Mobile Computational Photography with FCam

Kari Pulli and Alejandro Troccoli

Abstract In this chapter we cover the FCam (short for Frankencamera) architecture and API for computational cameras. We begin with the motivation, which is flexible programming of cameras, especially of camera phones and tablets. We cover the API and several example programs that run on the NVIDIA Tegra 3 prototype tablet and the Nokia N900 and N9 Linux-based phones. We discuss the implementation and porting of FCam to different platforms. We also describe how FCam has been used at many universities to teach computational photography.

1 Frankencamera: An experimental platform for Computational Photography

The Frankencamera platform creates an architecture for computational photography. The system was originally created in a joint research project between Nokia Research Center and Stanford University, in teams headed by Kari Pulli and Marc Levoy, respectively. It was described at SIGGRAPH 2010 by Adams et al. [1], and an open source implementation of the FCam API was also released in summer 2010. In this chapter we describe the motivation for this architecture, its key components, existing implementations, and some applications enabled by FCam.

Kari Pulli
NVIDIA Research, 2701 San Tomas Expressway, Santa Clara, CA 95050, e-mail: karip@nvidia.com

Alejandro Troccoli
NVIDIA Research, 2701 San Tomas Expressway, Santa Clara, CA 95050, e-mail: atroccoli@nvidia.com

1.1 Computational Photography

The term computational photography is today understood as a set of imaging techniques that enhance or extend the capabilities of digital photography. Often the output is an ordinary photograph, but one that could not have been taken by a traditional camera. Many of the methods try to overcome the limitations of normal cameras, often by taking several images with varying image parameters, and then combining the images, computing to extract more information out of the images, and synthesizing an image that is in some way better than any of the input images [5, 6]. Some approaches modify the camera itself, especially the optical path, including the lens system and aperture through which the light travels before hitting a sensor [4].

Even though much of the computational photography predates modern mobile devices such as camera phones, a smartphone is in some sense an ideal platform for computational photography. A smartphone is a full computer in a convenient and compact package, with a large touch display, and at least one digital camera. The small form factor precludes some of the plays with novel optics, and makes it challenging to manufacture a high-quality camera system with sufficiently large lens and sensor that can obtain good images in reasonable lighting conditions. Precisely because of this challenge, the opportunity to collect more data from several input images, and combine them to produce better ones, makes computational photography an important part of mobile visual computing. However, mobile computational photography comes with the added requirement of being able to deal with hand-held cameras that are likely to move either during the image exposure (causing blur) or also between capturing of the images in a burst (causing ghosting as the same objects have moved).

1.2 FCam architecture and API

Traditional camera APIs have usually been optimized for the common simple use cases such as taking an individual still image or capturing a video clip. If the setting of all camera parameters is automated, and the precise parameters and intervening image processing steps are not documented, it is difficult to properly combine the images to create better ones. This lack of control and transparency motivated the design of an experimental platform for computational photography. The FCam API has so far been implemented at Stanford for a large camera that accepts Canon SLR lenses (Frankencamera V2, or F2, see the inset in Figure 8), for two commercial Nokia smartphones (N900 and N9) running Linux, and on an NVIDIA Tegra 3 development tablet running Android.

Figure 1 illustrates the abstract Frankencamera architecture. A key innovation of this architecture with respect to the previous camera architectures lies in how the camera state is represented. Most traditional camera APIs combine the image sensor and image processor into a single conceptual camera object that has a global state: the current set of parameter values. However, a real camera sensor is a pipeline:

