Task 5) is properly synchronized with the capture of the high-resolution

image. In dark conditions, or equivalently for short exposure times, it doesn't matter when during the exposure the flash fires, since the light collected during the rest of the exposure will be insignificant compared to the flash's contribution as long as the flash does fire sometime during the exposure period.

However, more sophisticated cameras offer both first-curtain and second-curtain sync modes. First-curtain sync flash means that the flash will fire at the very beginning of exposure, and second-curtain sync means that the flash will fire at the very end of the exposure. (The "curtain" refers to the way mechanical focal plane shutters operate.) In these situations, the synchronization of the flash and the sensor must be accurate to the millisecond. Otherwise, the flash may fire outside the sensor's exposure period, and a black image could result.

In cameras with mechanical shutters, the shutter motion must similarly be accurately synchronized with sensor exposure. Exposure times as short as 1/8000 of a second are available in cameras, usually achieved in conjunction with a mechanical shutter. Note that flash synchronization rarely needs to operate at exposure times faster than 1/250 s since there's no reason to use exposures shorter than the flash pulse itself, and that while a mechanical shutter can be set to open for only a fraction of a millisecond, its synchronization with sensor exposure does not have to be as precise, since the sensor can simply start exposing some time before shutter activation, and continue for a short while afterward with no change in image quality.

Generalizing from these examples, a researcher may want to synchronize any arbitrary part of the camera's hardware with the sensor exposure with high timing precision. **A computational camera must allow synchronization of assorted camera hardware with image sensor capture with at least millisecond accuracy.**

Any system that meets these requirements can implement a basic camera application, and of course, all existing consumer cameras fulfill them. However, as discussed in Chapter 2, not all of them expose a sufficient API to reimplement a camera application, if

they expose an API at all.

The second case study takes a look at a more novel application, one that could not be implemented easily on any existing camera platform.

3.3 Case study: An HDR panorama capture application

The field of view of any camera is limited, and sometimes a photographer wants to capture an image of a scene that's much too large to fit into a single capture. Instead, a photographer can capture a panorama by recording multiple photographs, rotating the camera around its optical center between captures. These photographs can then downloaded to a general purpose computer, which detects overlaps between the images, and merges them all into a single seamless whole [15].

The capture process for panoramas is error-prone if unassisted. First, it is hard to confirm that every part of a desired scene has been recorded in at least one photograph. Second, even if the whole scene is recorded, it's hard to tell if the resulting image will be easy to crop to a rectangle for final storage. Third, it takes quite a while to capture the photographs, especially if using a tripod to guarantee that the camera viewpoint won't drift between exposures. Finally, it can also be difficult to ensure that sufficient overlap exists between each capture image to ensure that the stitching process will succeed.

Some cameras offer various limited panorama-assist modes. A typical method helps with capturing simple 1-D panoramas (rotation along one axis only) by superimposing a translucent image of the right edge of the previous captured image on the left edge of the viewfinder, making it easy to line up the next capture properly [57]. Very recently, some cameras have started offering a more automated panorama capture mode, where the camera itself analyzes the scene and determines when to take a new picture. The photographer simply needs to slowly pan the camera across the scene — the camera does the rest, including on-camera merging of the final panorama [58].

This case study is for an automated panorama capture application like the one described above, with the addition of high-dynamic-range capture. Instead of taking a single photograph when new scene content is seen, the HDR panorama capture application takes several images with varying exposures if needed, to span the entire dynamic range of the current section of the scene being imaged.

The same basic hardware as for the previous case study is assumed - nothing beyond what can be found in a typical consumer camera or camera phone. Additional hardware features may improve the application or change how it works, but for defining baseline platform requirements, using the lowest-common-denominator platform is best.

The HDR panorama application should work as follows:

1. The photographer aims the camera at some part of the scene they wish to capture, and presses the shutter button.

2. The HDR application captures a burst of images with varying exposure, and indicates on the viewfinder that the photographer should start moving the camera.

3. The photographer pans the camera over the scene.

4. The application tracks the camera motion, and when it determines that the scene contents are mostly new, it captures another HDR stack of images.

5. The application, using its tracking information, draws a map on the corner of the viewfinder, showing the shape of the overall panorama captured so far.

6. The photographer keeps moving the camera, with the application taking more pictures as needed, until the photographer is satisfied with the shape of the panorama, as shown by the map.

7. The photographer presses the shutter button again, and the HDR panorama application begins the high-quality merging process to create a final composite panorama image, and displays the result for review once done.

The preceding user interaction addresses many of the typical frustrations with capturing a panorama: When to capture a new image, what the shape of the final panorama might be, and ensuring that the overall exposure of the result is acceptable. The case study application here is also one of the example applications implemented on the Frankencamera platform, described in more detail in Section 7.1.2.6.

To do all of the above, what does the application need the platform to provide?

3.3.1 Access to the user interface

The HDR panorama application interacts with the photographer through two channels: The shutter button (Tasks 1 and 7) and the viewfinder (Tasks 2, 4, and 5). Unlike the basic camera application, which simply needs to transfer a stream of video from the sensor to the viewfinder, the HDR panorama program needs to communicate a few more things. It needs to draw the map of the captured panorama on top of the viewfinder. The application must also be able to access physical interfaces like the shutter button, or whatever other devices are attached to the camera or camera phone, like a touchscreen or a remote cable trigger.

In general, a novel camera application often requires a well-though-out custom user interface, typically taking over the whole of the platform's interface to the photographer while it runs. **A computational camera should have a fully open and flexible user interface, both physically though buttons and dials and virtually via a viewfinder.**

3.3.2 Burst capture

Enabling Task 2 turns out be instrumental in a large range of computational photography applications, many of which require capturing multiple images with varying settings as fast as possible [1, 13, 14, 19, 59]. Here, assuming a metering routine has already determined the number of images needed for an HDR stack, and the exposure and gain settings for each, all the camera needs to do is to capture them.

Note that it is essential that the capture process takes as little time as possible, since any scene motion between exposures will make reconstructing the high dynamic range image difficult. For an HDR application, furthermore, the data captured should be raw sensor images, with no demosaicking applied. To minimize the time it takes to capture the burst, **a computational camera should be able to provide raw sensor data at full sensor frame rate.**

However, simply recording a high-speed burst of raw data is insufficient. Each recorded frame must also have a different exposure and gain to form a proper HDR stack, and the application should not have to slow down its acquisition to guarantee this. **A computational camera should allow control of sensor and imaging pipeline settings on a per-frame basis, at full frame rate.**

3.3.3 Tracking and merging

The panorama application must track the camera's pose relative to the scene (Task 4), to know when it has to capture a new HDR stack. This is a real-time image processing task, and requires that the camera has sufficient computational resources to run such a tracking algorithm.

Similarly, once all the needed images have been captured, the panorama application needs to stitch together the images into a final, high-resolution high-dynamic-range panorama. While this task does not have to happen instantaneously, for a reasonable

user experience the processing should be fast enough to allow the user to review the result and recapture the panorama if necessary.

In general, **a computational camera needs to have sufficient computational resources so that typical algorithms from computational photography can run in a reasonable amount of time, and typical control feedback loops can run at sensor rate.**

Of course, this is a moving goal — there will always be algorithms that need more processing power than what's available on a power-limited mobile device, but at the same time, a platform that can't credibly handle any image processing will be a failure.

3.4 Summary of requirements

Distilling the demands of the two case study applications discussed above, the following nine requirements summarize the needed features of a computational camera:

1. Access to raw sensor data at maximum sensor rate.
   Enables minimum-time capture of multiple unprocessed frames, key for any algorithm that needs multiple images to be captured quickly.
2. Control of all sensor and imaging pipeline settings on a per-frame basis.
   Needed for rapid capture of multiple frames in many applications, such as HDR capture, that need different camera settings for each frame.
3. Control of all camera hardware synchronized with sensor exposures.
   Needed for basic camera functions like flash firing, and also key for research systems that add novel hardware to a camera.
4. Accurate knowledge of sensor and hardware settings used for the capture of any given frame.
   Essential for real-time feedback algorithms, and simplifies post-processing routines as well.

5. High enough frame rate for an electronic viewfinder to be implemented.
   Needed for basic camera operation, as many mobile platforms won't have space for an optical viewfinder.
6. High enough sensor resolution to take still photographs of acceptable quality.
   Needed for basic photography, and for credibly testing research ideas.
7. Sufficient computation power to implement on-camera control loops and post-processing algorithms.
   Too little processing power limits the range of applications that can be run on the camera itself.
8. Sufficient UI flexibility to implement novel physical and software interfaces for new applications.
   New applications will require new interfaces, and experimentation for them should not be unduly constrained.
9. Portable and battery-powered.
   Essential for verifying that new algorithms work in real-world conditions.

It is important to note that the requirements are meant to hold at the same time — for example, accurate knowledge of capture settings for each output frame must be possible simultaneously with per-frame control of these settings. And both must work simultaneously with receiving raw sensor data, or with the application spending substantial computational power on its own post-processing.

Written out as a list, these requirements make it easy to determine if a platform is sufficient for researchers, but for creating a new platform, they must be first combined into a unified model of a camera system. The next chapter, therefore, defines such a model: the abstract Frankencamera architecture.

Chapter 4

The Frankencamera architecture

Given the requirements for a computational camera defined in Chapter 3, the next step in designing the Frankencamera platform is to create an abstract model of the platform which fulfills these requirements. This abstract model can then guide the design of the hardware platforms and the software library for programming them.

Figure 4.1 shows the Frankencamera architecture. It consists of the application processor, a general-purpose unit which runs the user application; the image sensor itself; a fixed-function image processing unit which reads images from the sensor; and assorted hardware devices such as the lens and the flash, which are needed for a complete camera.

On a high level, the architecture largely matches the current reality in camera systems: a general-purpose processor controlling a sensor and a fixed-function imaging pipeline along with several other associated devices, all synchronized together. The primary innovation the Frankencamera architecture introduces is treating the imaging pipeline as something that transforms a queue of image requests into images on a 1:1 basis, instead of treating the pipeline as either a free-running frame source, or as a long-latency single-capture image source, as most current camera APIs do. In addition, the architecture explicitly models synchronization between the image sensor and the other hardware

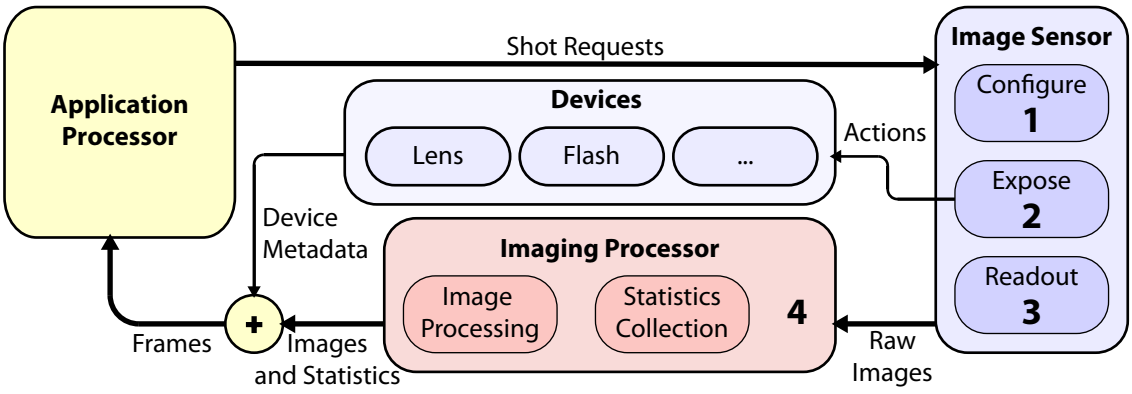


Figure 4.1: The Frankencamera abstract architecture. This block diagram shows the general architecture of a Frankencamera. An imaging pipeline, instead of being a black box with settings, transforms image requests to images. Device actions may be synchronized to sensor operation, and devices and statistics generators may attach metadata to the returned image data. The application has full control of the system, with no hidden routines for metering or autofocus. Multiple requests can be in flight at once, each with unique capture settings. The numbers indicate stages in the imaging pipeline, each of which can contain a single request or image in flight.

devices in the system, such as the lens or the flash, so that the entire capture process can easily be coordinated. The next sections take a look at each of the architectural building blocks in closer detail.

4.1 The application processor

On a programmable platform, the user code needs a place to run. In the Frankencamera architecture, the code is abstracted as running inside a single application processor. In reality, an application may be running on multiple cores simultaneously, or partly on specialized devices like a digital signal processor (DSP) or a graphics processing unit (GPU). But for the purposes of camera control, all that matters is that the user code is the source of requests sent to the imaging pipeline, no matter how they are generated. Multiple requests can be active at a time; equivalently, adding a new request into the pipeline is a non-blocking operation.

The application processor also receives the completed frames from the imaging pipeline. As described further below, these frames contain both image data and camera system configuration information for the corresponding exposure period.

While no researcher would turn down more processing power, mobile devices typically have strong limits on available processing power. Nevertheless, it is important that the application processor has enough computational horsepower to allow useful control algorithms and post-processing routines to run in a timely fashion (Requirement 7 from Section 3.4). Exact performance requirements would be difficult to specify since they'd be out of date quickly, but for the sake of comparison, the current Frankencamera implementation for the Nokia N900 can process a 5-megapixel raw sensor image into a color-corrected RGB image in roughly 700 milliseconds, and can composite an HDR burst together in roughly a minute. As more and more computational power becomes available on mobile platforms, these times will fall, and more and more of the algorithms developed in computational photography will run at usable rates.

4.2 The image sensor and requests

Unlike typical mental models of image sensors (see Chapter 2), the image sensor in the Frankecamera architecture has no persistent state of its own — the application processor does not directly configure any aspect of the sensor. Instead, all sensor configuration is contained within the requests flowing through the pipeline.

The requests define the output resolution, exposure times, frame rate, and so on. They also include the list of hardware device actions that need to occur during the capture of the image data for the request, and all the details of the post-processing to be done to the recorded image. So for example, a request in English would be something like:

Please capture an image with maximum resolution, an exposure time of 10 ms and total capture duration of 33 ms, at ISO 400. Fire the flash at full brightness

starting at 9 ms into the exposure for 1 ms. Calculate a 64-bucket histogram from the raw sensor data, and convert the raw image to the YUV colorspace.

One output frame is produced by the pipeline for each request placed into it. If no requests are in flight, no output will be produced. Compared to a traditional pipeline, in which the sensor simply outputs frames continually, it may appear onerous to require the application to send out a request for every frame that is needed.

However, the usual model of a sensor with built-in settings makes it very difficult to control camera settings on a per-frame basis: Not only is it not clear when the new set of configuration values will take effect, but often some of the new settings will actually end up being applied to a different frame than the others. This is because a typical rolling shutter image sensor is indeed a pipeline — when the sensor is accepting configuration updates for frame N, it is in the middle of exposing frame N-1, and simultaneously reading out frame N-2. Therefore, settings written together which apply to different phases of the sensor's operation may end up affecting different frames. Unless the application carefully models the timing behavior of a platform's imaging pipeline, it cannot achieve frame-level control in a typical camera system. On a Frankencamera system, this problem becomes trivial, fulfilling the programmable camera requirement for precise per-frame control (Requirement 2).

In addition, it is easy to provide a higher-level streaming facility that keeps the imaging pipeline full without constant programmer intervention, without losing per-frame control. One can simply define a holding slot for a request, and when the request queue to the sensor becomes empty, simply copy the streaming request into the queue. In this way, the application is guaranteed to receive frames at full pipeline rate for the given request, without having to enqueue them itself. But the streaming request may be modified at any time, and the application can simply push other requests into the pipeline at any time. Since the streaming request is not used unless the pipeline is empty, the requests directly submitted by the application have priority.

The output from the image sensor is a frame. The frame contains substantially more than just the raw image data itself. First of all, it contains the actual settings used to capture the image data, mirroring the request, along with timestamps for when the exposure took place. Due to hardware capability limits, it is possible that the request could not be fulfilled exactly, so the frame also contains the original request it was based on. As discussed in Chapter 3, including all the request metadata with the image is essential for both camera control routines, and for much of post-capture computational photography.

While the architecture diagram in Figure 4.1 only shows one sensor, multiple sensors could easily be used in a Frankencamera system, each with its own imaging pipeline. Or, if one sensor can easily be triggered externally, it can be encapsulated as a secondary sensor device and be triggered by the primary sensor. The main implementation challenges for a multiple-sensor system are in synchronizing the sensors' captures together as desired for an application, and the Frankencamera architecture places no restrictions on how that should be accomplished.

The Frankencamera architecture places a few requirements on the properties of the image sensor itself. First, the maximum resolution of the sensor needs to be high enough to be a credible camera (Requirement 6) for photography. As discussed in Section 3.2.3, this is a somewhat vague requirement, but arguably anything in the 4-6 megapixel range is sufficient. Second, the sensor must support a frame rate high enough for a smooth electronic viewfinder. This can be at a lower resolution as well, since typical electronic viewfinders have less than a megapixel of resolution. This allows the architecture to meet the programmable camera requirement for a smooth electronic viewfinder (Requirement 5).

These requirements cut out two sections of the sensor market. First, they remove small webcam-type sensors and some machine vision sensors from consideration due to insufficient output resolution. Second, the also rule out many very high-resolution CCD

image sensors. While such sensors have excellent image quality and a long history, they have less flexibility than CMOS sensors in general, and frequently cannot output images faster than a few frames per second due to their pixel counts, insufficient for an electronic viewfinder. Lower-resolution CCDs, such as those found in high-end point-and-shoot cameras, can be read out fast enough for a viewfinder, however.

In general, any sensor used in a recent consumer digital camera would meet these requirements, since nearly all of them provide both a live viewfinder stream and high-resolution photographs.

4.3 The imaging processor

For reasons of power efficiency and speed, most cameras contain a hardware fixed-function image processing unit, sometimes called the ISP, for Imaging Signal Processor. It frequently has a direct, fixed connection to the sensor, so all image data must pass through it before reaching the application. Since this has a clear impact on the latency of the imaging pipeline, and because it makes sense to take advantage of hardware acceleration if its available, the Frankencamera architecture assumes the presence of an ISP in the imaging pipeline. For systems that have no such hardware unit, the functions of the ISP can easily be emulated in software, albeit with a performance and power cost.

ISPs typically have two types of features. The first set involves transforming the image in various ways: demosaicking raw sensor data, doing color space conversions, adjusting white balance, or resizing the image, for example. The architecture requires that the ISP can provide at least two forms of output data: untouched raw sensor data, and a demosaicked format that is suited for display on a platform's viewfinder, at a frame rate high enough for smooth display. The ISP should not be a bottleneck for transporting raw data to the application, to meet Requirement 1. Raw sensor data is required by many computational photography applications, and being able to run a viewfinder

with minimal CPU overhead frees up computational resources for other tasks. The ISP processing should not add so much latency to frame output that the viewfinder display is noticeably lagging reality.

The second set of features involve the collection of image statistics, to aid camera control algorithms. These include units such as a histogram generator, or a sharpness map generator for autofocus. The Frankencamera architecture expects the ISP to at least generate histograms and low-resolution sharpness maps, to reduce the computational load for running basic camera control routines.

Since the capabilities of real ISPs vary greatly both in terms of features and in their reconfigurability (See Section 5.1.1 for the features of the OMAP3 ISP, as an example), the architecture only requires these two types of image outputs and two types of statistics collection units. Any additional features can be included as platform-specific options.

The capabilities of ISPs in mobile devices are clearly increasing rapidly, as a comparison between the OMAP3 ISP and its successor, the OMAP4 ISP, makes clear [60, 61]. In the future, large parts of the Frankencamera architecture can likely be run inside the ISPs themselves, freeing up the main processor units for other tasks. In addition, programmable stages in future ISP image processing pipelines could allow similar advances in camera applications as programmable shaders in GPUs brought to graphics and high-performance computing systems. As an inspiring example, moving computation from the CPU to the GPU allowed the Folding@Home project to accelerate their protein-folding code 60-700 times over contemporary CPU performance [62].

4.4 The other devices

Any usable camera needs more than just the image sensor and processing power. Lenses, flashes, gyroscopes, and other devices are integral parts of most cameras, and a complete camera control API needs to provide easy access to all these devices. Since research in

computational photography often involves novel hardware, it's important to allow as much flexibility in the device interfaces as possible.

Synchronizing devices with sensor capture is done through actions, which represent device behaviors that can be triggered during image exposure. A flash device may have an action for firing itself, and a lens may have an action for changing its focus distance during exposure. These actions also specify their firing time relative to the start of exposure, with millisecond accuracy. Since triggering many hardware devices may include some delay (for example, sending a command over a serial port may take several milliseconds), each action also needs to know its activation latency, so that they can be triggered sufficiently in advance. This latency can simply be pre-measured for each action, as in the FCam implementation described in Chapter 6, or more sophisticated prediction schemes could be used if needed.

Actions are attached to requests, and are activated by the imaging pipeline, which takes into account the declared latency and the desired firing time. As long as the activation latency is constant, device actions will be accurate to within a millisecond, meeting Requirement 3 for device synchronization.

Devices may also tag frames with device-specific metadata, to specify the device state during the frame's exposure. For example, a lens reports its focus distance, focal length, aperture, and their derivatives, while a flash reports whether it fired and when. The completed frames therefore have both assorted device metadata and the image pipeline metadata, which constitutes a complete snapshot of the camera system state at time of exposure, fulfilling Requirement 4.

The architecture described here is not a radical departure from the typical camera API model described in Chapter 2, and can be implemented on many existing camera systems. However, the few key changes – feeding the imaging pipeline with explicit requests instead of allowing a free-running sensor, and explicitly modeling the presence of other hardware that needs synchronization – allow the Frankecamera architecture to

cleanly support the requirements described in Chapter 3. With the architecture in hand, the next chapters will detail the construction of two Frankencamera hardware platforms and the software system that runs on them.

Chapter 5

The hardware

The Frankencamera architecture described in the previous chapter was specifically designed to allow implementation on typical camera systems that exist today. Therefore, one might expect that it should be possible to use most any camera or cell phone platform as a base for a Frankencamera implementation. Unfortunately when the Frankencamera project began, no manufacturer provided enough low-level access to their cameras or camera-bearing cellphones to implement the Frankencamera architecture properly. Therefore, a suitable hardware platform had to be built from scratch.

The first attempt at an open camera platform was posthumously named the F1 Frankencamera. It consisted of an Elphel [46] 10358 network camera connected over Ethernet to a Nokia N800 internet tablet. A Birger Engineering EF-232 Canon lens controller was attached to the front of the Elphel camera, to allow for the use of any Canon EF or EF-S lens. The system was powered by a custom lithium-polymer power supply, and the frame holding the components was made using 80/20 Inc. aluminum struts. The Elphel network cameras run Linux, with a small web server running a camera interface for changing sensor settings and retrieving image data. The N800 acted as the viewfinder and the user interface, communicating with the Elphel web front end. Figure 5.1 shows the completed

Figure 5.1: The F1 Frankencamera. The first experimental hardware platform, the F1 used an Elphel 10353 network camera connected to a Nokia N800 tablet. A Birger EF-232 lens controller allowed Canon EF and EF-S lenses to be used, and a custom power supply allowed for portable use.

prototype.

This first prototype had the benefit of being quick to build, but otherwise was rather limited. Streaming viewfinder images over Ethernet added a half-second lag to the system, making the user experience unpleasant. More importantly, nearly all of the processing power of the system was concentrated on a single FPGA on the Elphel camera — the microcontroller on the Elphel was slow and not in the image processing path, and the Nokia N800 could not control the camera settings or the lens controller (connected to the Elphel's serial interface) fast enough to allow it to do the image processing and control loops. As a last straw, it took two students a long three months to reimplement an existing, straightforward real-time frame alignment algorithm [63] in Verilog for the FPGA. It was clear that the F1 prototype was far too limited, and indeed did not meet most of the requirements listed in Chapter 3. A successor was needed.

The successor was named the F2 Frankencamera. For the F2, a member of the Texas Instruments OMAP3 [60] family was selected as the main processor. The OMAP3 is a system on a chip (SoC), a popular solution for cell phones and other small mobile devices.

An SoC contains nearly all the hardware blocks and interfaces needed for a typical mobile device, including a CPU, GPU, a display controller, and many I/O interfaces. While other manufacturers make similar SoC products (such as the Snapdragon line from Qualcomm), discussions with research partners at Nokia made it clear that the Nokia N900 smartphone would be built around the OMAP3 chip. The N900 is the successor to the N800 tablet used in the F1 Frankencamera, and like the N800 is an open platform running Linux.

Most importantly, however, the OMAP3 contains a direct camera interface and an ISP connected to it, with extensive public documentation available for both. Therefore, it made sense that the F2 should be built using the OMAP3-series of chips, so that when released, the N900 would provide a second platform with minimal extra reimplementation cost. The Nokia N900 is a consumer device that can be purchased by anyone and easily converted to a Frankencamera platform. The F2 allows for reconfigurability and customizability that the N900 as a consumer platform does not provide. Both fulfill the core Frankencamera architecture requirements.

Since both platforms use the OMAP3 series of chips, it will be covered next. Following that, the OMAP3's image signal processor (ISP), which receives and processes data from the image sensor, will be looked at in more detail. With the the basic capabilities of the system covered, the details of the F2 and the N900 will then be covered in more depth.

5.1 The TI OMAP3 System on a Chip

Since the design of the OMAP3 [60] is not part of the contributions of this dissertation, the chip will be explained in just enough detail to make its capabilities clear. The image signal processor (ISP), the most important part of the OMAP3 from the perspective of a camera control system, will be explored in the most detail.

The OMAP3 is a family of System-on-a-Chip (SoC)-style chips, designed to have as much of the needed functionality for a mobile device like a smartphone integrated onto it

as possible. The OMAP3 variants in both the F2 and the N900 include an ARM processor, a DSP core, a GPU unit, an ISP for the camera interface, and numerous I/O interfaces.

The main application processor inside the OMAP3 is an ARM Cortex A8 [64], a superscalar design running at 600 MHz. In addition to supporting the standard ARM instruction set, the Cortex A8 also supports a SIMD vector instruction set called NEON [65], which can greatly accelerate image processing applications (See Section 6.4.1 for an example). Since regular and NEON instructions can be mixed freely in a program, the NEON unit is the simplest way to accelerate image processing beyond what the base ARM architecture can provide.

The DSP core in the OMAP3 is a TI TMS320C64x+ core [66], a fixed-point VLIW design that can process up to four 16-bit multiply-and-accumulate instructions per cycle. In the OMAP3, the DSP runs at 520 MHz. As an independent processor from the Cortex A8, the DSP provides substantial additional processing power. However, using it is not straightforward, as programs for it must be loaded from the main processor, and data sharing between the two cores is not trivial. Therefore, it is not suitable for small tasks or those with large amounts of back-and-forth communication. Instead, it is best used for tasks like video or JPEG compression, where large buffers of data are moved at a time, and the DSP mostly operates independently.

For 3D output, the OMAP3 uses a PowerVR SGX GPU by Imagination Technologies. It supports OpenGL-ES 2.0 [67], allowing for programmable vertex and fragment shaders. Separately from the GPU, the OMAP3 has a display controller with 2D compositing features, including video overlay support. The overlay provides an efficient way to display the viewfinder video stream.

5.1.1 OMAP3 Image Signal Processor

The ISP in the OMAP3 is a fixed-function image processing pipeline, with several stages, each with varying degrees of configurability. It can be divided into three sections: the

CCDC, the preview pipeline, and the statistics collection units. For full details, see the OMAP3 technical reference manual [60].

The CCDC is the OMAP3's camera hardware interface. It receives digitized pixel data directly from an image sensor (or an analog-to-digital converter if the sensor does not include one), either in parallel or serial form. It can also generate a clock signal for the sensor, along with other timing and synchronization signals. It can handle images up to 4096x4096 pixels in size, and its peak reception rate is 75 megapixels/sec for 10-bit raw pixel data.

The CCDC can perform a number of optional operations on the input raw pixel stream, including:

- Estimating and subtracting the average pixel offset using masked sensor pixels.
- Subtracting a fixed per-color channel black-level offset from each pixel.
- Interpolating over faulty pixels based on a lookup table of faulty pixels stored in main memory.
- Correcting for vignetting, using a downsampled gain map from main memory.

After these operations, the image data can be written either directly to memory, or passed to the preview pipeline for further processing. Direct storage of raw data to memory is necessary to fulfill the Frankencamera architecture requirement that the ISP must be able to provide unprocessed data to the application.

On the first versions of the OMAP3 chips, hardware bugs in the ISP [68] prevent the proper use of the vignetting correction unit in the CCDC. As a result, the current Frankencamera implementations do not perform vignetting correction in hardware.

The preview pipeline is designed to quickly convert incoming image data for display, or for storing as a JPEG-encoded image. It takes in raw pixel data, typically 10-12 bits per pixel (bpp), and outputs 8 bpp RGB images. There are multiple configurable stages in the preview pipeline, much like the CCDC. These include:

- Dark-frame subtraction
- Horizontal median filtering/noise removal
- Color filter array interpolation (demosaicking)
- Color correction with a 3x3 matrix (white balance)
- Gamma correction with a lookup table
- Configurable RGB-to-YCbCr conversion
- Sharpening and false-color removal
- Resizing the image to between 0.25x - 4x of original size

The preview pipeline outputs images in the YCbCr color space, with the chroma channels (Cb and Cr) averaged down by 2 horizontally, known as the 4:2:2 format. This chroma downsampling cannot be disabled. The OMAP3 display manager supports the same YCbCr 4:2:2 format for hardware overlays, so the preview pipeline output meets the Frankencamera architecture requirement for a format suitable for viewfinder display.

The statistics modules operate on the pixel data after processing by the CCDC, so they operate on non-demosaicked values. The modules include an autofocus unit, an auto-exposure/auto-white balance (AE/AWB) unit (not covered here), and a histogram unit.

The autofocus unit runs all pixels through a pair of configurable IIR filters, typically to detect sharp edges in the image. The filter responses for neighboring pixels are averaged together into a low-resolution sharpness map, which is then sent to main memory.

The histogram unit constructs a histogram per color channel, and has a configurable number of buckets. Multiple non-overlapping regions can be defined, each producing an independent histogram.

This covers the essential features of the OMAP3 family. Its capabilities cover the processing needs of a Frankencamera — what is needed next is the sensor, optics, user

Figure 5.2: The F2 Frankencamera. A portable 5-megapixel computational camera that uses Canon lenses.

interface, and other supporting hardware to make a complete camera. Here, the F2 and the N900 diverge. First, let's take a closer look at the F2.

5.2 The F2 Frankencamera

Shown in Figure 5.2, the F2 meets the goals of the Frankencamera architecture: A portable, battery-powered camera with full control over the imaging pipeline. As a basic principle,

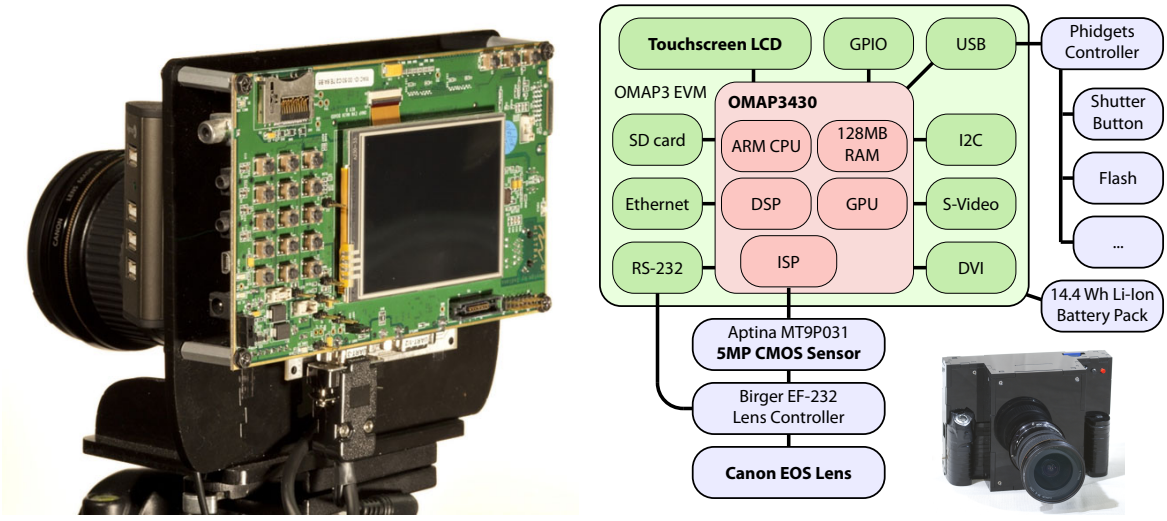
the F2 is built from off-the-shelf components and software. This not only made it faster to design and build, but also makes duplicating it much more straightforward. In addition, using common components allows much of the operating software of the F2 to be sourced from existing projects, minimizing the software engineering effort required for the camera.

The core of the F2 is the Mistral Solutions OMAP35x evaluation module (EVM) [69], a complete OMAP3-based system commissioned by TI for helping manufacturers to evaluate the OMAP3 and to begin software development before their own hardware is available. Figure 5.3(a) shows the stripped-down F2 development system, with the EVM board attached to the laser-cut plastic frame.

The EVM board has a 640x480-pixel resistive LCD touchscreen, and several I/O interfaces, including Ethernet, USB, RS-232, an MMC card slot, and audio/video input and output. It has 128 MB of DDR SDRAM, and 256 MB of on-board Flash memory. Figure 5.3(b) diagrams the features of the EVM.

The EVM also features two large expansion headers which carry the OMAP3 camera interface wires, along with many other signals. The availability of the camera signals is the main reason the EVM was selected over the other OMAP3-based option available at the time, the BeagleBoard [70]. The BeagleBoard is a low-cost OMAP3 system intended for hobbyist use, with a large community of users. Unfortunately, the BeagleBoard does not make the OMAP3 camera interface available, so it could not be used as the base of the F2.

The EVM relies on an external power supply, and has no image sensor or the rest of the hardware needed for a camera. It also has little support for a physical user interface — while there is a grid of buttons next to the touchscreen, these are not easily made accessible when the EVM is placed in a case. And since the EVM is a bare circuit board, a case needs to be built both for environmental protection, and to support all the camera



(a) The F2 development system, showing the main board.

(b) Block diagram of the major F2 components

Figure 5.3: F2 components. The F2 is built around the OMAP3 SoC on the TI OMAP3 evaluation board (EVM). The optical path consists of an Aptina MT9P031 5MP image sensor on an Elphel 10338 sensor board, a Birger Engineering EF-232 lens controller, and, typically, a Canon 10-22mm EF-S lens. In addition, a LiPoly battery pack and power supply allow for untethered operation, and a Phidgets interface kit allows assorted hardware to be connected. The EVM provides serial, USB, and Ethernet links, as well as a touchscreen.

components. Each of these design issues is discussed below in its own section.

5.2.1 Image sensor

In general, the larger the dimensions of the image sensor, the higher-quality images it produces, especially in low-light conditions. This is simply due to the larger sensor having more surface area per pixel, and thus being able to collect more total photons for the same exposure period. So at first glance, it would make sense to equip the F2 with at least a full-frame, 36x24 mm image sensor, like those found in high-end dSLR cameras.

However, this turns out to be quite challenging, for several reasons. First, finding a source for such sensors is difficult. Most camera manufacturers do not sell their sensors to outside parties, and even those that do typically require a substantial purchasing commitment, well outside the capabilities of a university research group. Next, many of the sensors that are available for sale do not support low-resolution, high-frame-rate modes needed to support an electronic viewfinder. For example, generally available full-frame Kodak CCDs like the KAI-11002 [71] do not support frame rates higher than 5 fps. Finally, many such sensors focus purely on maximizing image quality, and thus leave all support circuitry to the rest of the system, including timing control and analog-to-digital conversion. Designing an imaging board to support such sensors is a time-consuming and difficult task. While high-quality output is highly desirable, the goal of the F2 design was to show that the Frankencamera architecture works well in practice, and spending a great deal of extra design effort to improve the sensor beyond the baseline requirements was not attractive.

In the end, the decision was made to use the same image sensor as used in the F1: The Aptina MT9P031 5-megapixel CMOS sensor [72]. The MT9P031 is aimed at the cellphone market, so it is highly integrated: One simply needs to provide it a clock signal and power, and it begins to output 12-bit pixel data on a parallel data port. Its interface matches that of the OMAP3 ISP well, and it can produce 5-MP images at 14 fps, and 640x480 video at 53

fps. Its main drawback is its small size: 5.7 x 4.28 mm, roughly 1/36 of the surface area of a full-frame sensor. This means its low-light performance is far from that of a commercial dSLR camera.

As a CMOS sensor, the MT9P031 has several readout controls. It can subsample or average together neighboring pixel blocks for a 2-8x reduction in output resolution. It can also read out a region of interest (ROI) instead of the whole sensor, for increased frame rate, and it has integrated black-level correction and analog gain. Also supported is a global-reset exposure mode, as well as the typical rolling-shutter mode — in global-reset mode, a mechanical shutter is needed to guarantee equal exposure times for all pixels. It can also operate in snapshot mode, where each exposure is triggered either through a register write or an external signal. Accessing the sensor registers is done over the standard I^2C bus, operating at 400 kHz.

The Elphel 10353 network camera used in the F1 has two circuit boards: the main processing board, and a small sensor tile (the 10338 board). The sensor tile has a standard flex cable connector on it, which carries the needed sensor interface signals and power. This sensor board was adopted for the F2. All that was needed was a small board to couple the OMAP3 EVM expansion connector to the sensor tile flex connector. In addition, the OMAP3 I/O lines use 1.8 V signaling, while the Elphel sensor tile requires 2.5 V, requiring level translation circuitry.

A simple sensor I/O board was designed to bridge this gap. It consists of three 8-bit unidirectional level translator chips (the Texas Instruments SN74AVC8T245) for the camera data bus and timing signals, and one 4-bit bidirectional open-drain-compatible level translator (the Texas Instruments TXS0104E) for the I^2C bus. Beyond the level translators, the board simply contains a header to connect to one of the EVM expansion connectors, and a flex cable connector for the Elphel sensor board. Figure 5.4 has the schematic of the board.

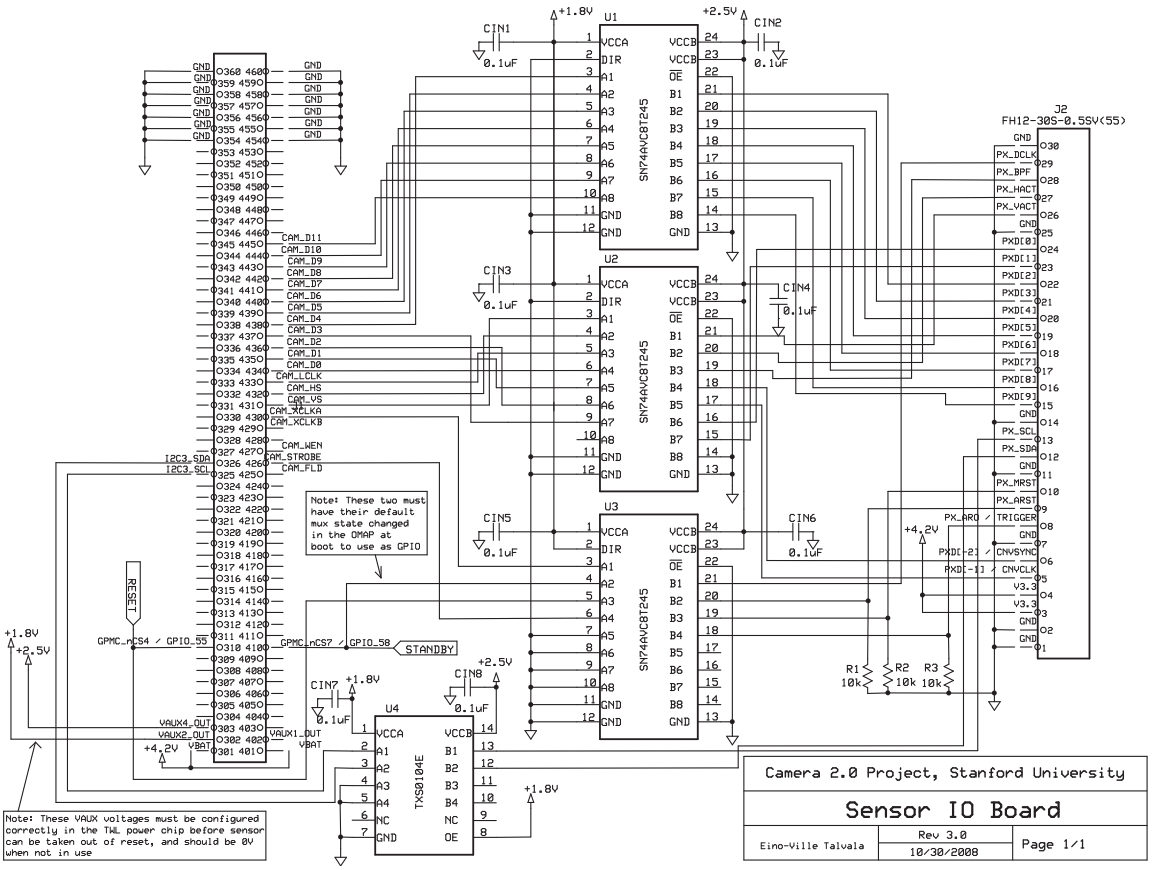


Figure 5.4: The F2 sensor I/O board. On the left is the EVM expansion connector, and on the right is the flex cable connector to the Elphel 10338 sensor board. In the center are three 8-bit unidirectional level translators for camera pixel data and timing signals, and one 4-bit bidirectional level translator for the I^2C bus to configure the sensor.

5.2.2 Optical path

The final essential camera components include a stable mount for the image sensor board, and a lens. The Elphel 10353 camera enclosure has a front plate with a C-mount lens adapter and an infrared filter, onto which the 10338 tile attaches. This could easily be adapted for the F2 design. The C-mount is a 1-inch diameter lens mount used commonly in security cameras, 16-mm film cameras, and some machine vision systems. However, typical C-mount lenses are fully manual, both for aperture adjustment and focusing.

While motorized machine-vision lenses for the C-mount exist, they often require bulky external controllers, infeasible for a portable device.

However, the Birger Engineering EF-232 lens controller, which attaches to a C-mount adapter, is capable of mounting and controlling most Canon EF or EF-S lenses. The Canon EF lens series is designed for Canon's SLR and dSLR cameras, and has built-in motors for aperture and focus control. The EF-S lens series is used by Canon's smaller dSLR cameras. The lenses are controlled through a Canon-proprietary electrical interface, which has been largely reverse-engineered by Birger. The EF-232 is controlled over a standard RS-232 connection, using a simple ASCII command set [73]. With it, any of the wide range of Canon lenses can be mounted and controlled by the F2.

Unfortunately, it isn't reasonable to use most of Canon's lenses with the F2. Because the MT9P031 is so small, it only sees a fraction of the image produced by a Canon EF lens, which is designed for a full-frame sensor. This cropping effect means that the field of view is roughly 6x narrower for the F2 compared to a regular Canon camera using the same lens. With such a large crop factor, only a few of Canon's widest-angle lenses can be used to provide a roughly standard field of view. The Canon EF-S 10-22mm USM lens is the shortest focal length lens Canon makes; mounted on the F2, it provides a field of view range between 31.8° and 14.7°. By comparison, a standard 50mm lens on a full-frame camera produces a field of view of 39.6°, so even with the widest available lens, the F2 has a slightly narrow field of view. Of course, for applications requiring narrow fields of view, the F2 is an excellent choice. Additionally, since only the center of the image produced by the lens is used by the F2, which is typically the sharpest and least distorted area, the image quality produced by the F2 optical path is excellent.

The follow-on design to the F2, the F3 Frankencamera, will use a full-frame 24x24 mm sensor, which will correct this mismatch and allow the full range of Canon lenses to be used.

5.2.3 Power supply

The F2 inherits the power supply design from the original F1 Frankencamera. The design is simple, with a pair of 2000 mAh lithium-polymer batteries in series connected to a power regulator board. The batteries have a safe continuous discharge rating of 4 A. Lithium-polymer batteries have a nominal output voltage of 3.7 V, with a usable voltage range between 3.0 and 4.2 V, depending on charge levels. Once the battery terminal voltage drops to 3 V, it is important to cut off the load to the battery; otherwise, it will be damaged and cannot be recharged. This leads to an overall low-voltage cutoff of 6 V for the pair. Additionally, being able to power the camera from an external source is also very useful during testing. 12 V power supplies are easy to come by, so the power regulator board is designed to accept input voltages between 6 and 12 V. A relay-based cutoff circuit completely disconnects the batteries from the regulator board once the battery voltage level drops to 6 V.

The regulator board provides three outputs, 3.3 V, 5 V, and 6 V, each capable of supplying 2.5 A at maximum. The 5 V supply is used to power the EVM board and through it, the image sensor. It can optionally be used to provide power to USB peripherals. The 6 V supply powers the Birger lens controller. The 3.3 V line is unused in the F2, but could be used to power novel hardware add-ons.

Each regulator output is generated using a National Semiconductor LM3102 step-down regulator, each of which can supply 2.5 A of output current. The power supply has no recharge capability, so an external battery charger is needed for recharging the batteries. Figure 5.5 shows the schematic of the power supply board.

5.2.4 Physical UI and case

The design of the F2 case and buttons was done by David Jacobs, and is included here for completeness.