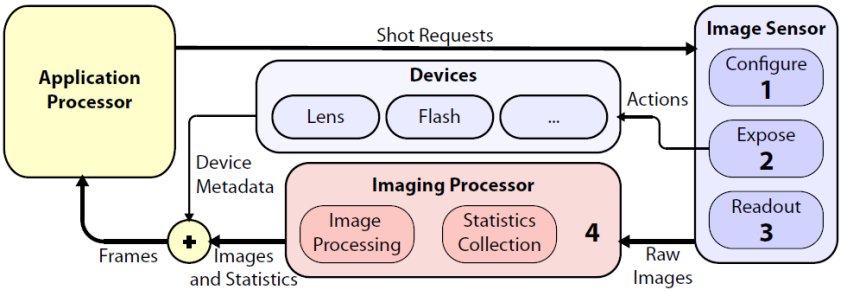

Fig. 1 The Frankencamera architecture: The application processor generates **`Shot`** requests that are sent to the **`Sensor`**. When the image sensor is exposed, any registered actions related to the **`Shot`** are executed on the **`Lens`**, **`Flash`**, or other devices. The image processor accepts the image data, computes statistics, and performs any requested image processing tasks. Finally, the image, the statistics plus tags from the devices are combined into a **`Frame`**. From Eino-Ville Talvala's dissertation [11].

while an image is being exposed, the capture parameters for the next image are being configured, and the previous image is being read out. Also the image processor is a pipeline: it first preprocesses the image in RAW or Bayer format, then demosaicks the image into an RGB and YUV image, and finally tonemaps the image so it can be displayed. If you now change the "state" of this camera system, the changed parameters may affect non-deterministically different images. In a streaming video application this is not so important, as the control algorithms change the values gradually and adaptively: the knowledge of exactly which frame is affected is often not crucial. For still imaging it is important that exactly the correct parameters affect deterministically only a single image. To guarantee determinism, the whole system may have to be reinitialized and the image streaming restarted, which creates latency especially if several images need to be captured. FCam takes a different approach to state handling by associating the state not with the camera, but with an individual image request called **`Shot`**. Now the state travels through the pipeline and allows the system to proceed at a higher speed even when different images have different parameters and state.

This innovation allows the following key capabilities to be applied at higher speeds:

- Burst control (per-frame parameter control for a collection of images),
- Synchronization of flash, lenses, etc.,
- Specialized algorithms for auto focus, auto exposure, and auto white balance.

In the following sections we describe in more detail how these features can be used via the FCam API, as well as some of the applications they enable.

2 Capture control

A salient feature of the FCam API is that the camera does not have any global state. Instead, the **`Sensor`** object receives capture requests that contain the state for the request, and turns these requests into image data, metadata, and actions, as shown in Figure 2. In this section we discuss the **`Shot`**, i.e. the request, and the **`Frame`**, the data container.

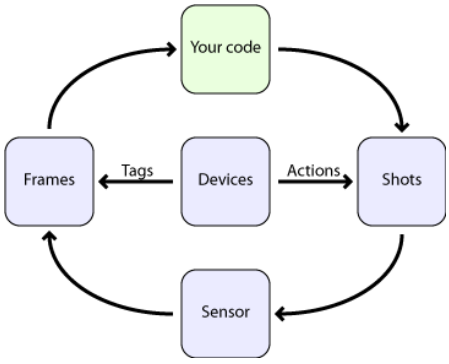

Fig. 2 The typical request generation and processing cycle. User code configures Shots, which control the Sensor, which again fills Frames with image data and metadata, which again are delivered back to user code. Devices can associate Actions with Shots, the Actions are executed at a time specified with respect to the image exposure. The Devices can also tag frames with additional metadata.

2.1 Shots and Frames

A capture request takes the form of a **`Shot`** class instance. A **`Shot`** defines the desired image sensor parameters such as exposure time, frame time, and analog gain. In addition, a **`Shot`** also has properties to configure the Image Signal Processor (ISP) to process the image with a given color temperature for white-balance and to configure the generation of statistics such as a sharpness map and image histogram. Finally, a **`Shot`** also takes an **`Image`** object that defines the image resolution and format. Figure 3 illustrates a piece of FCam API code that performs a capture request.