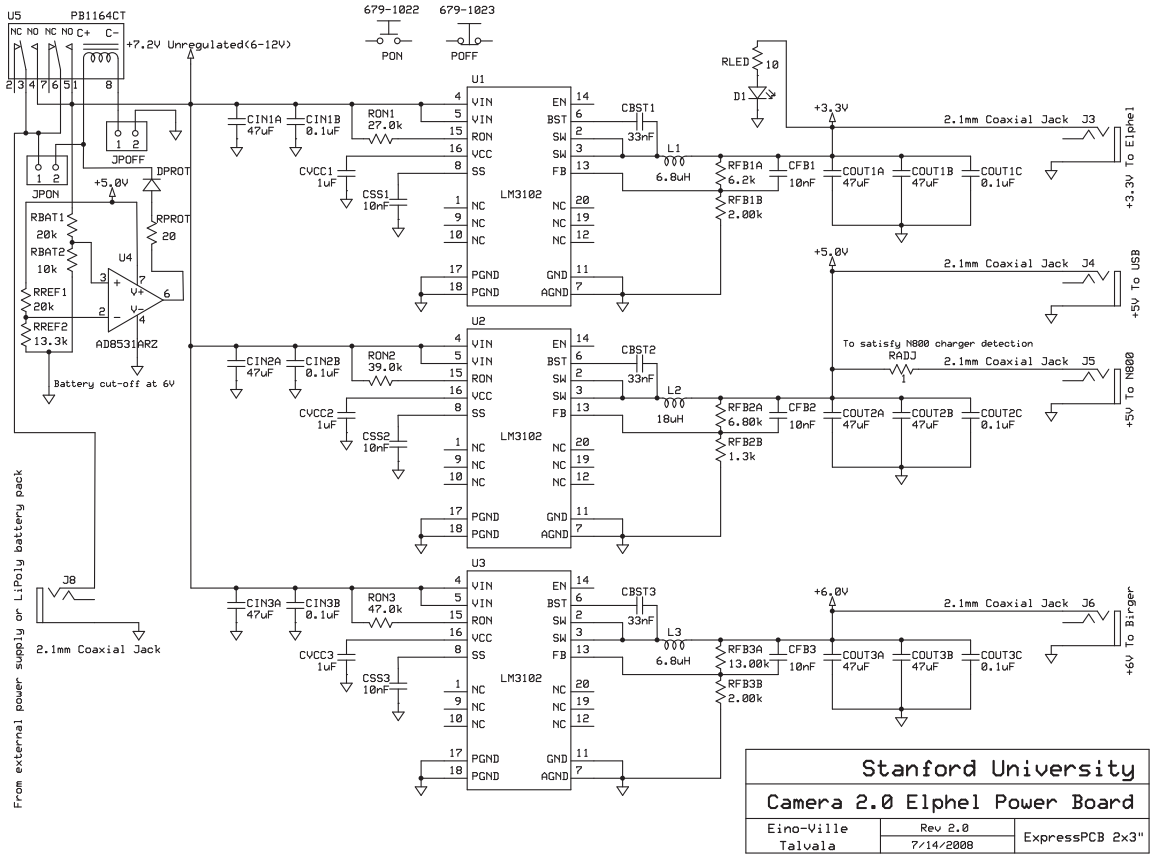


Figure 5.5: The F2 power supply schematic. Powered by a pair of LiPoly batteries, the power supply provides 3.3, 5, and 6V outputs. Battery low-voltage cutoff circuitry is at the top left, and each of the 3 voltage outputs is generated using an LM3102 chip at center, with associated discrete components on the right.

Novel camera applications are likely to be best served by novel user interfaces. Therefore, it is important that the camera can easily be modified to have additional buttons, switches, and so on. While the F2 has a USB port, directly connecting simple electronics to a USB port is not possible. Instead, the F2 body includes a Phidgets [74] interface kit, connected over USB. The Phidgets board provides a number of analog and digital inputs and outputs, along with a straightforward API for programming them. Phidgets are used for the shutter trigger button on the right handle of the F2, for the two touch-sensitive sliders on each side of the touchscreen, and for triggering flashes connected to

the detachable flash hot shoes.

The main structural element is a single aluminum bracket, which ties together the EVM board, the tripod attachment point on the base of the camera, and the sensor board and lens mount. The rest of the case is built from laser-cut acrylic. The power supply board and the Phidgets board attach to the acrylic case, along with power buttons and the shutter button. The case can be quickly taken apart to reach any part of the internals, for quick debugging and modification.

5.2.5 System-level software

While the Frankencamera API will be discussed in the next chapter, the F2 also needs a base operating system and system software. The Linux operating system is a natural choice, given its open nature and availability of embedded distributions for it. Specifically, several Texas Instruments and Nokia engineers have contributed a great deal of code to the Linux kernel to enable support for the OMAP3 family, including a driver for the OMAP3 ISP, and out-of-the-box support for the OMAP3 EVM board. Since the OMAP3 has virtual memory support, there are no real differences between the functionality available on Linux between the F2 and a standard desktop Linux system.

The operating system needs quite a bit more than just a kernel, however. The F2 uses the Ångström Linux distribution [75], an embedded Linux distribution which can be built for many different platforms. Ångström includes a centralized package management system, so additional programs can be trivially downloaded and installed on the F2, as long as it is connected to the network. Any programs not included by default can often be easily compiled for the F2 using the Ångström cross-compiler on a desktop Linux system.

The entire Ångström distribution is built directly from source code for a given platform, with a hierarchy of recipes defining the included software packages and configuration. Building the entire OS takes several hours on a typical desktop system. The customizability of Ångström allows the F2 to include all the needed software libraries

for development without any additional configuration. The kernel and the rest of the operating system is stored on the SD card plugged into the F2.

While Ångström supports the X11 windowing system, to minimize overhead, graphical applications for the F2 instead use the Qt/Embedded framework [76], which allows programs written using the Qt GUI libraries to directly write to the display. A desktop Qt application can often be recompiled for Qt/Embedded without modification. In general, the Qt framework provides most of the needed libraries for camera applications, outside the camera control API itself, including GUI support, threading, and networking.

Software development is best accomplished by connecting the F2 to an Ethernet network. The camera runs an SSH server, allowing developers to cross-compile their code on a desktop machine, and then remotely log into the F2 to test their programs. Standard Linux debugging tools such as `gdb` can be run directly on the F2 over the SSH link. Low-level boot modifications or kernel hacking require the use of one of the F2's serial ports for accessing the boot loader and reading kernel debugging messages.

The F2 uses a modified version of the Linux 2.6.28-omap kernel. The OMAP3 ISP drivers, not yet merged into mainline Linux, are added in from the Nokia N900 kernel tree. To operate the F2 camera systems, additions were made to the EVM board support files for powering up and configuring the sensor I/O board and the image sensor — this primarily required configuring the EVM power manager chip to turn on some of its auxiliary voltage outputs, and changing certain OMAP3 output pins to general-purpose I/O mode, to use as reset/standby signals to the MT9P031.

The largest missing component in the kernel tree was the driver for the MT9P031 sensor itself, which had to be written from scratch. It is a subsidiary driver to the OMAP3 ISP driver, responsible for configuring the image sensor to new modes as needed, and for calling the board-specific power management code. Both drivers use the Video for Linux 2 (V4L2) API, the Linux video capture kernel API [53]. The MT9P031 driver exposes a number of V4L2 controls for adjusting sensor settings such as exposure time, frame

duration, gain, and sensor readout parameters. It is also responsible for making sure any requested settings are valid for the sensor, and for adjusting any values that are out of bounds.

The FCam API, which provides the Frankencamera architecture-compatible camera control API, sits on top of this software stack, and will be discussed in detail in the next chapter.

5.3 The Nokia N900

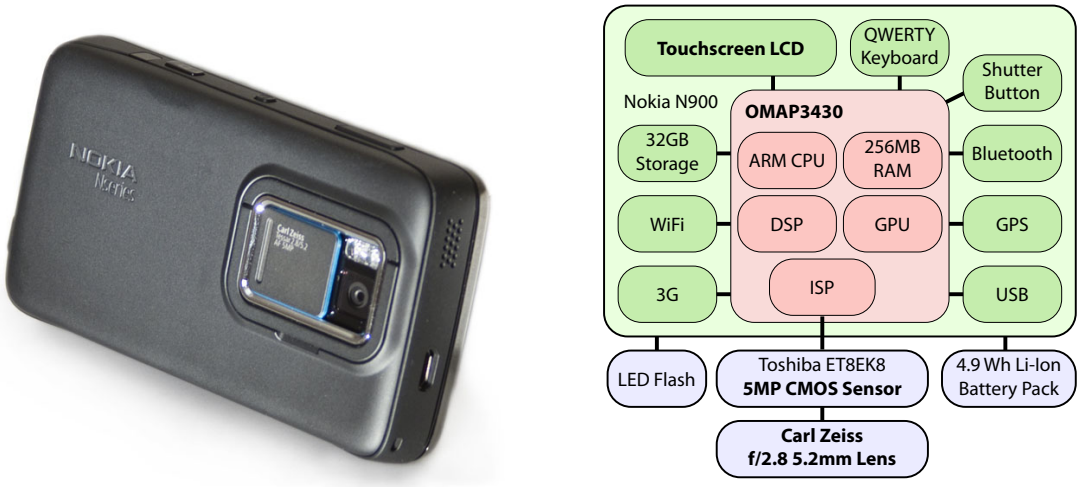
The Nokia N900 (shown in Figure 5.6(a)) is the newest member of Nokia's Linux-using internet tablet line, and the first to be an actual cell phone. Like its predecessors, the N900 uses the Maemo Linux distribution, Nokia's custom version of Linux targeted at small touchscreen devices.

Since the N900 also uses the OMAP3 chip, its basic capabilities are identical to those of the F2. They differ primarily in the available I/O interfaces, their extensibility, and the optical path. A block diagram of the N900 can be seen in Figure 5.6(b).

5.3.1 Hardware blocks

The N900 offers a variety of wireless interfaces, including a 3G cellular data connection, 802.11g WiFi, and a Bluetooth radio. It also contains a GPS receiver. It has one USB port for both charging and connecting to a host PC. Unfortunately, the USB port cannot be used in host mode — that is, the N900 cannot connect to other USB devices such as the Phidgets interface board, or USB flash drives. This substantially limits the expansion capabilities of the N900.

The resistive touchscreen on the N900 has a resolution of 800x480 pixels, and it also features a slide-out keyboard. A dedicated shutter button is located on the top-right of



(a) The Nokia N900 smartphone **(b)** Block diagram of the major N900 components

Figure 5.6: The Nokia N900. The Nokia N900 is a compact commercially available smartphone, built around the OMAP3430 SoC. Its optical path consists of the Toshiba ET8EK8 5MP image sensor coupled with a variable-focus, fixed-field-of-view Carl Zeiss lens. In addition, it has a high-resolution touchscreen; 3G, WiFi, and Bluetooth connectivity; GPS; and a device-mode USB port.

the device.

The N900 camera subsystem consists of the Toshiba ET8EK8 sensor; a Carl Zeiss-certified lens with variable focus distance, and fixed aperture and field of view; and a white LED for flash. The ET8EK8 [77] is a 5-megapixel sensor fairly similar to the Aptina MT9P031. It is a 10-bit CMOS sensor in the 1/2.5" form factor aimed at cellphone platforms, like the MT9P031. It is slightly less flexible than the MT9P031 in the F2, with no region-of-interest control beyond a digital zoom option. However, it also offers various subsampling and averaging modes, allowing for 640x480 output at 48 fps, and full resolution output at 13 fps. It has a built-in A/D converter and outputs pixel data over a high-speed serial link to the OMAP3.

The lens is integrated together with the sensor into a compact camera module. The module has an aperture of f/2.8, and a focal length of 5.2 mm, resulting in a field of view of 57.5°. Since the sensor size is roughly equal to the focal length, rays to the edge of the image sensor must travel at a large angle to the optical axis, and suffer substantial vignetting as a result. Because the lens is small and very lightweight, the focal plane can be moved very quickly, covering the entire focal range of 0 to 20 diopters (infinity to 5 cm focal distance) in 34 milliseconds.

The flash unit is a high-power white LED, which can be toggled on and off at will. While simple to program, it also is not bright enough to work well as the only source of scene illumination in many cases.

5.3.2 System software

The Maemo 5 Linux distribution on the N900 is mostly open, and freely modifiable. In terms of the camera interface, Nokia encapsulates the metering and focusing algorithms inside a closed-source user-space camera daemon with special access to the V4L2 interface. In addition, the code to the default camera application is not available. For the purposes of the FCam API described in the next chapter, this daemon is disabled each time an FCam application starts up, to prevent it from adjusting sensor parameters unexpectedly. Maemo uses a package management system, with a community software repository called maemo-extras. Software passing basic checks and receiving sufficient tester votes can be promoted into this repository, and then become available to N900 users worldwide. This allows Frankencamera programs to be run by anyone with an N900 after a one-time reboot to install the modified kernel modules, as described in Chapter 6.

Maemo uses the X11 windowing system, with a custom touch-friendly interface. Like on the F2, many Linux program can simply be recompiled for ARM, and will work right away, possibly with some customization to account for the touch interface.

Developing software for the N900 can be done either through the Maemo 5 SDK [78],

which sets up a semi-virtual development and cross-compilation environment called scratchbox inside a Linux system. This environment duplicates the software environment inside the N900, including access to the Maemo software repositories. Or, especially if using the Qt framework, Nokia has released a Nokia Qt SDK [79] with support for Maemo. The SDK is available for Windows, Linux, and Mac OS X, and allows IDE-driven programming of Qt applications. From the IDE, applications can be built, deployed, and remotely debugged on the N900 with a push of a button.

Neither the F2 nor the N900 is particularly unique as a digital camera, nor does either offer any unusual hardware features — however, they are both open platforms, with no cordoned-off sections in the imaging pipeline. This makes it possible to implement the Frankencamera architecture on both devices in software, which is the topic of the next chapter.

Chapter 6

The software

The application programming interface (API) for the Frankencamera platforms is called FCam. It is intended for application developers, and focuses nearly exclusively on the task of camera control. It also includes an optional basic image processing pipeline to make writing a complete camera program straightforward. FCam is not meant to be a large framework, and the application writer can choose any libraries they wish for user interfaces, display, networking, and other needs. FCam aims to have minimal dependencies on extra libraries, to maximize its portability and flexibility.

In planning FCam, the key goals were to create a system that is as simple and unsurprising as possible. To that end, several iterations of the software stack were built on the F1 and early versions of the F2, both for system testing and for implementing sample applications. The final design is based on the experience gained from the early prototypes, and also from presenting the design to other researchers at many points during the iterative process. Such presentations quickly made it clear which sections of the API were unintuitive, and helped uncover unforeseen consequences of design decisions before they became fixed in stone. After several iterations, the original (somewhat baroque) design was whittled down to the simple core needed for a Frankencamera. The design and implementation of the FCam API and runtime, as well as the architecture as a whole,

was done in collaboration with Andrew Adams.

In the next section, the design of the API and its public interfaces are described in detail. The sections that follow focus on the details of the API implementation, first describing the platform-specific implementations for the F2 and the N900, and then discussing the innards of the cross-platform parts of the API.

6.1 The FCam API

The API is written in C++, selected because it allows for easy access to the operating system interfaces written in C. C++ also makes it easy to use low-level primitives for performance gains when needed, without losing the ability to cleanly abstract away a great deal of unnecessary detail. The final API would translate relatively cleanly to other high-level languages, such as Python.

The FCam API has very few visible dependencies. Mostly, it depends on the standard C++ template libraries (STL), including some features from the TR1 standard library additions [80], such as the reference-counted pointer class `shared_ptr`.

6.1.1 Key concepts

The FCam API follows the Frankencamera architectural model, modeling a camera pipeline that converts capture requests, called `Shots`, to bundles of captured image data and metadata, called `Frames`. Typical flows between elements of the API can be seen in Figure 6.1. The four basic object types in the FCam API are `Shots`, `Sensors`, `Frames`, and `Devices`. The next sections describe each of these in detail.

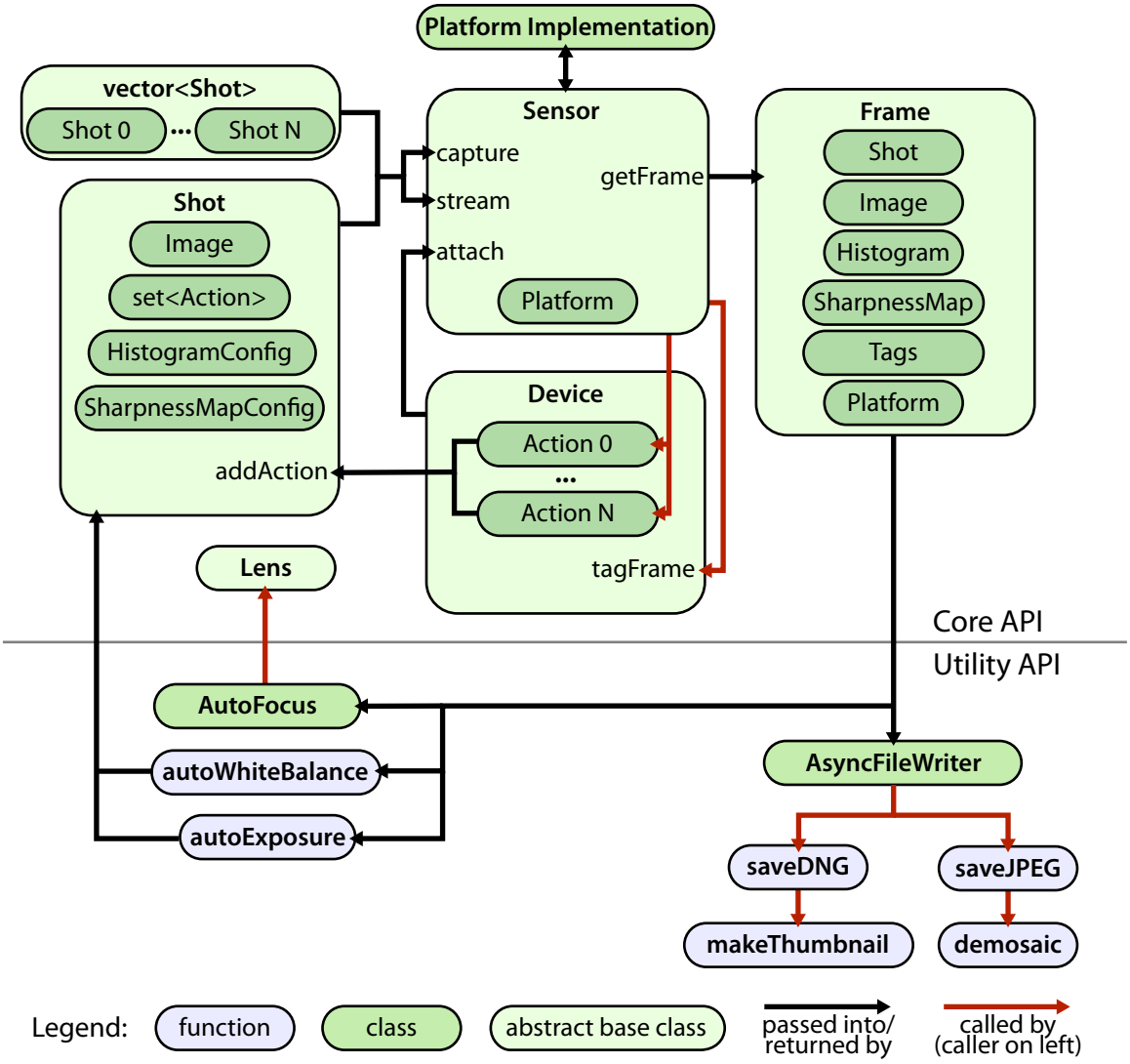


Figure 6.1: FCam usage model. This diagram shows how different elements of the API are passed into and out of methods and functions, as well as elements contained by others. In typical usage, `Shots` or `vectors` of `Shots` are passed into the `Sensor`, possibly with `Device Actions` included. The `Sensor` triggers the `Actions` as required during exposure, and allows attached `Devices` to add metadata `Tags` to nearly complete `Frames`. The Sensor provides completed `Frames`, which can be saved to disk using `AsyncFileWriter`, fed into algorithms like `autoExposure`, or used for application-specific tasks.

6.1.1.1 Shots

A `Shot` is a bundle of parameters that fully 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 configure frame rate independently of exposure time. `Shots` also include the set of `Actions` to be taken by devices during image exposure (as a standard STL `set`). Finally, `Shots` have unique IDs auto-generated on construction, which assist in identifying returned frames.

The example code in Listing 6.1 configures a `Shot` requesting 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, post-processed by the ISP to be in a YUV 4:2:2 format.

```
FCam::Shot shot;
shot.gain = 1.0;
shot.exposure  = 10000;
shot.frameTime = 33333;
shot.image = FCam::Image(640, 480, FCam::UYVY);
shot.histogram.enabled = true;
shot.histogram.region = FCam::Rect(0, 0, 640, 480);
```

Listing 6.1: Defining a Shot.

6.1.1.2 Sensors

After creation, a `Shot` can be passed to a `Sensor` in one of two ways — by `capturing` it or by `streaming` it. A `capture` call pushes a `Shot` into a request queue at the top of the imaging pipeline (Figure 4.1) and returns immediately:

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

Listing 6.2: Basic Shot capture.

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 pipeline is ready to begin configuring itself for the next request. For typical imaging pipelines, if the pipeline is ready, but the request queue is empty, then a blank request (a bubble) is necessarily pushed into the pipeline. The imaging chip cannot simply be paused until a `Shot` is available, because it has several internal stages; there may be a capture currently exposing, and another currently being read out. Bubbles configure the imaging chip to use the minimum frame time and exposure time, and the unwanted image data produced by bubbles is silently discarded before it is seen by the user.

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`. Calling `stream` copies the `Shot` into a holding slot alongside the request queue. Then, whenever the request queue is empty, and the imaging pipeline is ready for configuration, a copy of the holding slot is pushed into the pipeline instead of a bubble. That way, the imaging pipeline is always providing output at full sensor rate, without the application having to explicitly enqueue every request. `Streaming` a `Shot` is simple: `sensor.stream(shot)`.

A `Sensor` may also be asked to `capture` or `stream` vectors of `Shots`, or bursts, in the

same way that it captures or streams single Shots. Capturing a burst enqueues the Shots at the top of the pipeline in the order given, and is useful, for example, to capture a full HDR stack in the minimum amount of time. As with a Shot, streaming a burst causes the Sensor to make an internal copy of the burst, and then atomically enqueue all of its constituent Shots into the request queue whenever the imaging pipeline is about to become idle. Thus, bursts are atomic — the API will never produce a partial or interrupted burst. The code in Listing 6.3 makes a burst from two copies of the Shot, doubles the exposure of one of them, and then uses the stream method to create a video stream that alternates exposures 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 is one of the key requirements of the Frankencamera architecture. It is heavily exploited by the sample applications described in Section 7.1.

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

Listing 6.3: Burst streaming.

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 simply passes in a modified Shot or burst into stream again. Since the internal holding slot is atomically updated by the new call to stream, no partially updated burst or Shot is ever issued into the imaging pipeline.

Note that the specification of stream and capture allows for them to be interleaved at will. A capture request always has priority over stream, since the stream holding slot is only used if the request pipeline is empty. Therefore, a capture request will always be processed as quickly as possible, given the other requests already on the queue. It is important to note that this does not mean that a capture can interrupt a streaming

burst — such bursts are enqueued atomically, and a simultaneously issued capture will therefore either be enqueued before or after the streaming burst, not in the middle of it.

6.1.1.3 Frames

On the output side, the `Sensor` produces `Frames`, retrieved from a queue of pending captures via the `Sensor::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`, allowing easy matching between a request and the resulting data, which is needed for fulfilling Requirement 4. The code in Listing 6.4 displays the longer exposure of the two `Frames` specified in Listing 6.3, while using the shorter of the two to perform metering using the FCam auto-exposure utility function. The utility function modifies the exposure and gain of a `Shot` based on the metadata of the `Frame` passed to it. The function `displayImage` is a hypothetical function that is not part of the API.

In simple programs it is typically not necessary to check the `ids` of returned `Frames`, because the 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. In the current N900 implementation, as an example, image data may be lost if memory consumption rises high enough to require heavy access to the swap file, which interferes even with real-time priority threads. In

```
while (1) {
  FCam::Frame frame = sensor.getFrame();
  if (frame.shot().id == burst[1].id) {
    displayImage(frame.image());
  } else if (frame.shot().id == burst[0].id) {
    FCam::autoExpose(&burst[0], frame);
    burst[1].exposure = burst[0].exposure*2;
    burst[1].gain = burst[0].gain;
    sensor.stream(burst);
  }
}
```

Listing 6.4: Frame identification.

such cases, the V4L2 subsystem [53] may run out of space in its circular buffer, and start discarding image data before the FCam application can retrieve it.

6.1.1.4 Devices

In the FCam API, each hardware device such as a lens or flash is represented by a `Device` object with methods for performing its various functions. Each `Device` also contains a set of `Actions` which are used to synchronize these functions with image capture, and a set of `Tags` representing the metadata attached to returned `Frames`. A `Device` may be `attached` to a `Sensor`, which will then call the `Device` `tagFrame` method with every newly-built `Frame`. The `tagFrame` method adds the appropriate `Tags` describing the hardware state at the time of image capture to the `Frame` metadata. While the exact list of devices is platform-specific, the API includes abstract base classes that specify the interfaces to a generic lens and flash. Figure 6.2 illustrates using Actions to trigger two separate external flash units at different points of time in a single exposure.

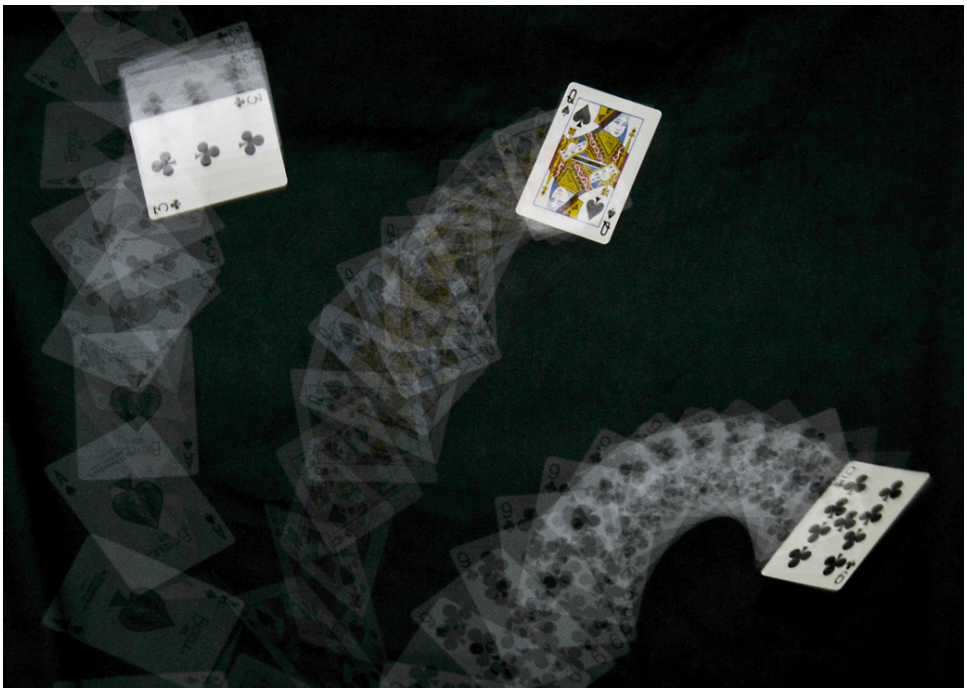


Figure 6.2: Synchronization control. The Frankencamera API provides precise timing control of secondary devices like the flash. To produce the image above, two Canon flash units were mounted on an F2. The weaker of the two was strobed for the entire one-second exposure, producing the card trails. The brighter of the two was fired once at the end of the exposure, producing the crisp images of the three cards. *Photograph and implementation by David Jacobs*

The lens. A `Lens` can be directly asked to initiate a change to any of its parameters — focal length, focus distance (measured in diopters) and aperture — using the three methods `setZoom`, `setFocus` and `setAperture`, respectively. These calls return immediately, and the lens starts moving in the background. Each call has an optional second argument that specifies the speed with which the change should occur.

For cases in which lens movement should be synchronized to exposure, the `Lens` defines three `Actions` to do the same. Additionally, each parameter can be queried to see if it is currently changing, what its bounds are, and what its current value is. The code in Listing 6.5 moves the lens from its current position to infinity focus over the course of two seconds:

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

Listing 6.5: Direct Lens use.

A `Lens` can tag each returned `Frame` with the state of each of its three parameters during that `Frame`'s capture. `Tags` can be retrieved from a frame as follows:

```
FCam::Frame frame = sensor.getFrame();
Lens::Tags tags(frame);
cout << "The lens was at: " << tags.focus;
```

Listing 6.6: Lens metadata retrieval.

which uses the convenience `Lens::Tags` structure, or if the `Tag` name is known, directly by:

```
FCam::Frame frame = sensor.getFrame(); cout << "The lens was at: "
    << frame["lens.focus"];
```

Listing 6.7: Alternate Lens metadata retrieval.

Diopters [1/m] are used as the units for focal distance, instead of distance in the scene, because typically a linear movement of the focusing lens element(s) causes a linear change in the focal distance in diopters. This means that it is far simpler to build a motion model of the lens focusing elements when using diopters rather than meters, and to report the maximum rate at which the lens can change focus (constant in diopters). For the same reason, the precision of the lens focal distance setting is constant in diopters, while it varies greatly when measured in meters. Using diopters also avoids having to deal with infinity as a valid setting — 0 diopters represents focusing at infinity, and the minimum focal distance is a reasonable value as well (for example, a 5 cm minimum focus distance translates to 20 diopters).

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

Listing 6.8: Device Action creation.

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. Flash units with more capabilities (such as the strobing flash used to generate Figure 6.2) can be implemented as subclasses of the base `Flash` class. The `Flash` can tag each returned `Frame` with its state, indicating whether it fired during that `Frame`'s capture, along with the settings it used to fire.

The code example in Listing 6.8 adds an `Action` to a `Shot` to fire the flash briefly at the end of the exposure (second-curtain sync).

Other devices. Incorporating external devices and having the API manage the timing of their actions is straightforward. One merely needs to inherit from the `Device` base class, add methods to control the hardware device in question, and then define any appropriate `Actions`, `Tags`, and `Events` (discussed in Section 6.1.6). This flexibility is critical for computational photography research, where it is common to experiment with novel hardware.

6.1.2 Overall API structure

Large sections of the FCam API implementation are highly platform-dependent, because they need to accurately model the timing behavior of the underlying hardware. But at the same time, the API itself aims to be easily portable to many platforms, so that only

the platform-specific code must be rewritten for a new FCam platform, and application developers can easily transition their code to a new platform. Figure 6.3 lays out the structure of the whole API.

To make extension as straightforward as possible, FCam heavily uses object-oriented polymorphism, through C++ abstract base classes (ABCs). FCam uses ABCs to specify the general API interface, and these base classes are then inherited by specific implementations. The API also heavily uses namespaces for organization, with the generic API interfaces and cross-platform implementation living in the `FCam` namespace, and specific implementations living in a sub-namespace, such as `FCam::N900`. For example, `FCam::Sensor` is the abstract base class of `FCam::N900::Sensor`, the platform-specific object.

In a specific implementation, the classes then implement the actual functionality behind the abstract interface, possibly adding in support for platform-specific functionality. So, for example, the `FCam::Lens` defines a generic lens control class, with a large number of virtual functions (see the FCam API project page [9] for details). The `FCam::N900::Lens` inherits from this base class, defining concrete implementations for all the base `Lens` virtual functions. Each implementation is free to add new methods and objects as well, for platform-specific features — for example, the Aptina sensor in the F2 has support for selecting a region of interest to read out, which is not part of the base FCam interface. So the F2 implementation, while still inheriting the basic interfaces, adds fields to `FCam::F2::Shot`, `FCam::F2::Frame`, and `FCam::F2::Sensor` to support this new feature. Section 6.3.1 has more discussion on what is needed to support a new FCam platform.

Using the abstract base class approach, it is easy to write generic algorithms and helper functions that will work for all implementations. For example, the auto-exposure routine needs to be able to access the `Frame` histogram on any platform. This can

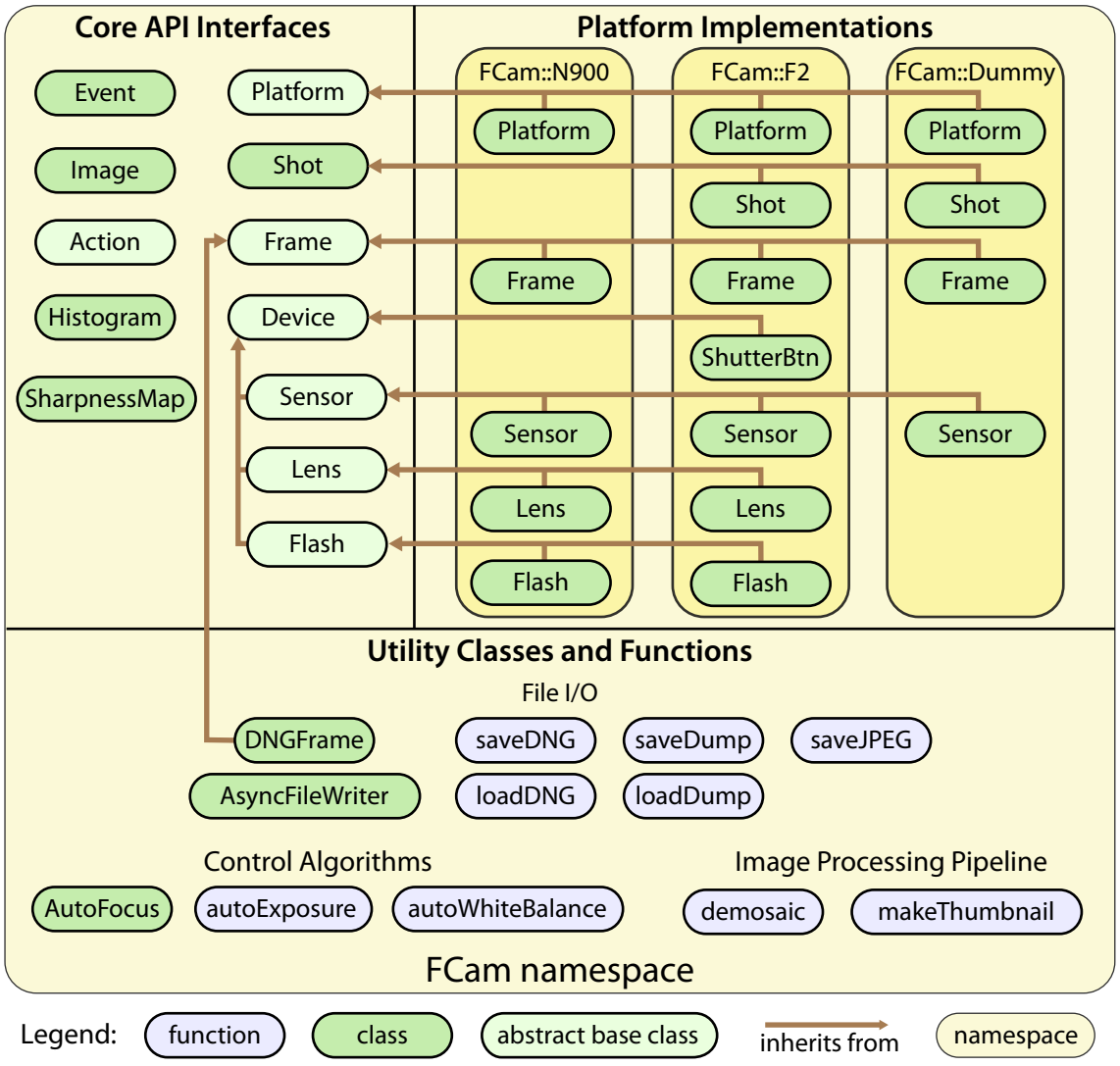


Figure 6.3: The FCam namespace. The FCam API can be divided into three sections. The core camera control abstract base classes define fundamental behaviors and requirements. The platform implementations inherit from these base classes and add on platform-specific features. The utility classes and functions operate on the abstract base classes, allowing one utility code base to be used for all the platforms. The `Dummy` platform simulates a simple camera system for application testing and debugging on development PCs. The utility interfaces include asynchronous file I/O support, a raw image processing pipeline, and basic control algorithms. Only key classes and functions are shown here.

easily be accomplished by having `autoExpose` take in a base-class `Frame` object as an argument: `void autoExpose(const FCam::Frame &f){...}`. Then an instance of any actual implementation of the `FCam::Frame`, say `FCam::N900::Frame`, can be passed into this function safely. The same argument applies to generic autofocus and the `FCam::Lens` object, or to entire sub-applications which need to manipulate the main `FCam::Sensor` object. So one can write an entire HDR stack capture routine without having to know or care about the specific platform that the routine will run on.

Outside of the core camera control classes, FCam includes a number of utility and image processing routines that take advantage of this organization, allowing them to be written once and work on all supported platforms.

6.1.3 Utility functions

The utility sections of the FCam API are designed to make it easy to write a complete camera application, providing file I/O, basic control routines, and a software image processing pipeline. However, the application writer is free to ignore all these, and write their own as needed.

6.1.3.1 Control routines

Nearly every camera program needs to adjust sensor exposure and gain to account for scene brightness, to adjust white balance to account for scene illumination, and to focus the lens on the scene. FCam includes two utility functions, `FCam::autoExpose` and `FCam::autoWhiteBalance`, and one utility class, `FCam::AutoFocus` for these tasks.

The `autoExpose` function uses a `Frame`'s histogram information to adjust the exposure and gain settings of a `Shot`. It tries to make sure that no more than 0.5% of pixels are oversaturated, and then tries to make sure that at least 2% of all pixels are in the top third of the histogram (these thresholds were chosen empirically). To increase image

brightness, it first increases the exposure time up to the maximum allowed by handshake, and then increases the gain until it is maximized. Finally, it increases exposure again until it reaches the user-defined maximum exposure limit. Reducing brightness follows the opposite path. A smoothness term slows down the exposure adaption, ensuring a smoothly varying exposure.

The `autoWhiteBalance` function also uses the `Frame` histogram, in order to estimate the scene illumination's color temperature. It uses a gray world approximation [81], which assumes that the average reflectance of a scene is gray. Specifically, the algorithm calculates the mean blue and red values of the scene, and then finds a color temperature at which they are equal. Note that `autoWhiteBalance` does not modify any image data, and merely updates a `Shot`'s white balance field. The white balance field is then used both by the ISP's hardware pipeline for adjusting incoming pixel data, and by FCam's software processing pipeline.

Finally, `AutoFocus` can be used to focus the lens on the scene. Unlike the other two algorithms, the autofocus algorithm requires keeping track of past history, encapsulated within the `AutoFocus` class. An `AutoFocus` instance is passed a pointer to the platform's `Lens` at creation, so that `AutoFocus` can move the lens as needed. Listing 6.9 shows an autofocus loop focusing on a target section of the scene.

`AutoFocus` uses the sharpness map generated by the ISP to determine the degree of focus in a frame, without having to analyze image data at all. It sweeps the lens across its whole focus range, and then jumps to the sharpest location found. The `Frame` metadata is used to find the lens focal distances for each frame as the lens does its sweep.

6.1.3.2 Raw image processing

Converting a buffer of raw sensor data into a usable image is a multi-stage process, with many tradeoffs between image quality and processing speed. FCam includes two

```
FCam::Sensor sensor;
FCam::Lens lens;
FCam::Shot shot;
... // Define shot parameters
shot.sharpness.enabled = true;

FCam::AutoFocus af(&lens);

sensor.attach(&lens);
sensor.stream(shot);
af.setTarget(targetRect);
af.startSweep();
while(!af.focused()) {
  FCam::Frame f = sensor.getFrame();
  af.update(f);
}
// Lens is focused at targetRect
```

Listing 6.9: AutoFocus example

conversion routines, both tuned more for speed than for quality. The `demosaic` function converts a high-bit-depth Bayer-pattern raw `Frame` into a 8-bpp RGB `Image` of the same resolution, with control over gamma correction, contrast enhancement, and black level. The `makeThumbnail` function converts a high-resolution raw `Frame` into a lower-resolution thumbnail `Image` for display on a viewfinder. It is much faster than `demosaic`, since it can start by downsampling its input. Details on the processing pipeline implementation and all the processing steps will be covered in Section 6.4.1.

Both of these routines only operate on `Frames`, not `Images`, as they need access to the `Frame`'s `Platform` data for color transforms and valid raw value ranges.

6.1.3.3 File I/O

It's a rare camera application that won't wish to save any image data for later processing or viewing by the photographer. To that end, FCam includes support for three image formats: JPEG, DNG, and a simple dump format.

JPEG [82] is the industry-standard lossy image format for storing processed photographs. It stores 8-bit RGB images using a DCT-based compression technique. FCam supports saving both raw and 8-bpp image data as JPEGs — raw `Frames` are first run through the `demosaic` routine. FCam does not currently support storing any metadata in the JPEG images, such as the industry-standard EXIF or IPTC fields, because the system libraries on the F2 and the N900 do not support such metadata. These libraries could of course be rewritten to add such support if desired.

For saving and loading unprocessed sensor data, FCam supports the DNG format [83]. DNG is an attempt by Adobe to define a common raw image format for the camera industry. Currently, most manufacturers use their own proprietary raw file formats, with some even encrypting sections of the image metadata. While most raw formats have been reverse-engineered [84], none besides DNG are officially documented and freely usable. Since many photographic post-processing packages, such as Adobe's Photoshop and Lightroom, and the open-source dcraw [84], support DNG, using it for FCam's native raw format makes sense. DNG is also effectively the serialization format for `FCam::Frame` — most `Frame` information and all `Tags` are stored in the DNG file, partly within a manufacturer-specific section, so that they can be restored accurately on load. The `makeThumbnail` function is used to generate an 8-bpp RGB preview image embedded in the DNG. The DNG format is based on the standard TIFF format, and if fed to a typical TIFF reader, an FCam DNG file would be read as just the thumbnail image. Section 6.4.3 has more details on the FCam DNG implementation.

As a rapid debugging and testing tool, FCam can also dump raw sensor data, with a minimal header, into a file. The dump format is easily supported by ad-hoc programs, and it can also be read by the ImageStack command-line image processing tool [85]. Since writing images to disk can take some time, FCam provides a background image saving routine through the `AsyncFileWriter` class. It has an internal queue of `Frames` to save, and processes them one by one in a background thread.

6.1.3.4 The Dummy platform

While existing mobile SDKs often include device emulators to test code quickly before deploying to a real device, these rarely include any support for camera use. This means any program that tries to access the camera typically simply fails with an error when run. However, it can be valuable to be able to run an FCam-based application on a development PC emulator. This can be used to quickly test changes to sections of the program not directly related to camera operation (such as the user interface), or to allow for development without the camera itself present. It can sometimes also be valuable to be able to replay a certain set of images through the application to experiment with how an algorithm under modification behaves with a consistent input.

For these purposes, FCam includes a simple `Dummy` platform. Unlike the other implementations that require real camera hardware, the `Dummy` platform works anywhere. The semantics of submitting `Shots` and receiving `Frames` are exactly the same, but the `Frames` returned by the `Dummy` contain either a test pattern image, or are constructed from DNG files loaded from disk. When producing test patterns, the `Dummy` platform simulates the effects of exposure and gain, and in both cases simulates frame duration properly so that applications receive frames at expected rates. There is no model for sensor noise, or provision for more complex test scenes (such as moving shapes, etc) — the `Dummy` isn't meant for replacing real-world testing, just for simplifying application development.

6.1.4 Platform information

Several common pieces of configuration information need to be accessible from multiple places in the API, and are fixed for a given platform. These include the sensor's color filter array arrangement, the color matrices for converting from the sensor's color space to sRGB, the valid range for the sensor's raw pixel encoding, and the manufacturer and

model name for the platform itself. These should be accessible through the `Sensor` class, since they are mostly its properties, but they also need to be accessible through individual `Frames`, since the information is needed for saving DNG files and for converting raw data to RGB images. Therefore, this information is centralized in the `Platform` class, and referenced from the `Sensor` and `Frame`. This separation also allows the DNG loader to easily expose the `Platform` data for `Frames` loaded from files, which have no `Sensor` object to query.

6.1.5 The Image class

Images are by far the largest objects handled by FCam, and in limited-memory mobile devices that are FCam's target platforms, they must be efficiently represented and handled. To that end, FCam's `Image` class is by default a reference-counted type, like the `Frame` — passing an `Image` to a function does not copy the `Image` data. An `Image` can also be used to wrap around a bare memory array created by some other library or OS call, with weak reference semantics. This allows hardware overlays, other libraries' image buffers, and similar objects to be handled by FCam's routines without any extra image copies. For example, a `Shot` can use an `Image` wrapped around the hardware overlay framebuffer as its target, resulting in the output `Frame` image data being copied directly on screen by the FCam implementation. Empty `Images` can also be created with two special modes: discard and auto-allocate. When used as a target in a `Shot`, the former tells FCam not to include the received image data in the output `Frame` at all — only the metadata is wanted by the application. The latter tells FCam to allocate a new `Image` each time a `Frame` is constructed, instead of the default of reusing the same target `Image`.

The `Image` class also supports image buffers where the memory stride between two rows of the image is larger than the width of the image. This is needed to handle OMAP3's hardware-rotated overlay frame buffers, and allows the creation of `Images` that refer to a subwindow of the parent `Image` with no copying of pixel data.

Finally, an `Image` can be locked by an application, which prevents the `Sensor` from overwriting its data if the `Image` is being used as a `Shot` target. This allows for modification of the overlay framebuffer, for example, without the danger that the next viewfinder `Frame` will overwrite it before display.

In general, FCam tries to keep memory allocation and deallocation from bothering the application developer any more than necessary, while providing sufficient control over memory usage to allow applications to run on embedded platforms with limited memory. To save on memory and to simplify object lifetime tracking, all FCam objects that may refer to large amounts of memory have reference semantics — copying them does not copy their internal state. This is also important from a performance perspective, because just copying a 640x480 image buffer 30 times a second consumes roughly 10% of the CPU on both the Nokia N900 and the F2.

6.1.6 Event and error handling

Since `Shot` requests are submitted to the `Sensor` asynchronously, there must be a way to query the FCam runtime about any failure conditions encountered during the processing of a `Frame`. In addition, many devices such as the shutter button or the lens may wish to report changes to their state — the button reporting being pushed down, and the lens controller informing the system that a new lens has been attached. These requirements are easily addressed by a basic event subsystem, which can be used both for reporting errors and for changes in device state. While most GUI libraries and frameworks include such event queues, one of the goals of FCam is to be as independent as possible from other libraries, so tying FCam to a specific GUI library is undesirable. Instead, FCam has its own small event system.

`Events` consist of an event type (such as “Error” or “Shutter Pressed”), a single integer of data, and a human-readable description of the event. They also include a time of occurrence, and a pointer to the object that created the `Event`. The event queue can be

accessed in two ways: The application can simply go through the queue entry-by-entry, or it can filter the event list by type, data, or creator. Filtering is necessary since multiple threads can be reading and writing to the queue, and a given thread may not wish to handle all possible events. For example, an application loading a DNG file would only want to receive errors relating to the loading process, not errors relating to image capture which may be occurring concurrently in a different thread.

6.2 A complete basic camera program

Using the API features described in Section 6.1 above, a complete basic camera program much like that described in the first case study (Section 3.2) is easy to implement. Listing 6.10 demonstrates such a program for the F2 Frankencamera.

The program displays a viewfinder, metering and adjusting white balance automatically. When the shutter is half-depressed, it runs an autofocus loop. Once autofocus completes, a full-resolution image can be captured by fully depressing the shutter. The only non-FCam function used in the program is the fictional `getFramebuffer` call, which returns an FCam `Image` encapsulating the hardware overlay framebuffer. Note that this function can be implemented, but its exact semantics depend on the GUI toolkit used by the application — the FCamera sample program described in Section 7.1.1 uses this technique with the Qt framework [76]. Using this `Image` as the target of a `Shot` allows for a live viewfinder display without any application-level image copying.