The call into the **`Sensor`** to capture a **`Shot`** is non-blocking, so we can keep doing more work while the image is being captured, and even issue additional requests. For each **`Shot`** that we pass down we can expect a corresponding **`Frame`** to be returned. That **`Frame`** object contains the image data and additional information that describes both the *requested* and the *actual* parameters that were used for

```
FCam::Tegra::Sensor sensor;
FCam::Tegra::Shot   shot;
FCam::Tegra::Frame  frame;

shot.gain         = 1.0f;  // Unit gain
shot.exposure     = 25000; // Exposure time in microseconds
shot.whiteBalance = 6500;  // Color temperature
// Image size and format
shot.image = FCam::Image(2592, 1944, FCam::YUV240p);

// Enable the histogram generation and sharpness computation
shot.histogram.enabled = true;
shot.sharpness.enabled = true;

// Send the request to the Sensor
sensor.capture(shot);

// Wait for the Frame
frame = sensor.getFrame();
```

Fig. 3 A typical capture request.

the capture, plus the statistics that we have requested. The actual parameters may differ from the requested ones when the **`Shot`** includes a request that cannot be completely satisfied as specified, such as too short or long an exposure time. As we will see in Section 3, a **`Frame`** can also contain additional metadata about devices such as the state of the flash and the position of the lens. To retrieve a **`Frame`** we call `Sensor::getFrame()`, which is a blocking call that will only return once the **`Frame`** is ready.

2.2 Image bursts

Many computational photography applications need to capture several images taken in quick succession, and often with slightly different parameters. We call such a set of images a burst, and represent it in the API as a vector of **`Shot`** instances. The FCam API runtime will do its best to capture the burst with the minimum latency.

The prototypical application of image bursts is high-dynamic-range (HDR) imaging. A scene we are interested in may contain a much larger dynamic range than we can capture with a single image. That is, if we set the exposure parameters so that details in bright areas can be seen, the dark areas remain too dark to resolve any details, and vice versa. By combining information from images taken with different exposure times we can generate a new image that preserves details both in the dark and bright regions.

In Figure 4 we show sample code to generate a burst of varying exposure times with the FCam API using a vector of **`Shot`** instances, and in Figure 5 we show the

```
FCam::Tegra::Sensor sensor;

std::vector<FCam::Tegra::Shot>  burst(3);
std::vector<FCam::Tegra::Frame> frames(3);

// Prepare shot with color temperature 6500K,
// unity gain and 10,000 microseconds exposure
burst[0].gain         = 1.0f;
burst[0].whiteBalance = 6500;
burst[0].exposure     = 10000;

// Copy the shot parameters
burst[1] = burst[2] = burst[0];

// Change the exposure time for the other shots
burst[1].exposure = 20000;
burst[2].exposure =  5000;

// Reserve one storage image for each frame
burst[0].image = FCam::Image(2592, 1944, FCam::YUV420p);
burst[1].image = FCam::Image(2592, 1944, FCam::YUV420p);
burst[2].image = FCam::Image(2592, 1944, FCam::YUV420p);

// Send the request to the Sensor
sensor.capture(burst);

// Read back the Frames as they are produced
frame[0] = sensor.getFrame();
frame[1] = sensor.getFrame();
frame[2] = sensor.getFrame();
```

Fig. 4 Example code that produces a burst capture of 3 consecutive frames while varying the exposure time.

results of a varying exposure burst which we combined into a single image using exposure fusion [8].

3 External devices and synchronization

A camera subsystem consists of the imaging sensor plus other devices, such as the flash and the lens focusing motor. It is important that the image sensor and the devices are synchronized properly to achieve the highest throughput and correct results. The FCam API provides a mechanism to set the behavior of these devices per **`Shot`**, as we will describe below.

Fig. 5 A burst of five image taken with different exposures (left and bottom) are fused into a single image that shows details both in the bright and dark areas better than in any of the input images.