As its first action, the program creates an `Image` encapsulating the F2 display overlay framebuffer, which will be used for a viewfinder. Then, the program instantiates the F2 `Sensor` and `Lens`, and the shutter button. Next, it creates a file writer object for saving images, and an autofocus helper which stores a pointer to the `Lens` so it can control it. The program then defines a `Shot` for running a viewfinding mode, with the destination

```
#include <FCam/F2.h>
  ...
  < Initialize display >
  ...
  FCam::Image framebuffer = getFrameBuffer();

  FCam::F2::Sensor sensor;
  FCam::F2::Lens lens;
  FCam::F2::ShutterButton buttonListener; // For
      button events
  FCam::AsyncFileWriter writer;
  FCam::AutoFocus af(&lens);

  FCam::F2::Shot viewfinding;
  viewfinding.image = framebuffer;
  viewfinding.histogram.enabled = true;
  viewfinding.sharpness.enabled = true;
  viewfinding.frameTime = 33333; // 30 fps
      viewfinder

  FCam::F2::Shot photo;
  photo.image = FCam::Image(sensor.maxImageSize(),
       FCam::RAW, FCam::Image::AutoAllocate);
  int photoCounter = 1;
  bool capturePhoto = false;

  sensor.stream(viewfinding);
  while (1) {
    // Get the next frame from the sensor and...
    FCam::Frame f = sensor.getFrame();
    if (f.shot().id == viewfinding.id) {
      // ...use it to update exposure and white
          balance settings
      FCam::autoExpose(&viewfinding, f);
      FCam::autoWhiteBalance(&viewfinding, f);
      if (!af.idle()) {
        af.update(f); // Update autofocus helper
            if it is active
      }
      sensor.stream(viewfinding);
    } else if (f.shot().id == photo.id) {
      // ...save it to disk
      std::stringstream filename;
      filename << "photo_" << photoCounter++;
      writer.saveDNG(f, filename.str() );
    }
    // Check for button presses
    FCam::Event e;
    while(FCam::getNextEvent(&e)) {
      switch (e.type) {
      case FCam::Event::FocusPressed:
        if (af.idle()) {
          af.startSweep(); // Begin autofocus
              sequence
        }
        break;
      case FCam::Event::ShutterPressed:
        capturePhoto = true;
        break;
      }
    }
    // Check if a high-resolution image should be
        captured
    if (af.focused() && capturePhoto && writer.
        savesPending() < 4) {
      photo.exposure = viewfinding.exposure;
      photo.gain = viewfinding.gain;
      photo.whiteBalance = viewfinding.
          whiteBalance;
      sensor.capture(photo);
      capturePhoto = false;
    }
  }
```

Listing 6.10: A basic camera program.

image set to the framebuffer, and both histogram and sharpness map generation are enabled. To conclude initialization, the program then defines a `Shot` for high-resolution capture, with the output resolution set to maximum, and a new `Image` to be allocated for each capture request. Finally, the `Sensor` is set to `stream` the viewfinding `Shot`.

Inside its main loop, the program first fetches the next `Frame` from the sensor (blocking execution until such a frame is ready). It then identifies the `Frame` as resulting from either a viewfinding `Shot` or a high-resolution capture `Shot`. In the former case, the `Frame` is used to update the `Shot` exposure, gain, and white balance fields, and for updating the autofocus routine if it is active. The updated `Shot` is then passed to the `Sensor`. In the latter case, the `Frame` is saved to disk as a DNG file.

Next, the program checks for user input, and based on the state of the shutter button, possibly initiates an autofocus process, or the process of capturing of a high-resolution image. Finally, the program checks whether it needs to `capture` a high-resolution photograph, by confirming that autofocus has completed, that the user has requested high-resolution capture, and that the file writer queue isn't too full. If those conditions hold, it updates the high-resolution `Shot` metering settings, and tells the `Sensor` to capture that shot.

The program is quite short, but encapsulates all the key features of a typical point-and-shoot camera. The camera control flow isn't obscured by many low-level details, and this basic program can easily be expanded to include new functionality. Only a few functions outside of FCam are needed.

6.3 Platform-specific FCam implementation

As described in Chapter 5, both the F2 Frankencamera and the Nokia N900 platforms use Linux as their operating system, running on versions of the Texas Instruments OMAP3 SoC. Consequently, the implementations of the FCam API on the two platforms are

identically architected, with differences mostly due to the different sensors and other available hardware on the two platforms. They'll be described together, with differences pointed out as needed.

Additionally, several sections of the API implementation are cross-platform, working on any reasonably POSIX-compliant [86] system. These sections include the software post-processing pipeline, the file I/O functions including the support for the DNG format and the Dummy platform. Taken together, the cross-platform aspects of the FCam API make it possible to use FCam in desktop applications that read and further process Frankencamera-produced image data. These will be discussed in Section 6.4. Figure 6.4 shows the software layers and libraries involved in a complete FCam-using application.

Since the FCam API is built to be extensible to new platforms, it is worthwhile to discuss exactly what is platform-specific and needs to be implemented anew for every platform. The core implementation of FCam on the OMAP3 will be investigated next, and then the implementation of the hardware devices unique to each platform will be covered.

6.3.1 Writing a new implementation

In short, every FCam implementation needs to inherit and implement four abstract base classes: `FCam::Shot`, `FCam::Sensor`, `FCam::Frame`, and `FCam::Platform`. By convention, these specific implementations live in a namespace based on their platform, and their names match the base classes. So the N900 implementation defines an `FCam::N900::Sensor` that inherits and implements `FCam::Sensor`, and so on. The implementations are free to expose new capabilities, as long as they also implement the required base functionality as well. As discussed in Section 6.1.2, the API structure allows control algorithms to be platform-agnostic — the platform-specific `Shot`, `Frame`, and `Sensor` objects can be passed into functions expecting a base class type, and will work as expected.

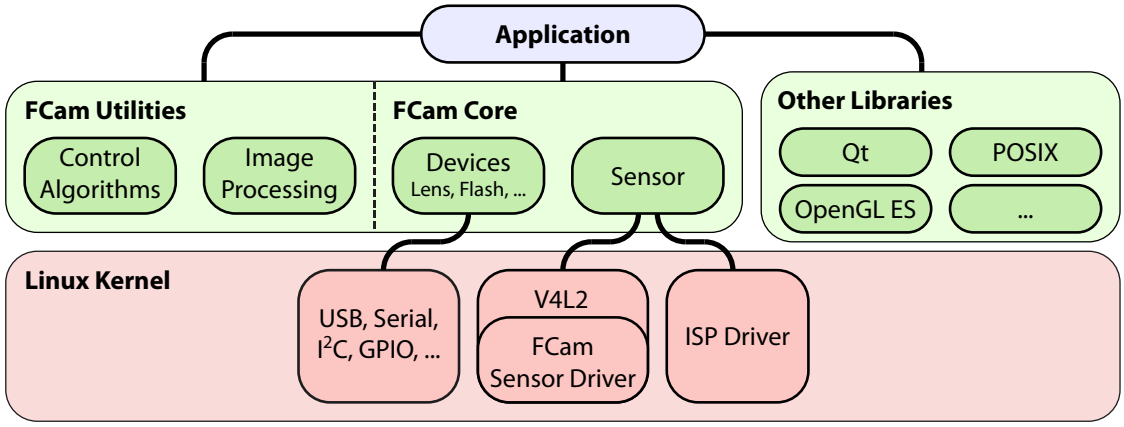


Figure 6.4: The FCam software stack. The platform-specific FCam implementations layer on top of the Linux V4L2 kernel API and the driver for the OMAP3 ISP, while the cross-platform implementation requires standard POSIX libraries. System and third-party libraries such as the Qt framework are needed for writing a complete application, since FCam concerns itself only with camera control and basic processing.

Beyond the core objects, each platform needs to expose implementations of the remaining camera hardware, such as lenses and flashes. These also live in the platform-specific namespace, and should also inherit from the corresponding API abstract base class, if one exists. So the `FCam::N900::Lens` inherits from `FCam::Lens` and so on, to maximize the ability of application developers to write generic code. All device implementations need to inherit from `FCam::Device` if no more specific parent exists, so that they can connect up to the `Frame`-tagging and `Action`-triggering mechanisms of FCam. Each `Device` should support `tagging` new `Frames` with relevant metadata through the `tagFrame()` method, and should implement useful `Actions` to synchronize with sensor exposure.

6.3.2 FCam on Linux and OMAP3

As already mentioned, FCam as an API attempts to be platform and OS-agnostic and aims to abstract away irrelevant platform details as much as possible. FCam implementations, conversely, are very platform-specific, since any FCam platform will likely have a different application processor, image sensor, or imaging pipeline, not to speak of all the other hardware needed to make a whole camera. For the rest of this chapter, "FCam" will primarily refer to the FCam API implementations on the Nokia N900 and the F2 Frankencamera, not to the API interface in general.

As much as possible, FCam is built on existing software infrastructure present in the Linux kernel. Primarily, this consists of the Linux kernel video API, Video For Linux 2 (V4L2) [53], and its associated drivers. Among these is the driver for the OMAP3 camera processing pipeline (ISP), described in Section 5.1.1, and the drivers for the Nokia N900 imaging sensor, lens, and flash. Reusing as much infrastructure as possible greatly reduces the implementation time and effort, and makes it far easier for others to extend the API to new devices. The V4L2 kernel interface used for FCam is the one that is included in the Nokia N900 Linux kernel (2.6.28-omap1 with patches), which has some custom capabilities added by Nokia (most of these capabilities have since been reworked and will soon be parts of the standard Linux kernel). These will be described as needed below.

V4L2 has its origins as an API for video capture devices such as TV tuners or generic webcams, and for video output devices such as hardware overlays or TV outputs. It provides a C kernel API to user applications for querying and configuring such devices, and for efficiently streaming image buffers to and from them. It is the native camera control API for the Nokia N900, used by the built-in camera application for all of its functionality.

V4L2 is a streaming video API, and much like other such APIs, it decouples all configuration that does not affect the image stream format from the streaming interface.

It exposes a generic controls API, which allows video device drivers to advertise both pre-defined controls such as exposure and gain, or completely custom controls that are unique to a device. An entirely separate negotiation interface is used to select the streaming video format (resolution, pixel format, and frame rate). Finally, V4L2 has an interface for allocating shared image buffers, and when video streaming is active, for using these buffers to receive image data. There is no synchronization between the controls interface and the image buffer interface, and in general, the only timing information available to the user is the time stamps attached to each V4L2 image buffer.

While extensible and generally well-engineered, V4L2 has several limitations from the perspective of the Frankencamera architecture: There is no straightforward way to change capture parameters on a per-frame basis, no clear way to know what capture parameters were used for any given frame passed to the user application, and no way to synchronize other hardware devices to image capture. FCam must provide all these facilities to fulfill the Frankencamera architectural requirements. Figure 6.5 shows a block diagram of the N900 FCam implementation for `FCam::N900::Sensor` and the related parts of the other hardware devices. The F2 implementation is identical in its architecture, though slightly different in the specifics, especially relating to sensor timing.

Per the FCam API, the `Sensor` object accepts `Shot` objects, which define all the capture and post-processing parameters of a single `Frame` of image data, as well as all `Device Actions` during that single image capture. For each `Shot` passed in, the `Sensor` must asynchronously generate one `Frame` object which contains the `Shot` as well as the image data and metadata produced as a result of the `Shot`, and it must trigger the device `Actions` contained in the `Shot` during the image exposure. And to be useful for real applications, this must be accomplished with minimal delays throughout the system — in general the rate of output of `Frames` should match the frame rate of the image sensor itself.

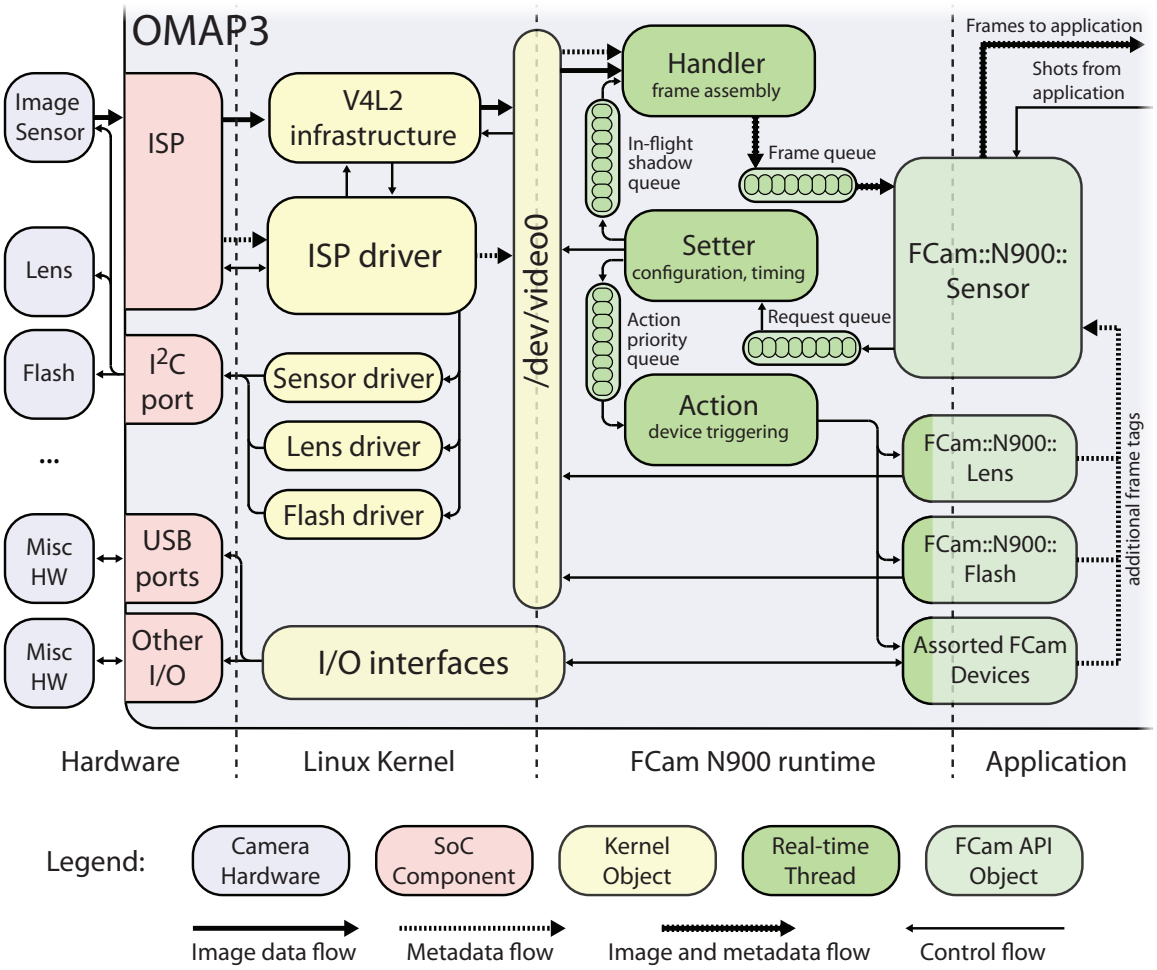


Figure 6.5: FCam::N900::Sensor implementation. The FCam implementation uses as much as of the existing Linux kernel infrastructure as possible. Everything to the right side of the kernel/user space boundary, represented by the /dev/video0 device, is newly written for FCam and is implemented in C++.

6.3.3 Hardware control challenges

Like most highly-integrated image sensors, the Toshiba ET8EK8 sensor and the Aptina MT9P031 sensor on the N900 and the F2, respectively, have a variety of configurable parameters. Both are controlled by a set of registers exposed over an I^2C interface, which define parameters such as exposure time, analog gain, image subsampling modes, and so on. In general, writes to these registers are latched so that they only take effect at the start of the next frame, ensuring that a given frame has consistent settings for all its pixels. However, since an image sensor in rolling shutter mode is simultaneously reading out frame N while exposing frame $N+1$, it is important to note that some sensor settings apply only to exposure, and some only to readout. Therefore, it is possible to update two settings such as exposure time and analog gain back-to-back so they are updated at the same frame start point, and have the exposure setting affect frame $N+1$ while the gain affects frame N. This is case for the ET8EK8 sensor in the N900, but not the case for the MT9P031 in the F2.

Therefore, for precise frame-level control of these image sensors, FCam needs to know how many frames in the future a given register write will take effect, and it needs to be able to update all the needed registers on an image sensor during a single frame interval, without the write sequence overlapping with the start of the next frame. Assuming an image sensor is running at 30 fps, for example, there is a 33 ms window for each frame during which FCam must set up all image sensor registers for that frame. This introduces a real-time constraint on the FCam implementation — sensor configuration updates must occur predictably and without interruption to ensure correctness.

Since precise timing is essential for properly configuring the image sensor and processing pipeline, and for triggering FCam `Devices` with millisecond accuracy, regular user-space threads are not sufficient. Linux uses preemptive multitasking, and with typical process time slices on the order of 10 ms, if a user-space control thread were to

be preempted when it needs to be updating sensor controls for an upcoming frame, the image sensor will use the partially updated configuration and output an incorrect result.

To avoid this problem, FCam uses a collection of real-time threads to manage the imaging pipeline. While stock Linux is not a true real-time operating system, Linux real-time priority threads are not preempted by the OS scheduler and are guaranteed to run if they are ready whenever a context-switch occurs. This is sufficient for reliable FCam operation on our current platforms. FCam's three real-time threads live inside the FCam runtime `Daemon` class, accessed through the `Sensor` class. The `Sensor` is connected to the `Daemon` through two queues, the first of which conveys new frame requests to the `Daemon`, and the second returning mostly-complete `Frames` to the `Sensor`. These are the "Request queue" and "Frame queue" shown in Figure 6.5, respectively; the `Daemon` itself consists of the three real-time thread blocks in the figure.

In general, real-time priority threads can only be launched by a privileged process, which is problematic for FCam applications that often have substantial graphical interfaces and use many libraries — the likelihood of a security flaw occurring in one such library becomes quite high. Instead, a custom kernel call was added to the V4L2 interface, which grants the real-time thread capability to the calling process. At worst, then, a security hole in an FCam application might allow for a denial of service attack by launching real-time threads that use up the entire CPU, instead of gaining full root-level access to the device.

6.3.3.1 Per-frame control

The first of the three real-time threads is the `Setter` thread, which is responsible for overall timing of the FCam runtime, and for configuring the imaging pipeline as needed. It is connected to the incoming request queue, and connects via additional queues to the other two real-time threads, as seen in Figure 6.5.

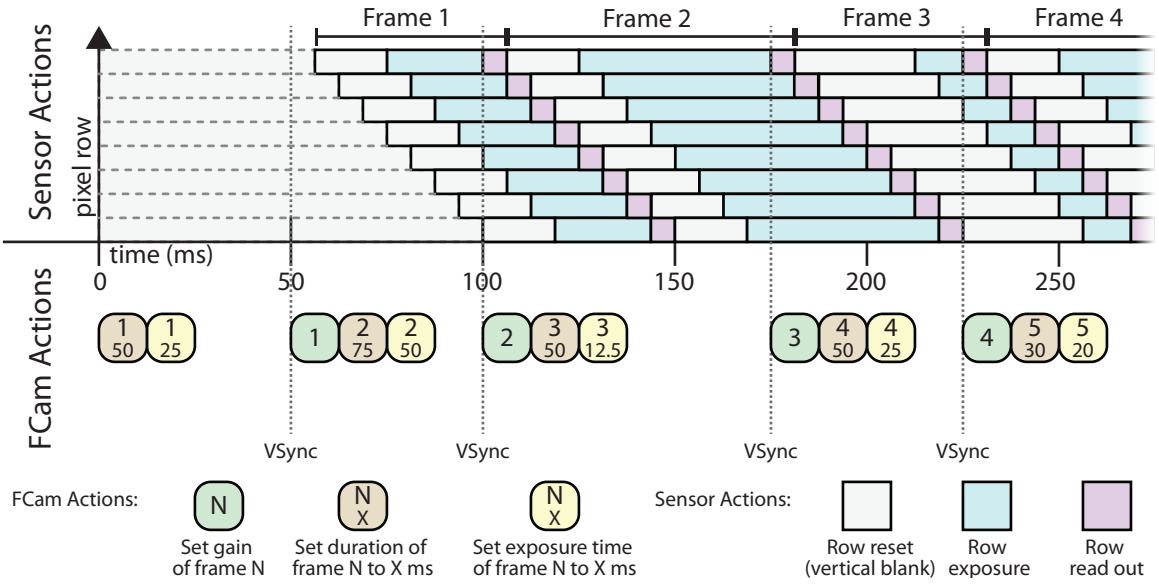


Figure 6.6: FCam sensor configuration timing. For a rolling shutter CMOS sensor, sensor configuration changes must occur in the correct time window for a given future frame. The timing here is roughly that of the Toshiba ET8EK8 sensor in the Nokia N900. Only 8 rows of pixels are shown for simplicity.

The `Setter` has precise knowledge of a given image sensor's timing behavior, which includes being aware of any extra delays that may be caused by parameter changes. For example, the Aptina MT9P031 sensor adds extra vertical blanking time whenever its region-of-interest settings are adjusted, which affects proper `Action` triggering. For the Toshiba ET8EK8 sensor, exposure controls for a frame must be set one frame before the gain controls, while for the Aptina sensor they are set at the same time. These timing models were built partly based on the relevant sensor data sheets [72, 77], and partly by empirical testing when the data sheets were vague or silent on timing details. Figure 6.6 diagrams the timeline for sensor configuration for the ET8EK8 in rolling shutter mode. VSync indicates the start of frame readout, which is also the point at which new configuration values are made active. The start of frame readout creates a vertical sync interrupt inside the OMAP3 ISP, which is then propagated to the ISP driver. The vertical sync interrupt is used as the synchronization signal for the entire FCam runtime. When

not updating sensor registers, the `Setter` thread is blocked on a kernel call, waiting for the next interrupt to come in, at which point it is guaranteed that all prior writes have taken effect.

One final complication for the `Setter` is that V4L2 has several parameters that cannot be changed without stopping the image stream: the pixel format, the image resolution, and the frame rate. The frame rate can easily be circumvented, as nothing in the V4L2 implementation actually depends on a fixed frame rate. At the sensor level, the frame rate can be adjusted on a per-frame basis by changing the amount of extra vertical blanking time added to each frame. Therefore, a custom V4L2 control that changes the frame rate is simple to write. This does break the semantics of V4L2, since the negotiated format has a fixed frame rate. But since this negotiation is not visible to the user of FCam in any case, it has no impact on the operation of FCam-based applications.

The pixel format and image resolution are not so easily circumvented. For the N900 and the F2, FCam supports streaming raw sensor data, as well as YUV-encoded image data produced by the OMAP3 ISP. Switching between the two requires reconfiguring the ISP completely. Similarly, changing image resolution requires both reconfiguring the ISP, and reallocating the kernel buffers V4L2 uses to pass data to FCam. Both of these operations can only be accomplished by shutting down image streaming, reconfiguring the V4L2 video device, and restarting streaming.

When the `Setter` receives a request that requires it to change either pixel format or resolution, it must therefore first wait until all currently in-flight `Frames` have been received by FCam. Then, it shuts down the streaming pipeline, reconfigures it, and restarts streaming. This is a lengthy operation, requiring roughly 700 ms in total, partly because the process includes completely shutting down and powering-up the image sensor and associated devices. This delay is the greatest weakness of the current FCam implementation, and is mostly due to lower-level driver issues, not the underlying hardware.

Once the `Setter` has finished configuring the image sensor for a given request, it

needs to do two more things: First, it places the request on the in-flight queue for the `Handler` thread, and it calculates the timing for any `Actions` associated with the request. An `Action` contains a trigger time relative to the start of image exposure, which isn't known until the `Setter` determines it. Once the `Setter` determines the actual firing times of the `Actions`, it places them on a priority queue for the `Action` thread.

If the rate of incoming requests is slower than the last-set frame period, the image sensor will produce captures that have not been requested. With analogy to empty pipeline stages in a CPU, these are called "bubbles" — effectively unused slots in the output of the image sensor. The `Setter` knows when such a bubble will be output, and pushes a special request to the in-flight queue, indicating that the sensor data from the bubble should be discarded and not delivered to the user application. From a power management and efficiency perspective, it would be better to simply pause the image sensor when a bubble condition would happen. However, given the structure of the V4L2 implementation on the N900, this would incur a similar 700ms overhead as a resolution switch does, which is unacceptable for what might be a single-frame bubble. The API `Sensor::stream` method also provides a way to keep the imaging pipeline busy with valid requests at all times, but it may not be appropriate to use for applications.

6.3.3.2 Device triggering

The second of the real-time threads is the `Action` thread. It is fed by the priority queue of scheduled device `Actions` from the `Setter` thread. Its task is straightforward — it simply dequeues the next `Action` (sorted in time by the priority queue) and then sleeps until a few moments before the scheduled time for the action. The remaining time is spent in a busy-wait loop, to ensure accurate timing at the cost of some system performance. The thread then calls the `Action` trigger function, and once it returns from the call, dequeues the next `Action`.

This implementation, while simple, does have several drawbacks. First, note that

`Actions` scheduled for a given frame are not passed to the `Action` thread until the previous frame has started readout. This means that `Actions` that must fire at the start of exposure cannot have latencies longer than the minimum vertical blanking interval for the sensor. This is equal to a few hundred microseconds on both platforms. Similarly, `Actions` with latencies greater than the current frame time cannot be successfully triggered. Fixing this would require FCam to be able to predict frame timing multiple frames into the future, so that a schedule of future `Actions` and exposures could be constructed and updated each time a new request is passed to the system. In general, this might require adding in bubbles (discarded frames) to allow for enough time to fire long-latency `Actions` properly.

The second drawback for the current implementation is that there are no limits on the code called by the `Action` trigger function. This means the callback could spend arbitrary amounts of time processing its event, and thus cause other `Actions` to be lost, if control is returned to the `Action` thread too late to schedule them. This is a far harder problem to solve technically, since the OMAP3 is a single-core chip (ignoring the special-purpose hardware modules). That means there's no real way to support multiple `Actions` that must be called at the same time (meaning their firing times minus firing latencies are equal). If the `Device` code is structured so that callbacks take as little time as possible, this problem can be minimized, but not eliminated.

6.3.3.3 Frame assembly

The final real-time thread in the FCam runtime is the `Handler` thread. It receives image buffers through the V4L2 API, as well as the in-flight requests from the `Setter` thread. Using the timestamps on the V4L2 buffers, and the predicted timing from the in-flight request, it matches up the requests and image buffers to correctly match up image data and metadata into a complete `Frame`. It is also responsible for querying the hardware statistics modules on the OMAP3 for frame statistics, again using timestamps to match

up the image buffers to the matching statistics.

The `Handler` deals with all the memory allocation inside the FCam runtime. The circular buffer used by the V4L2 API for passing image data across the kernel boundary is of fixed size, and typically small (by default, only 2 buffers are used for 5-megapixel streaming on the OMAP3), and the FCam API places no restriction on the lifetime of a given `Frame` object. Therefore image data must be copied out of the V4L2 buffers into some other buffer to allow the V4L2 buffers to be immediately reused.

The FCam API allows several kinds of memory targets — a `Shot` request may contain a destination `Image`, it may request new memory to be allocated for each output `Frame`, or it may request the image data to be completely discarded leaving only the metadata. In the first two cases, a memory copy is inevitable, and in the current FCam implementation, this memory copy causes most of the CPU overhead from using FCam, roughly 10% of the CPU on the N900.

Clearly, it would be best to avoid any copying of image data, but this would require substantial re-engineering of the V4L2 API as a whole, to allow destination buffers to be arbitrary user-space pointers instead of one of a small set of pre-allocated buffers. This may be difficult to accomplish in general, since passing information between the kernel and user space is fairly restricted due to security and system stability concerns.

6.3.4 Changes to the ISP drivers

The F2 and the N900 share the same kernel module for the OMAP3 ISP driver, which is a modified version of the default N900 module. The additional modifications to the base code to enable FCam functionality are summarized here.

First, for the N900, a method had to be added to disable the default N900 camera daemon, which normally controls exposure and lens focusing. This was done at the kernel level to guarantee that the daemon becomes active again when an FCam application closes down or crashes. It was also necessary to add a kernel hook to allow an FCam

application to launch real-time threads, as described in Section 6.3.3. Finally, a frame duration control was added to the N900 sensor driver.

For both the F2 and the N900, fixes were made to the ISP driver, to obtain runtime information needed for FCam to configure the ISP statistics units properly, and to fix bugs in retrieving statistics data.

Since the F2 is custom-built, it comes with all the modifications pre-installed. To use an N900 as a Frankencamera platform, the photographer has to install the modified kernel drivers, a simple process that nevertheless requires a one-time reboot of the phone. This is because, on the N900, currently loaded kernel modules cannot be removed without a reboot.

6.3.5 F2-specific API and implementation

The Aptina MT9P031 [72] on the F2 has a few features beyond those of the base FCam `Sensor` class. Specifically, it can be configured to only read out a specific rectangular region of interest for each frame, and it also supports several subsampling and downsampling modes. To support these extra features, the F2 namespace contains F2-specific `Shot` and `Frame` classes which control these settings. The F2 `Sensor` class accepts both the base and F2 versions of `Shot`, filling in defaults as needed. In addition to the `Sensor` the F2 implementation contains several additional devices that add up to a complete camera.

The F2 lens controller has a standard RS-232 serial interface, receiving text commands from the main board. The command set [73] includes basic lens actions such as changing the lens aperture and focal distance, and queries about lens parameters such as the lens ID (since the lenses are interchangeable) and the state of any switches on the lens. Since serial communication is relatively slow, the `F2::Lens` class spawns a thread to communicate with the lens controller, connected to the rest of the class by simple message queues. Since most any Canon lens can be connected to the Birger controller on the F2, the `Lens`

includes a database of lens parameters, such as a table of minimum aperture versus focal length, which can be constructed by running a simple calibration utility with the lens connected. The `Lens` class also monitors whether the lens is attached to the Birger, and sends FCam events to the application whenever a lens is connected or disconnected from the F2.

The F2 body incorporates a Phidgets [74] interface kit board, to allow assorted hardware to be easily connected to the camera. The F2 FCam implementation has support for tying the Phidgets board to FCam `Device` classes. Using this support, the F2 can trigger external flashes using Phidgets and a standard lens hot shoe. Most professional camera manufacturers have developed proprietary flash control protocols, but since none of these have been publicly reverse-engineered, currently only flash triggering is supported. Any other settings for the flash have to be configured manually on the flash unit.

Phidgets support is also used for the main shutter button on the F2 body, and the touch-sensitive slider strips bracketing the touchscreen, by connecting them to the FCam events subsystem. Other user interface elements and assorted devices can be connected as needed.

6.3.6 N900-specific API and implementation

The Toshiba ET8EK8 [77] sensor on the N900 doesn't have any additional features over the base FCam `Sensor` (except for digital zoom, which is currently not implemented in FCam), so it uses the base `Shot`, and adds no new features to its derived `Frame` class.

Like the F2, the N900 contains a lens and a flash for the camera. Unlike the F2, both of these are controlled through the V4L2 interface, with custom controls defined by Nokia. The `FCam::N900::Lens` and `FCam::N900::Flash` classes are simple wrappers around this V4L2 interface. Since the N900 lens has a variable focus distance, but not a variable focal length(zoom) or aperture, the Lens class just needs a model of the motion of the lens focusing mechanism to accurately predict the lens focus distance at any given time.

This is essential for creating accurate autofocus algorithms. Since the lens is very light, the lens motion is effectively inertialess, so linearly interpolating between commanded lens focal positions in diopters accurately predicts the lens position at any time. The Flash is even simpler, as it only needs to keep track of past flash firing actions, and to translate from the FCam API to the the straightforward V4L2 flash controls.

6.4 Cross-platform FCam implementation

In addition to the platform-specific segments of FCam, the API contains a substantial amount of interfaces and code that work on a large number of platforms. This includes a basic software imaging pipeline for converting raw sensor data to 8-bit RGB images, functions for loading and saving RAW and JPEG data, as well as testing and debugging frameworks. This allows developers to easily write post-processing applications on desktop machines that can interpret data from FCam applications.

6.4.1 Processing pipeline

The software processing pipeline converts 16-bit raw sensor data into 8-bit per channel RGB images. It consists of denoising, demosaicking, color space conversion, and gamma/contrast curve application. Since this pipeline needs to be run for every image that is converted to a JPEG, it is performance-sensitive, and tradeoffs need to be made between quality and processing time. Figure 6.7 illustrates the stages of the processing pipeline.

To minimize memory access latencies, the pipeline breaks up the input image into tiles. Each tile passes through the entire pipeline before the next is processed. This allows maximum cache re-use and minimum temporary storage for intermediate data during processing. To avoid artifacts at tile edges, each tile overlaps the next, and only the center of the computed tile is written to the final image. Beyond fitting in the L1 cache, the block

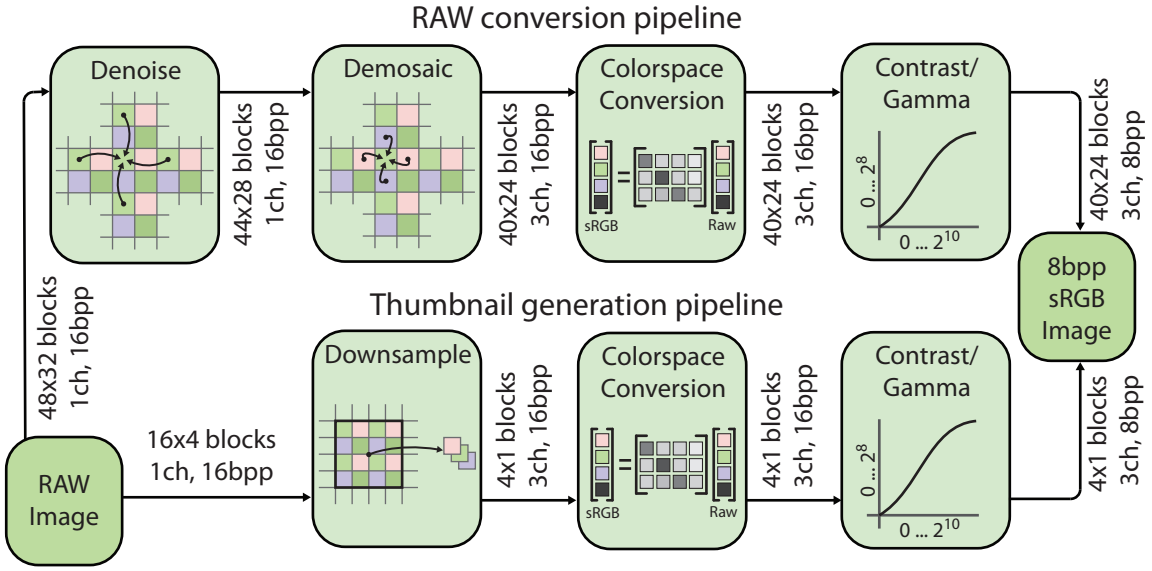


Figure 6.7: The software RAW processing pipeline. On top are the stages for generating a full-resolution RGB image from a raw sensor image. On bottom are the stages for creating a downsampled thumbnail RGB preview from a raw sensor image.

size is also constrained by the vector width of the ARM NEON unit, which is 8 for 16-bit values. The pipeline reads in 48x32-pixel tiles, and writes out the center 40x24-pixel region.

The pipeline begins with denoising, which attempts to remove hot pixels from the raw data. Hot pixels are pixels that, for various sensor-level reasons, have a much larger signal bias or offset than their neighbors. In long exposures, they appear as bright spots in the image. Especially in small image sensors and low-light conditions, they can be quite common, and need to be addressed. The denoising stage attempts to remove hot pixels by clamping every pixel to be within the range of its four immediate neighbors. While this also has a spatial low-pass filter effect, it is relatively invisible given the low-pass effect of the Bayer color filter already applied to the raw pixel data. Many more sophisticated methods exist, such as median filtering or high-dimensional Gauss transforms [87], but they run far too slowly even on desktop machines for quick user feedback.

Next, the demosaicking stage converts the Bayer-pattern image data to a full RGB

image. Each raw pixel has information for only one color channel, so the demosaicking algorithm must interpolate color information from neighboring pixels to estimate full color for each pixel. There are numerous algorithms for demosaicking [54, 88] with various performance/quality tradeoffs and goals. The FCam implementation uses a simple algorithm similar to Adaptive Color Plane Interpolation (ACPI) [89]. The general approach is to assume that the small image patch surrounding the pixel being interpolated may contain a single image edge, and that interpolation should not be performed across such an edge. Since there are twice as many green pixels as red and blue pixels, the green channel is interpolated first. Green channel gradients are used to decide whether to interpolate data horizontally or vertically for each pixel, with a low gradient being preferred. Blue and red channels are then interpolated using the complete green channel for first-order corrections.

The RGB data is still in the camera's color space, and needs to be mapped to some standard color space for application use. FCam uses sRGB [90], since it is the most common color space for computing systems. This mapping requires knowledge of the desired camera white balance, since the transformation matrix depends on the camera's white point, which depends on scene illumination. To find the matrix, FCam uses the requested color temperature to interpolate between two pre-calibrated transform matrices captured under known illumination, using the reciprocal of the color temperature. This is recommended by the DNG specification [83], as it is more perceptually uniform. The matrices are 3x4 to allow for black-level compensation as well. Finally, the image data must be converted from sensor bit depth (10 bits per pixel for our current platforms) to the typical 8 bits per pixel. This is done using a look-up table which encodes a typical gamma curve plus optional contrast and black-level adjustments.

On the F2 and the N900, a specialized version of the pipeline is used, written largely in ARM assembly using NEON SIMD [65] instructions to vectorize the image processing steps. This optimized pipeline runs in roughly 760 ms on the Nokia N900, as opposed

to 5.6 s for the generic code, a 7X improvement. The assembly version uses fixed-point calculations for color conversion, while the generic code uses standard IEEE floating-point types, but otherwise the pipelines perform identical operations. When FCam is used on a desktop computer, for example for debugging with `Dummy` or when writing an application that processes FCam DNG images, the generic code is used.

While the DSP present on the OMAP3 could be used to off-load the software processing pipeline entirely from the main CPU, the FCam implementation is currently entirely on the OMAP3's main ARM core. This is primarily due to the difficulty of loading and running code on the DSP, a process that was poorly documented and under heavy revision during FCam's initial implementation. In the future, moving the pipeline to a DSP or a GPU is likely to be worthwhile.

6.4.2 Thumbnail generation

Since many applications need to present low-resolution versions of captured images to the user as quickly as possible, a fast thumbnail-generating function is essential. Simply running the full-resolution pipeline and then downsampling the result would take nearly a second on the cameras, which is far too long for the typical use case of the photographer looking at the captured image thumbnail to evaluate it immediately.

Instead, the thumbnailing function can greatly reduce the processing requirements by downsampling the image data as the first step in the process of making a thumbnail. Instead of an adaptive demosaicking algorithm, the thumbnail function can simply average together all the values for each color channel under each output pixel's footprint. The typical case on the N900 is reducing the 5-MP (2592x1968) RAW image to a 640x480 thumbnail for display. This is accomplished fastest with a slight crop to 2560x1920, followed by a 4x reduction. Figure 6.7 illustrates the thumbnailing pipeline.

For a 4x downsampling, each output pixel covers 8 green pixels and 4 red and blue pixels. Each color channel is averaged together independently. Once this averaging

has been completed, the remaining stages can be run on 16x fewer pixels than in the full pipeline. As a result, the assembly-optimized version of the thumbnailing function runs in roughly 70 ms on the Nokia N900. By comparison, simply reading through a five-megapixel image and writing back an empty 640x480 RGB image with no actual processing takes roughly 50 ms, so the thumbnail code is mostly memory-bandwidth bound.

6.4.3 Raw image I/O

As described in Section 6.1.3.3, FCam uses the Adobe DNG format [83] as its RAW image format. While the DNG format is based on the standard TIFF image file format, the system TIFF libraries on the two platforms are not flexible enough to be used for DNG reading and writing. Adobe publishes an SDK for manipulating DNG files, but it is not designed for embedded platforms. Therefore, DNG support in FCam is written from scratch, requiring no libraries besides those for standard file I/O and memory mapping.

To save on processing time, both the RAW image data and the embedded thumbnail are stored uncompressed in the DNG file. This also allows efficient memory-mapped reading of the DNG files created by FCam, without having to copy the file data into in-memory buffers before processing or display. All metadata `Tags` added to a `Frame` are stored in the DNG private data entry, and restored on load. Similarly, all standard `Frame` fields are stored in the private data, to allow for perfect round-trip transport of information such as the white balance fields, which must be substantially transformed when written into standard DNG color data fields.

6.5 Implementation limitations

The current Frankencamera implementations are not perfect, and have several limitations that curtail their use for some applications. By far the most serious is the substantial

700 ms delay in frame capture when the application submits a `Shot` with a different resolution or pixel format from the previous `Shot`. This contributes substantially to "shutter lag" in a standard camera application — the delay between pressing the shutter button and the camera actually recording an image, a common problem in lower-end point-and-shoots. It also makes certain types of computational photography applications infeasible, such as possible variants of foveal imaging [8] in which the resolution varies on a per-frame basis. This limitation stems from two sources: First, the V4L2 Linux kernel API [53] underlying the FCam implementation was originally designed as a video capture API, in which resolution and pixel format are invariants for a capture stream. Therefore, to switch resolutions requires shutting down video streaming and reconfiguring the video device, a relatively expensive operation involving reallocating and mapping video buffers for the user application and the kernel. Second, the implementation of the ISP driver on the OMAP3 (written by Texas Instruments) is not optimized for this use case at all, and turns off the entire imaging pipeline, including the sensor, whenever streaming is halted. These problems are mostly software issues, not hardware limitations, and fixes will be included in the upcoming F3, which is discussed further in Section 7.3.1.

Second, as discussed in Section 6.3.3.2, the scheduling of device `Actions` is fairly limited. `Actions` cannot include more than a single frame duration's worth of firing latency, since the actual time of their firing is not calculated until the start of exposure for the previous frame. This could be fixed by improving the FCam sensor timing models to be able to predict sensor behavior many frames into the future, allowing `Action` triggering to happen many frames before the target frame begins exposing.

Another problem with `Action` triggering is that the `Action` callbacks run in a real-time priority thread, with no checks on what the callback method actually does. This allows for poorly written `Actions` to starve the entire system and break the FCam API timing guarantees. Similarly, two `Actions` cannot be fired at the same time (after their

firing latencies are accounted for), since both the Frankencamera platforms are single-processor devices (and in general, the devices may sit on the same external bus). Designing new `Actions` with these issues in mind avoids these problems in most cases, but good design practices cannot be enforced by the API itself.

Finally, the API implementations are not completely robust, since they are essentially attempting to enforce real-time guarantees (sensor state updates must complete by a hard deadline) in a non-real-time system. Failures can occur when the system performs a critical low-level operation with a long latency, like accessing the virtual memory swap file when available memory runs low. While the FCam API has reasonable fallback behavior in such cases, ensuring smooth operation at all times would be far preferable. This may not be possible without some type of hardware assistance, either by offloading some of the FCam runtime to a secondary processor, or by improving the image sensor interface to make state updates less fragile. For example, the sensor could have an input queue of future frame configurations, so that occasional hiccups on the OMAP3 side could be buffered. Exploring the trade-offs between hardware assistance and generality is an interesting question in general, and the current FCam implementation is a good starting point for such studies.

With the Frankencamera hardware and software defined, the next question is whether they meet the goals of the Frankencamera project, and what the next steps should be. These topics are taken up in the next chapter.

Chapter 7

Conclusion

This dissertation has presented the Frankencamera programmable camera, consisting of the abstract Frankencamera architecture specification, two hardware implementations, and the FCam API for programming Frankencameras. The abstract architecture, unlike previous camera APIs, explicitly views the camera as an imaging pipeline which can have multiple images flowing through it at once. This allows for per-frame control of all capture parameters without compromising on framerate. In addition, the architecture includes the other devices required to make a fully functional camera, providing for synchronization between them and the image sensor.

The F2 Frankencamera is a fully self-contained camera, able to capture 5-megapixel images at 10 frames per second. Based on the OMAP3 series of chips, it runs Linux and its entire software stack is open and modifiable. It is also easy to extend with new hardware devices and physical interfaces. The Nokia N900 is a compact smartphone also built around the OMAP3 chip, released in late 2009. It requires only an updated package of kernel modules to become a Frankencamera platform, with an equal level of control of the camera pipeline as the F2.

The FCam API provides a high-level interface for programming Frankencameras, focusing on easy and precise camera control. It also comes with basic utility functions

required for any camera application, such as simple control loops, file I/O, and a software image processing pipeline.

7.1 Frankencamera platform validation

The aim of the Frankencamera is to provide new, easy-to-use capabilities for researchers and students in computational photography. As such, it is clearly important to evaluate whether the Frankencamera succeeds at this goal, by having the target audience actually use both the F2 and the N900 to create novel computational photography applications. As a validation method, one can examine the experiences and results of three categories of users who have created applications for the Frankencamera. These users are, in decreasing order of expertise:

1. The main implementers of the Frankencamera who wrote FCamera, a sample full camera program for the N900.

2. Other researchers affiliated with the Frankencamera project, who wrote several demonstration applications.

3. Students in a graduate-level computational photography course, who used N900 Frankencameras for their 1-month final projects.

7.1.1 FCamera

FCamera was written to provide a full sample FCam-using program on the N900, and to create a starting point for quick development and exploration for novel applications. It provides manual control of standard camera parameters, and saves raw sensor data as DNG files. It can be used as a full replacement for the N900's built-in camera application.

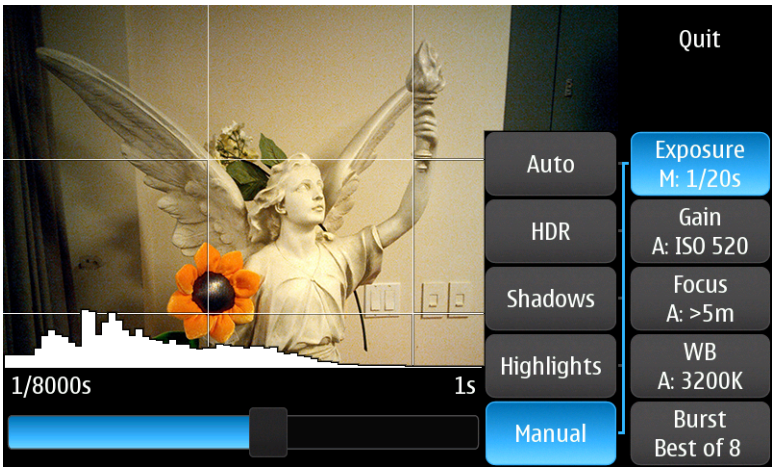


Figure 7.1: FCamera viewfinder. FCamera allows for full manual control of typical camera settings using a touchscreen interface, and also supports HDR viewfinding and capture, as well as a best-of-8 lucky imaging mode.

As recommended by Nokia's developer guidelines, FCamera is built using the Qt GUI framework [76], a cross-platform set of libraries for C++, with native support on the N900. Figure 7.1 shows a screenshot of the FCamera main window.

FCamera allows manual control of metering, white balance, focusing, and the ISO setting (gain). For metering, standard automatic metering can be used, or the algorithm can be biased toward preserving highlights or minimizing shadows. Manual exposure mode enables a large slider to make it easy to select the exposure time by hand. Additionally, an HDR mode can be enabled, which turns on an HDR viewfinder like that described in Section 7.1.2.4, and has the application capture an HDR stack of variable size when the shutter button is pressed. The number of images in the HDR burst is selected based on the dynamic range visible in the scene. FCamera does not, however, combine the images to a final HDR result. White balance, focusing, and ISO control can be switched to manual or automatic, with a large slider for manual control.

Outside of the HDR mode, FCamera supports a best-of-eight burst mode, in which the camera captures 8 images at full 10 fps, and only keeps the image measured to be the

sharpest, similar to the Lucky Imaging application described in Section 7.1.2.2.

FCamera saves DNG files by default, and optionally uses the FCam processing pipeline to save JPEG versions of the images as well. The program also incorporates a viewer for previously captured images, accessible by swiping upward on the viewfinder. From the viewer, images can be uploaded to the web using the N900's built-in sharing functionality. Additional camera settings can be found in the configuration screen accessible by swiping downward on the viewfinder.

While FCamera does not demonstrate any particularly novel computational photography features, it provides a base on which additional features can easily be added. It is also a convenient demonstration application to show off basic abilities of the FCam API, and to illustrate how to combine FCam with other libraries. It also demonstrates that the Frankencamera platform can meet the basic requirements of a digital camera: It is portable, self-powered, and capable of operating a live viewfinder and taking high-resolution raw images (Requirements 1, 5, 6, and 9 from Section 3.4).

7.1.2 Demonstration applications

During final FCam development, several applications were developed in parallel with the API, both as tests for the API, and as demonstrations of the Frankencamera platform in general. The developers were researchers in computational photography, affiliated with the Frankencamera project, but not directly involved in the core system implementation. Most applications were written in a few weeks of total development time, and show that a knowledgeable researcher can make good use of a Frankencamera quickly.

Most run on both 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 the Frankencamera 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.

7.1.2.1 Rephotography

Implementation by Jongmin Baek

This application is a reimplemention of the system of Bae et al. [38], which guides a user to the viewpoint of a historic photograph for the purpose of recapturing the same shot. The user begins by providing the historic image, and taking two shots of the modern scene from different viewpoints, which creates a baseline for stereo reconstruction. Using both SIFT [15] keypoints and the KLT tracker [91], the application then guides the user to the right position to capture the identical viewpoint. Figure 7.2 shows the user interface and the resulting photograph, overlaid with the "historic" example. In the sample implementation on the N900, a frame rate of 1.5 fps can be achieved, 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. The application demonstrates the user interface flexibility of a programmable camera, as well as the ability for substantial image processing in real time (Requirements 8 and 7).

7.1.2.2 Gyroscope-based lucky imaging

Implementation by Jennifer Dolson

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 7.3). The lucky imaging application uses an experimental Nokia 3-axis gyroscope affixed to the front of the N900 to detect hand shake. (Some smartphones, such as the Apple

Figure 7.2: Rephotography. A Frankencamera platform makes it possible to 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.

iPhone 4, have built-in gyros of similar quality.) Using 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. An external gyroscope is required because the internal accelerometer in the N900 is not sufficiently accurate for this task. This application demonstrates adding new hardware to a Frankencamera, and for synchronizing metadata from that device with `Frames` (Requirements 3 and 4).

This technique can be extended to longer exposure times where capturing a "lucky image" on its own becomes very unlikely. Indeed, Joshi et al. [22] show how to deconvolve the captured images using the motion path recorded by the inertial measurement unit as a prior.

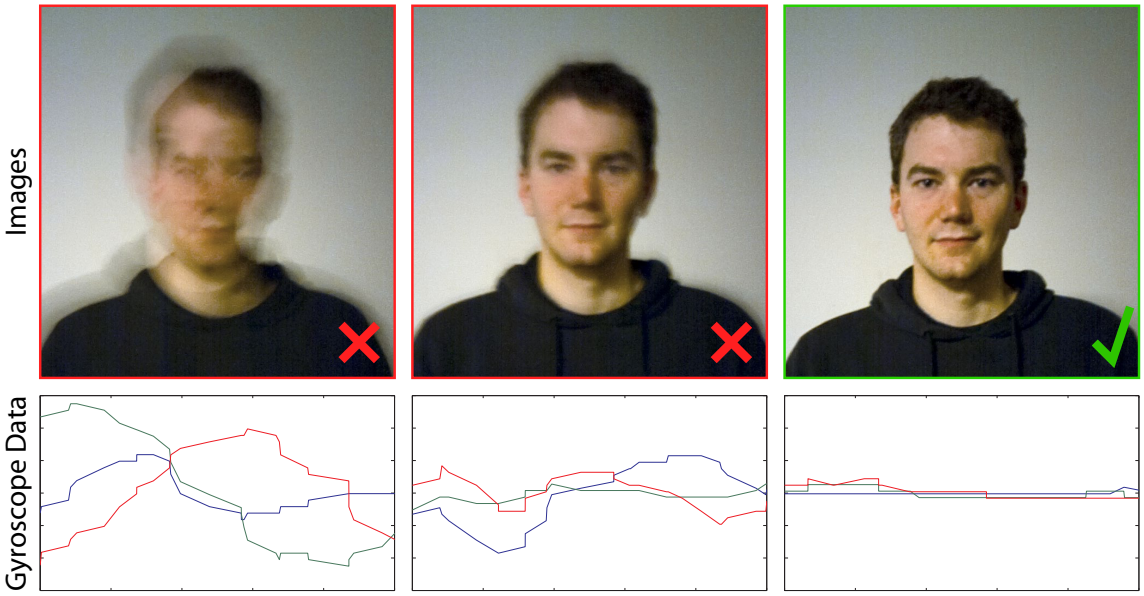


Figure 7.3: Lucky imaging. An image stream and 3-axis gyroscope data for a burst of three images with half-second exposure times. The Frankencamera API makes it easy to tag image frames with the corresponding gyroscope data. For each returned frame, the gyroscope data is analyzed to determine if the camera was moving during the exposure. Only the frames determined to have low motion are saved to storage.

7.1.2.3 Foveal imaging

Implementation by Sung Hee Park

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 either Frankencamera platform. 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 there is 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.

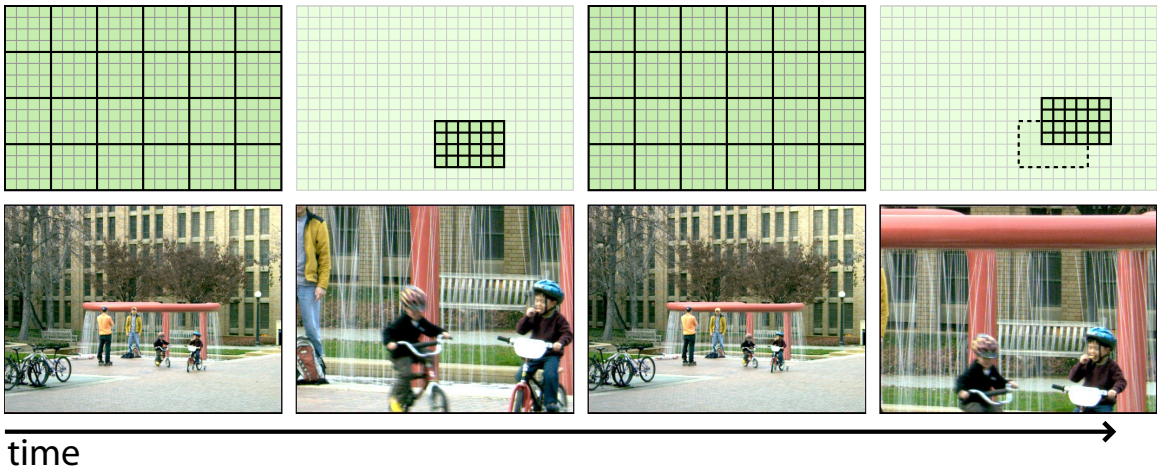


Figure 7.4: Foveal imaging records a video stream that alternates between a downsampled view of the whole scene and full-detail insets of a small region of interest. The inset can be used to record areas of high detail, track motion, or gather texture samples for synthesizing a high-resolution video. In this example, the inset is set to scan over the scene, the region of interest moving slightly between each pair of inset frames.

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. 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. This application simply acquires foveal data, impossible on other platforms, and presents results produced by moving the fovea along a prescribed path. In the future, this data could be used to synthesize full-resolution high-framerate video, similar to the work of Bhat et al. [92]. Figure 7.4 demonstrates the basic capture process. This application demonstrates the per-frame control and accurate metadata tagging provided by FCam (Requirements 2 and 4).

7.1.2.4 HDR viewfinding and capture

Implementation by Natasha Gelfand, Andrew Adams, and Sung Hee Park

HDR photography is frequently done by taking several photographs and merging them into a single image that better captures the range of intensities of the scene [1]. While modern cameras often 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 high-resolution HDR shots. The FCam API is used to implement such an application on the F2 and N900 platforms.

HDR metering and viewfinding is done by streaming a burst of two 640×480 `Shots`, whose exposure times are adjusted based on the scene content, in a manner similar to Kang et al. [93]. As the pair is streamed by the sensor, the two most recently captured images are simply averaged together 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. Note that since an output frame is displayed for every input frame, there is no loss in frame rate, although a small increase in motion blur is evident, since each display frame is now a sum of two sequential input frames. For final full-resolution captures, the Exposure Fusion algorithm [94] is used. More detail about a refined version of this application can be found in [95]. Figure 7.5 shows the captured images and results produced by the N900 implementation. This application primarily demonstrates raw sensor data access, per-frame control, and real-time processing (Requirements 1, 2, and 7).

7.1.2.5 Low-light viewfinding and capture

Implementation by Marius Tico

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

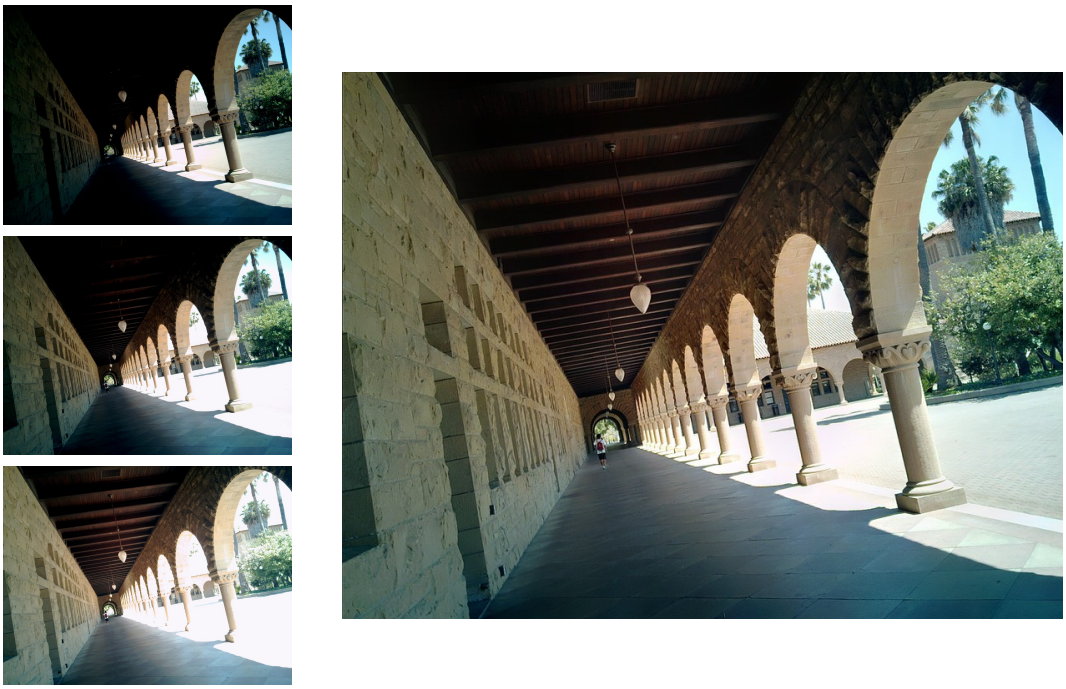


Figure 7.5: HDR imaging. The HDR capture program uses the per-frame control afforded by the FCam API to capture a set of full-resolution images with varying exposure at full sensor frame rate. Shown on the left is one such burst, of three photographs. On the right is the resulting image produced using the exposure fusion [94] algorithm, created on-camera.

exposure time, which introduces motion blur and lowers the frame rate of the viewfinder. In this application, the capabilities of the FCam API are used to implement a low-light camera mode, which augments viewfinding and image capture using the algorithms of Adams et al. [63] and Tico and Pulli [14], respectively.

The viewfinder in the 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, the application captures a pair of images: 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. The resulting frames are fused using the algorithm of

Figure 7.6: Low-light imaging. The FCam API is used to create a low-light camera mode. For viewfinding, the method of [63] is adopted, which aligns and averages viewfinder frames. For capture, the method of Tico and Pulli [14] is used, 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.

Tico and Pulli [14], which combines the best features of each image to produce a crisp, low-noise photograph, as shown in Figure 7.6. This application again demonstrates per-frame control, along with real-time and near-real-time processing (Requirements 2 and 7).

7.1.2.6 Panorama capture

Implementation by Daniel Vaquero

This application is similar to the hypothetical HDR panorama application used as a case study in Chapter 3, implemented using the FCam API on the N900. 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 the capture interface, the viewfinder alignment algorithm of Adams et al. [63] is

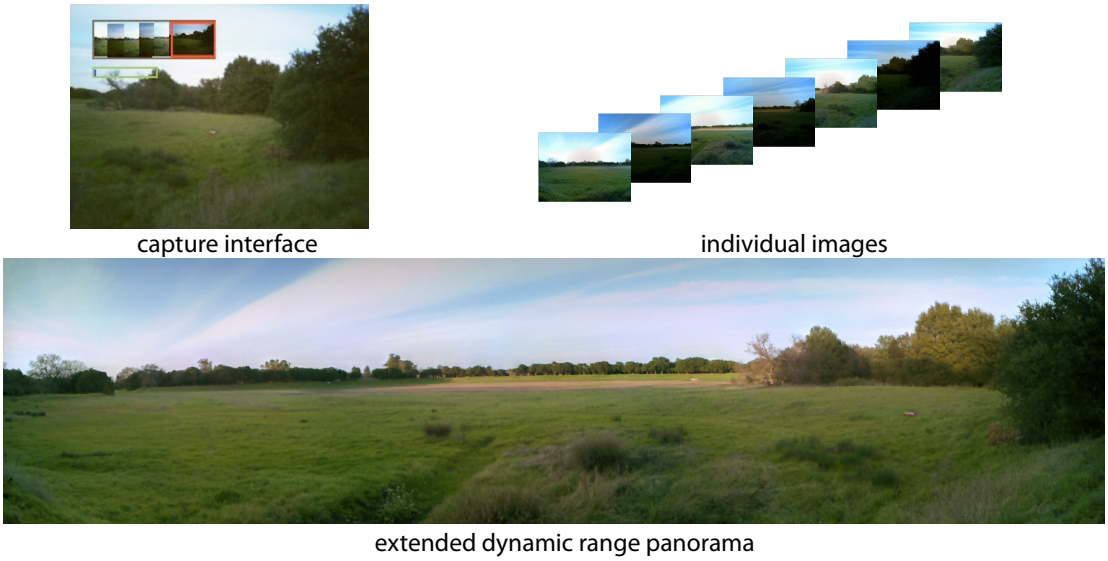


Figure 7.7: Extended dynamic range panorama capture. A Frankencamera platform allows for experimentation with novel capture interfaces and camera modes. Shown here is a semi-automated panorama capture program. The image on the upper left shows the capture interface, with a map of the captured images 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.

used to track the position of the camera, 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 7.7). In addition, varying exposure times are used to create an extended-dynamic range capture, in the manner of Wilburn et al. [16]. Once the user has covered the desired field of view, the images are stitched into a panorama on the camera, and the result can be viewed for immediate assessment. This application primarily demonstrates a novel UI, and in-camera computation (Requirements 8 and 7).

In summary, the demonstration applications plus FCamera taken together touch on every item on the computational camera requirements list (Section 3.4), and confirm

that the Frankencamera architecture meets them. They also show that the FCam API can be used by researchers to build novel applications, and quickly: all the demonstration programs were written with at most a few weeks of effort.

7.1.3 Using Frankencameras in the classroom

Shortly after the Frankencamera platforms became usable, they were used to help teach the latest version of the biennial graduate course on computational photography [96] at Stanford University. Nokia provided sufficient N900s so that each of the roughly 20 students could have their own Frankencamera for the duration of the course. The students in this course were not experts on computational photography, and had no prior familiarity with the Frankencamera.

7.1.3.1 Autofocus replacement

To familiarize the students with the FCam API and the N900, their first assignment was to write a new autofocus routine for their N900 Frankencamera. They were given a week to complete the assignment, and then their routines were tested for robustness and speed on four test scenes, which they did not have access to during development.

In Table 7.1, the top five solutions are compared to the default FCam implementation and the built-in autofocus routine on the N900 used by its default camera application. The duration for the N900 autofocus routine was determined by triggering autofocusing while filming the N900 viewfinder with a second camera, so there is a reasonable margin of error on those numbers. The student and the default FCam implementations could easily be directly instrumented to determine their duration.

The simple default FCam solution is the fastest, performing a single sweep over the image. However, it is also the least robust, failing to focus on one of the test scenes successfully. The fastest student solutions use a two-sweep algorithm, first scanning the

Implementation	Focusing speed			
	Scene 1	Scene 2	Scene 3	Scene 4
N900 default	1.5 s	1.6 s	1.5 s	1.9 s
FCam default	0.55 s	0.39 s	fail	0.31 s
Student #1	0.86 s	0.54 s	0.80 s	0.95 s
Student #2	1.27 s	1.15 s	1.19 s	1.39 s
Students #3–#5	1.5 s–1.7 s for all			

Table 7.1: Student autofocus implementations. In one week, using FCam on the N900, graduate-level students were able to write autofocus implementations that appear competitive with the Nokia N900 built-in routine. *Data courtesy of Jongmin Baek.*

whole focal range of the lens, and then sweeping again over the sharpest area found to refine the focus. On the four test scenes, the fastest student solutions also appear faster than the built-in Nokia implementation, and equally robust. Given the limited testing, it would not be appropriate to claim that the student implementations are clearly better than the built-in algorithm. However, it is fair to say that in one week, students were able to write solutions using FCam that are at least in the same ballpark as the built-in solution.

This is the first time that an assignment like this, replacing a key part of a real camera's functionality in a very short time period, could even be tried out in a classroom setting. The results were very encouraging.

7.1.3.2 Student projects

With one month for a final project, most student groups chose to use the N900 and FCam for their projects. Feedback collected after the course was generally positive. A few of the more interesting projects are discussed below.

Painted aperture for portraits. *By Edward Luong.* In this project, the N900 is used to simulate a camera with a much larger aperture than the N900 possesses. For portraiture,

Figure 7.8: Painted aperture. By moving the camera over a circular region while recording images, and then aligning the images together at a foreground object, this application synthesizes images that appear to come from cameras with a much larger aperture, decreasing depth of field for a pleasing portrait effect. On the left, a single captured frame with feature tracking points displayed. On the right, the result of aligning and merging 16 frames. *Images courtesy of Edward Luong.*

it is often desirable to blur our background details to focus the viewer's attention on the main subject. This is difficult to do with small cameras, which due to their small lens apertures have wide depths of field. Here, the camera is moved over a small circular region of space by the photographer, "painting" the desired virtual aperture. The N900 captures images throughout the painting process. After capture, the images are transferred to a desktop where they are analyzed for SIFT features [15], which are then used to align the images on the desired subject. Figure 7.8 demonstrates one result from the application. This approach is similar to the synthetic aperture imaging with camera arrays [59, 16, 97], except that the camera is handheld and uncalibrated.

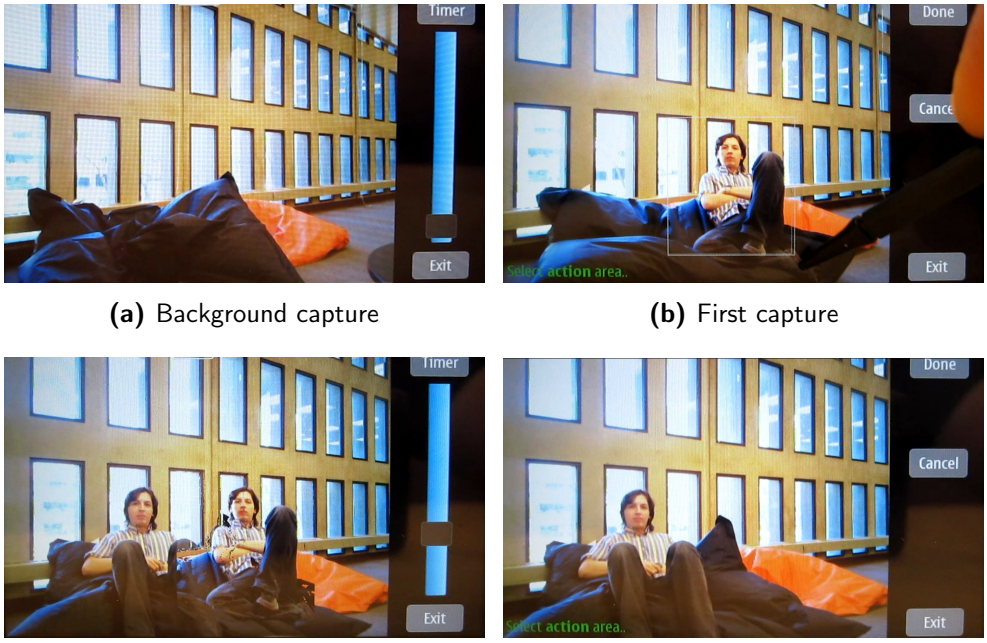
Remote flash. *By David Keeler and Michael Barrientos.* In this project, the students created a Bluetooth-based communication library connected to FCam which allowed them to synchronize `Actions` across a pair of N900s. With this framework, they implemented a

(a) On-camera flash. **(b)** Flash from helper N900.

Figure 7.9: Remote flash application. In this project, the students synchronized two N900 so that the other N900 fired its flash when the first took a picture. *Images courtesy of David Keeler and Michael Barrientos, with post-processing for color balance and contrast.*

remote flash application, in which a helper N900 fires its flash when the primary N900 is capturing an image. This improves the quality of the resulting images, since straight-on flash often causes redeye problems and very stark lighting on the subject. Figure 7.9 shows a sample result from the application.

Photomontage assistance. *By Juan Manuel Tamayo and Nikhil Gupta.* This application primarily enhances the camera user interface to simplify capturing source photographs intended for a photomontage. In a photomontage, multiple images of nearly the same scene are combined together to create images where, say, a single person appears multiple times. Capturing the source images for such an image can be difficult, since it is hard to align objects or people across multiple captures without visual aids. In this application, a pilot image of the base scene is used to allow the changing sections of the scene to persist on the viewfinder after capture, to make it easy to line up the next capture properly.



(a) Background capture **(b)** First capture

(c) Capture overlaid on live viewfinder **(d)** Next capture

Figure 7.10: Photomontage user interface. The photomontage application uses an initial base image of a scene to allow scene changes captured in later photographs to persist in the viewfinder, assisting in the alignment of the next capture. *Images courtesy of Juan Manuel Tamayo and Nikhil Gupta.*

Figure 7.10 shows images from the application viewfinder during the capture process.

7.1.4 Validation conclusions

The results from Frankencamera deployment have been encouraging. Both researchers and students were able to create applications that could not have been written on existing platforms, and could do so in a matter of weeks. In addition, feedback collected from the students after the class was generally positive, and nearly all the students used FCam for their final projects. By comparison, when the course was run two years prior using the N95 smartphone with its native camera API, only 2 students tried using it for their final projects, after an initial required assignment. Some of the criticisms raised, such as the

lack of white balancing in the initial FCam implementation, and the lack of a full sample application such as FCamera, have since been addressed. In general, the Frankencamera architecture and the FCam API design have met their goal: To make life easier for anyone wanting to program their camera.

7.2 Public release

In late July of 2010, the FCam API was released publicly for the Nokia N900 [9]. The release included the development API and reference documentation, several simple example applications, and a kernel driver package for the Nokia N900 to enable Frankencamera functionality. The driver modifications coexist with the built-in N900 camera application, so no functionality is lost when the FCam drivers are installed.

The FCamera sample program described in Section 7.1.1 was also uploaded to the Maemo software testing repository along with the kernel driver packages. After community feedback, FCamera was promoted to the main public extras repository, visible in the N900's built-in application manager for all N900 owners. In the first three months after release FCamera was downloaded over 25,000 times, indicating a substantial interest in improved camera applications even from regular users. The team at Nokia Research Center Palo Alto also released two applications that use the FCam framework, which are improved versions of the HDR capture application and the low-light application described in Sections 7.1.2.4 and 7.1.2.5, respectively.

7.3 Future work

The contributions detailed in this dissertation are just a first step in opening up the core of the camera for researchers and photographers. Several obvious steps present themselves as possibilities for continued research in this area.

7.3.1 Improved camera hardware

The F2 is a research prototype with several limitations, as discussed previously. Replacing it with a new, better engineered platform is important in order to provide researchers with a high-image-quality platform. The design of the F3 Frankencamera has already begun — it will be based on the Cypress LUPA-4000 image sensor [98], a 4-megapixel 24x24 mm CMOS image sensor, with advanced non-destructive readout and high-dynamic-range features.

In the future, the processing power limits of the OMAP3 platform will become more and more limiting — already, popular algorithms such as SIFT feature extraction [15] struggle to run on the chip at reasonable rates. Texas Instruments has released the follow-up design, the OMAP4 family [61], with substantially increased processing power. Additionally, the OMAP4 ISP is far more powerful and programmable, containing its own ARM processor for configuration and control of the imaging pipeline. It might well be possible to move the entire timing-critical section of the FCam implementation into that processor, improving the robustness and reducing the processing overhead of FCam.

Ideally, the FCam API would become available on a wide variety of platforms, including other smartphones as well as regular digital cameras. This would be best accomplished by convincing camera and cell phone manufacturers that providing the end-user more powerful control over the camera's operation is a worthwhile goal. So far, several cell phone vendors have expressed interest in the Frankencamera, and the applications it enables. This interest will hopefully translate to improved commercial devices soon.

7.3.2 Video processing

This dissertation has focused primarily on efforts to produce compelling still images, not video. However, the FCam API can easily be used to output a constant-frame-rate video stream, enabling exploration of video applications. Unfortunately, the memory limits

and storage speeds on the F2 and the N900 preclude easily storing uncompressed videos in real time. While the OMAP3's DSP can be used to encode video in real time, there is no support in FCam's utility libraries for this yet. In the future, adding support for video encoding will expand the range of applications for the Frankencamera even further.

7.3.3 Easy image processing

While FCam makes it straightforward to control a Frankencamera with precision, a traditional part of computational photography remains cumbersome on mobile platforms: image processing. Because the processing resources on chips like the OMAP3 are spread over several heterogeneous units (the CPU, its SIMD extensions, the DSP, the GPU, and so on), it quickly becomes very difficult for researchers to access all the available processing power on a programmable camera. As an example, to make the software imaging pipeline in FCam run at acceptable speed, it had to be implemented with the ARM NEON SIMD extensions, in a mix of C intrinsics and ARM assembly code. The resulting code, while fast, is hard to read and harder to modify. If one wished to move the pipeline to the DSP or the GPU, it would have to be written again from scratch.

Some type of abstraction is needed to allow researchers to easily write high-performance code, which can then be run on any of the processing units in a typical mobile device. A domain-specific language is one possibility, which can describe an algorithm and its inherent parallelism at a high level, allowing automated compilation for different targets. If ISPs or ISP stages become programmable in the future, the same language could then be used to program them, allowing for even larger improvements in programmer efficiency.

7.4 Keeping the black box open

To maximize the chances that future camera platforms become programmable and sufficiently open for research, it is clear that camera and camera phone manufacturers must be convinced that such access is valuable to provide — then, if one manufacturer provides an FCam-like API across their devices, the others may be forced to follow.

A compelling case can be made to the cell phone manufacturers that opening up their camera APIs would not just benefit researchers in computational photography (who are unlikely to be a large enough group to matter to such large companies), but in fact would have wide appeal. If the camera subsystems of cell phones become easily programmable, many hobbyist and professional programmers may begin writing powerful new photography applications for these devices, alongside the researchers. One day it might be possible to download a new camera application that improves photography for niche cases that the manufacturers themselves could never justify supporting, but that in the aggregate convert the humble camera phone into a powerful computational photography device. Faced with such a challenge, the traditional digital camera manufacturers may find themselves having to open up their devices as well to stay competitive, making high-quality dSLRs and point-and-shoots programmable for all.

While one might argue about how likely such a scenario really is in the large, it is at least true on the Nokia N900, for which anyone can now write an FCam-based application, and make it available to anyone with an N900 to download and run. And that is a good start.

Bibliography

[1] E. Reinhard, G. Ward, S. Pattanaik, and P. Debevec, *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. The Morgan Kaufmann Series in Computer Graphics, San Francisco, CA, USA: Morgan Kaufman Publishers, 2006.

[2] S. Mann and R. W. Picard, "On Being 'undigital' With Digital Cameras: Extending Dynamic Range By Combining Differently Exposed Pictures," in *Proceedings of IS&T's 48th Annual Conference*, pp. 422–428, 1995.

[3] P. E. Debevec and J. Malik, "Recovering high dynamic range radiance maps from photographs," in *Proceedings of SIGGRAPH 1997*, pp. 369–378, ACM Press/ACM SIGGRAPH, 1997.

[4] S. W. Hasinoff, F. Durand, and W. T. Freeman, "Noise-Optimal Capture for High Dynamic Range Photography," in *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 553–560, 2010.

[5] Camera & Imaging Products Association, *Digital Cameras: Statistical Data*, 2010. http://www.cipa.jp/english/data/dizital.

[6] Camera & Imaging Products Association, *Interchangeable Lens: Statistical Data*, 2010. http://www.cipa.jp/english/data/silver.

[7] M. Levoy, "Experimental Platforms for Computational Photography," *IEEE Computer Graphics and Applications*, vol. 30, no. 5, pp. 81–87, 2010.

[8] A. Adams, E.-V. Talvala, S. H. Park, D. E. Jacobs, B. Ajdin, N. Gelfand, J. Dolson, D. Vaquero, J. Baek, M. Tico, H. P. A. Lensch, W. Matusik, K. Pulli, M. Horowitz, and M. Levoy, "The Frankencamera: An experimental platform for computational photography," *ACM Transactions on Graphics (SIGGRAPH 2010)*, vol. 29, no. 4, pp. 29:1–29:12, 2010.

[9] E.-V. Talvala and A. Adams, "The FCam API." http://fcam.garage.maemo.org, 2010.

[10] N. Goldberg, *Camera Technology: The Dark Side of the Lens*. San Diego, CA, USA: Academic Press, Inc., 1992.

[11] J. Dolson, J. Baek, C. Plagemann, and S. Thrun, "Upsampling Range Data in Dynamic Environments," in *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1141–1148, 2010.

[12] M. Levoy, Z. Zhang, and I. McDowall, "Recording and controlling the 4D light field in a microscope using microlens arrays," *Journal of Microscopy*, vol. 235, no. 2, pp. 144–162, 2009.

[13] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama, "Digital photography with flash and no-flash image pairs," *ACM Transactions on Graphics (SIGGRAPH 2004)*, vol. 23, no. 3, pp. 664–672, 2004.

[14] M. Tico and K. Pulli, "Image enhancement method via blur and noisy image fusion," in *Proceedings of the 16th IEEE International Conference on Image Processing (ICIP)*, pp. 1521–1524, 2009.

[15] M. Brown and D. G. Lowe, "Automatic Panoramic Image Stitching using Invariant Features," *International Journal of Computer Vision*, vol. 74, no. 1, pp. 59–73, 2007.

[16] B. Wilburn, N. Joshi, V. Vaish, E.-V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz, and M. Levoy, "High performance imaging using large camera arrays," *ACM Transactions on Graphics (SIGGRAPH 2005)*, vol. 24, no. 3, pp. 765–776, 2005.

[17] G. Garg, E.-V. Talvala, M. Levoy, and H. P. Lensch, "Symmetric Photography: Exploiting Data-Sparseness in Reflectance Fields," in *Proceedings of Eurographics Symposium on Rendering*, 2006.

[18] P. Peers, D. K. Mahajan, B. Lamond, A. Ghosh, W. Matusik, R. Ramamoorthi, and P. Debevec, "Compressive light transport sensing," *ACM Transactions on Graphics*, vol. 28, no. 1, pp. 3:1–3:18, 2009.

[19] P. Sen and S. Darabi, "Compressive Dual Photography," *Computer Graphics Forum (Eurographics 2009)*, vol. 28, no. 2, pp. 609–618, 2009.

[20] A. Román and H. P. Lensch, "Automatic Multiperspective Images," in *Proceedings of Eurographics Symposium on Rendering*, 2006.

[21] N. Snavely, S. M. Seitz, and R. Szeliski, "Photo tourism: Exploring photo collections in 3D," *ACM Transactions on Graphics (SIGGRAPH 2006)*, vol. 25, no. 3, pp. 835–846, 2006.

[22] N. Joshi, S. B. Kang, C. L. Zitnick, and R. Szeliski, "Image Deblurring using Inertial Measurement Sensors," *ACM Transactions on Graphics (SIGGRAPH 2010)*, vol. 29, no. 4, pp. 30:1–30:9, 2010.

[23] R. Ng, "Fourier slice photography," *ACM Transactions on Graphics (SIGGRAPH 2005)*, vol. 24, no. 3, pp. 735–744, 2005.

[24] E. H. Adelson and J. Y. A. Wang, "Single Lens Stereo with a Plenoptic Camera," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 99–106, 1992.

[25] A. Veeraraghavan, R. Raskar, A. Agrawal, A. Mohan, and J. Tumblin, "Dappled Photography: Mask Enhanced Cameras for Heterodyned Light Fields and Coded Aperture Refocusing," *ACM Transactions on Graphics (SIGGRAPH 2007)*, vol. 26, no. 3, pp. 69:1–69:12, 2007.

[26] A. Levin, R. Fergus, F. Durand, and W. T. Freeman, "Image and depth from a conventional camera with a coded aperture," *ACM Transactions on Graphics (SIGGRAPH 2007)*, vol. 26, no. 3, pp. 70:1–70:9, 2007.

[27] A. Levin, P. Sand, T. S. Cho, F. Durand, and W. T. Freeman, "Motion-Invariant Photography," *ACM Transactions on Graphics (SIGGRAPH 2008)*, vol. 27, no. 3, pp. 71:1–71:9, 2008.

[28] A. Levin, S. W. Hasinoff, P. Green, F. Durand, and W. T. Freeman, "4D Frequency Analysis of Computational Cameras for Depth of Field Extension," *ACM Transactions on Graphics (SIGGRAPH 2009)*, vol. 28, no. 3, pp. 97:1–97:14, 2009.

[29] E. R. Dowski, Jr. and W. T. Cathey, "Extended depth of field through wave-front coding," *Applied Optics*, vol. 34, no. 11, pp. 1859–1866, 1995.

[30] F. C. A. Groen, I. T. Young, and G. Ligthart, "A comparison of different focus functions for use in autofocus algorithms," *Cytometry*, vol. 6, no. 2, pp. 81–91, 1985.

[31] J.-S. Lee, Y.-Y. Jung, B.-S. Kim, and S.-J. Ko, "An Advanced Video Camera System with Robust AF, AE, and AWB Control," *IEEE Transactions on Consumer Electronics*, vol. 47, no. 3, pp. 694–699, 2001.

[32] M. Gamadia and N. Kehtarnavaz, "Enhanced low-light auto-focus system model in digital still and cell-phone cameras," in *Proceedings of the 16th IEEE International Conference on Image Processing (ICIP)*, pp. 2677–2680, 2009.

[33] S. J. Lee, S. H. Jin, and J. W. Jeon, "FPGA based auto focus system using touch screen," in *Proceedings of the IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pp. 350–353, 2008.

[34] PENTAX Imaging Company, Golden, CO, USA, *SLR Digital Camera K-7 Operating Manual*, 2009.

[35] Canon Inc., Tokyo, Japan, *Canon EOS 7D Instruction Manual*, 2009.

[36] E.-V. Talvala, A. Adams, M. Horowitz, and M. Levoy, "Veiling glare in high dynamic range imaging," *ACM Transactions on Graphics (SIGGRAPH 2007)*, vol. 26, no. 3, pp. 37:1–37:9, 2007.

[37] Canon Inc., Tokyo, Japan, *Canon EOS Digital SDK EDSDK2.9 API Programming Reference*, August 2010.

[38] S. Bae, A. Agarwala, and F. Durand, "Computational rephotography," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 3, pp. 24:1–24:15, 2010.

[39] The CHDK project, "The Canon Hack Development Kit." http://chdk.wikia.com, 2010.

[40] The Magic Lantern Project, "Magic Lantern Firmware." http://magiclantern.wikia.com/wiki/, 2010.

[41] T. Hudson, "State of the Latern: 1 year anniversary." E-mail, http://groups.google.com/group/ml-devel/browse_thread/thread/648f25d8d543e58, June 2010.

[42] Automated Imaging Association, *CameraLink: Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers*, 2000.

[43] QImaging Inc., Surrey, BC, Canada, *Retiga-4000R Datasheet*, 2008.

[44] Point Grey Research Inc., Richmond, BC, Canada, *Point Grey Grasshopper datasheet*, 2010.

[45] Basler AG, Ahrensburg, Germany, *Basler A400k User's Manual*, 2009.

[46] A. Filippov, "Reconfigurable high resolution network camera," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 276–277, 2003.

[47] Elphel, Inc., West Valley City, UT, USA, 2010. http://www3.elphel.com/.

[48] A. Rowe, A. Goode, D. Goel, and I. Nourbakhsh, "CMUcam3: An Open Programmable Embedded Vision Sensor," Technical Report RI-TR-07-13, Carnegie Mellon Robotics Institute, May 2007. http://www.cmucam.org/.

[49] "Class android.hardware.camera, Android 2.2 r1 Developer Reference." http://developer.android.com/reference/android/hardware/Camera.html, 2010.

[50] Apple Inc., Cupertino, CA, USA, *AV Foundation Programming Guide, Audio & Video*, 2010.

[51] "Symbianˆ3 Application Developer Library." http://developer.symbian.org/main/documentation/reference/s3/sdk/index.html, 2010.

[52] Nokia Corp., Espoo, Finland, *Nokia N900 User Guide, Issue 1.3*, 2010.

[53] "The Linux V4L-DVB Framework." http://linuxtv.org/wiki/index.php/Main_Page, 2010.

[54] B. Gunturk, J. Glotzbach, Y. Altunbasak, R. Schafer, and R. Mersereau, "Demosaicking: Color Filter Array Interpolation," *IEEE Signal Processing Magazine*, vol. 22, no. 1, pp. 44–54, 2005.

[55] S. E. Butner and M. Ghodoussi, "Transforming a surgical robot for human telesurgery," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 5, pp. 818–824, 2003.

[56] B. A. Wandell, *Foundations of Vision*. Sunderland, MA, USA: Sinauer Associates, 1995.

[57] Nikon Corporation, Tokyo, Japan, *Nikon Coolpix S570 User's Manual*, 2009.

[58] Sony Corporation, Tokyo, Japan, *Cyber-shot Digital Still Camera DSX-HX1 Instruction Manual*, 2009.

[59] M. Levoy, B. Chen, V. Vaish, M. Horowitz, I. McDowall, and M. Bolas, "Synthetic aperture confocal imaging," *ACM Transactions on Graphics (SIGGRAPH 2004)*, vol. 23, no. 3, pp. 825–834, 2004.

[60] Texas Instruments Incorporated, Dallas, TX, USA, *OMAP35x Applications Processor, Texas Instruments OMAP™ Family of Products, Technical Reference Manual*, September 2008.

[61] Texas Instruments Incorporated, Dallas, TX, USA, *OMAP4430 Multimedia Device, Silicon Revision 2.x, Texas Instruments OMAP™ Family of Products, Version L, Technical Reference Manual*, October 2010.

[62] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry*, vol. 30, no. 6, pp. 864–872, 2009.

[63] A. Adams, N. Gelfand, and K. Pulli, "Viewfinder Alignment," *Computer Graphics Forum (Eurographics 2008)*, vol. 27, no. 2, pp. 597–606, 2008.

[64] ARM Limited, *Cortex™-A8 Technical Reference Manual, Revision: r3p0*, 2008. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344c/.

[65] ARM Holdings, Cambridge, UK, *ARM Compiler toolchain Assembler Reference, Version 4.1*, 2010. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489a/.

[66] Texas Instruments Incorporated, Dallas, TX, USA, *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, July 2010.

[67] The Khronos Group Inc., Beaverton, OR, USA, *OpenGL® ES Common Profile Specification, Version 2.0.24*, April 2009.

[68] Texas Instruments Incorporated, Dallas, TX, USA, *OMAP3530/25/15/03 Applications Processor, Silicon Revisions 3.1.2, 3.1, 3.0, 2.1, and 2.0, Silicon Errata*, October 2010.

[69] Mistral Solutions Pvt. Ltd., Bangalore, India, *Hardware User Guide, OMAP35x Evaluation Module*, 2009.

[70] beagleboard.org, Richardson, TX, USA, *BeagleBoard System Reference Manual Rev C4*, 2009.

[71] Eastman Kodak Company, Rochester, NY, USA, *Kodak KAI-11002 Image Sensor 4008(H) X 2672(V) Interline CCD Image Sensor, Device Performance Specification, Revision 3.0 MTD/PS-0938*, 2010.

[72] "1/2.5-Inch 5Mp CMOS Digital Image Sensor: MT9P031," Data Sheet Rev. E 7/10 EN, Aptina Imaging Corporation, San Jose, CA, USA, 2010.

[73] Birger Engineering, Inc., Boston, MA, USA, *Canon EF-232 Library User Manual, Rev 1.3*, 2009.

[74] S. Greenberg and C. Fitchett, "Phidgets: Easy development of physical interfaces through physical widgets," in *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST'01)*, pp. 209–218, 2001.

[75] "The Ångström Linux Distribution." http://www.angstrom-distribution.org/, 2010.

[76] "Qt Online Reference Documentation." http://doc.qt.nokia.com/, 2010.

[77] Toshiba Corporation, Tokyo, Japan, *APPLICATION NOTE for SMIA95-5M Auto Focus camera module, Module model name: TCM8341MD, Sensor model name: ET8EK8-AS, Ver. 2.30b*, May 2009.

[78] "Maemo.org Development." http://maemo.org/development/, 2010.

[79] "Nokia Qt SDK." http://www.forum.nokia.com/Develop/Qt/Tools, 2010.

[80] "Technical Report on C++ Library Extensions," ISO/IEC Technical Report 19768, International Organization for Standardization, Geneva, Switzerland, 2007.

[81] V. Chikane and C.-S. Fuh, "Automatic white balance for digital still cameras," *Journal of Information Science and Engineering*, vol. 22, pp. 497–509, 2006.

[82] The Joint Photographic Experts Group, *ISO/IEC IS 10918-1 | ITU-T Recommendation T.81 (JPEG)*, 1992.

[83] Adobe Systems Incorporated, San Jose, CA, USA, *Digital Negative (DNG) Specification, Version 1.3.0.0*, June 2009. http://www.adobe.com/products/dng/pdfs/dng_spec.pdf.

[84] D. Coffin, "dcraw - Decoding raw digital photos in Linux." http://www.cybercom.net/~dcoffin/dcraw/, 2009.

[85] A. Adams, "ImageStack: A command line stack calculator for images." http://code.google.com/p/imagestack, 2010.

[86] The IEEE and The Open Group, *POSIX.1-2008 / IEEE Std 1003.1-2008 / The Open Group Technical Standard Base Specifications Issue 7*, 2008.

[87] A. Adams, J. Baek, and M. A. Davis, "Fast High-Dimensional Filtering Using the Permutohedral Lattice," *Computer Graphics Forum (Eurographics 2010)*, vol. 29, no. 2, pp. 753–762, 2010.

[88] B. Ajdin, M. B. Hullin, C. Fuchs, H.-P. Seidel, and H. P. A. Lensch, "Demosaicing by Smoothing along 1D Features," in *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–8, 2008.

[89] J. E. Adams, Jr. and J. F. Hamilton, Jr., "Adaptive color plane interpolation in single sensor color electronic camera," United States Patent 5652621, 1997.

[90] M. Anderson, R. Motta, S. Chandrasekar, and M. Stokes, "Proposal for a Standard Default Color Space for the Internet: sRGB," in *Proceedings of the Fourth Color Imaging Conference: Color Science, Systems, and Applications*, pp. 238–245, Society for Imaging Science and Technology, 1996. http://www.color.org/sRGB.xalter.

[91] C. Tomasi and T. Kanade, "Shape and Motion from Image Streams: a Factorization Method Part 3: Detection and Tracking of Point Features," Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.

[92] P. Bhat, C. L. Zitnick, N. Snavely, A. Agarwala, M. Agrawala, M. Cohen, B. Curless, and S. B. Kang, "Using Photographs to Enhance Videos of a Static Scene," in *Proceedings of Eurographics Symposium on Rendering*, 2007.

[93] S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High dynamic range video," *ACM Transactions on Graphics (SIGGRAPH 2003)*, vol. 22, no. 3, pp. 319–325, 2003.

[94] T. Mertens, J. Kautz, and F. Van Reeth, "Exposure Fusion," in *Proceedings of Pacific Graphics*, 2007.

[95] N. Gelfand, A. Adams, S. H. Park, and K. Pulli, "Multi-exposure imaging on mobile devices," in *Proceedings of the ACM International Conference on Multimedia*, pp. 823–826, 2010.

[96] M. Levoy, F. Durand, and J. Baek, "CS448A Computational Photography - Winter 2010." http://graphics.stanford.edu/courses/cs448a-10, 2010.

[97] V. Vaish, G. Garg, E.-V. Talvala, E. Antunez, B. Wilburn, M. Horowitz, and M. Levoy, "Synthetic Aperture Focusing using a Shear-Warp Factorization of the Viewing Transform," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops (CVPR Workshops)*, June 2005.

[98] "LUPA 4000: 4 Megapixel CMOS Image Sensor," Data Sheet 38-05712 Rev. *F, Cypress Semiconductor Corporation, San Jose, CA, USA, 2010.