3.1 Devices and Actions

For each external device that needs to be synchronized with the exposure, there is a corresponding proxy class in the FCam implementation. We represent such devices under a class called **`Device`**, and its behavior can be either programmed to take effect immediately, as the exposure of a given **`Shot`** starts, or at some later time. The behavior is controlled using another class called **`Action`**. A basic **`Action`** has a time field that defines the execution time relative to the beginning of the **`Shot`** exposure. When an **`Action`** is added to a **`Shot`**, the FCam runtime will take all the necessary steps so it is ready to execute it, synchronized with the **`Shot`** exposure. This synchronization is possible when the underlying camera subsystem has predictable latencies.

3.1.1 Flash

As a first example, we will take a look at the **`Flash`** device and its **`FireAction`**. The **`Flash`** class represents the camera flash and has methods to query its properties, such as maximum and minimum supported duration and brightness. It also

has a method called `fire()` that sends the commands to the hardware device to turn the flash on. The latency between the call to `fire()` and the actual flash being fired can be queried with the method `fireLatency()`.

In addition, to synchronize the flash with a given **`Shot`**, the **`Flash`** class provides a predefined **`FireAction`**, which specifies the starting time plus the duration and brightness for the flash. By setting the brightness and the duration of the **`FireAction`** we can trigger the flash. Figure 6 puts these concepts together and shows an example of flash/no-flash photography, in which two different requests are sent to the **`Sensor`**: a **`Shot`** with a **`FireFlash`** action followed by a shot without flash.

```
FCam::Tegra::Sensor sensor;
FCam::Tegra::Flash  flash;

sensor.attach(&flash);

std::vector<FCam::Tegra::Shot>  shots(2);
std::vector<FCam::Tegra::Frame> frames(2);

// Prepare the shots
shots[0].gain         = 1.0f;
shots[0].whiteBalance = 6500;
shots[0].exposure     = 30000;
shots[1] = shots[0];

// Add flash action to fire the flash for the duration of
// the entire frame and with maximum brightness
FCam::Flash::FireAction fire(&flash);

fire.duration   = shots[0].frameTime;
fire.time       = 0;
fire.brightness = flash.maxBrightness();

shots[0].addAction(fire);

// Reserve one storage image for each frame
shots[0].image = FCam::Image(2592, 1944, FCam::YUV420p);
shots[1].image = FCam::Image(2592, 1944, FCam::YUV420p);

// Send the request to the Sensor
sensor.capture(burst);

// Read back the frames as they are produced
frame[0] = sensor.getFrame();
frame[1] = sensor.getFrame();
```

Fig. 6 Example code that produces a flash/no-flash image pair.

3.1.2 Lens

Another device that is readily available in FCam is the **Lens**. The **Lens** device has query methods to retrieve the lens focal range, aperture range, and zoom range; and state setting methods to set the lens to a particular focus position, zoom focal length, or aperture. Of course, not all lenses will support all settings and the query functions return a single-valued range for those properties that are fixed. For functions that affect the focus of the lens, the unit that is used is called a diopter; lens position and lens speed are given in diopters and diopters/sec, respectively. Diopters can be obtained from $100\text{cm}/f$, where f is the focusing distance, with zero corresponding to infinity, and 20 corresponding to a focusing distance of 5cm. This unit is particularly suitable for working with lens positions because lens movement is linear in diopters, and depth of field is a fixed number in diopters regardless of the depth you are focused at.

The **Lens** device provides three different kinds of **Action** classes: **FocusAction**, **ApertureAction**, and **ZoomAction** to control focus, aperture, and focal length, respectively. In the Tegra implementation of FCam, there is also a **FocusStepping** action that allows to cover a focal range in a given number of steps and is useful for covering the focal range during auto focus.

Figure 7 contains a code snippet that shows how to move the lens to the nearest focus position and capture a shot.

```
FCam::Tegra::Sensor sensor;
FCam::Tegra::Lens   lens;

sensor.attach(&lens);

FCam::Tegra::Shot  shot;
FCam::Tegra::Frame frame;

// Setup the shot parameters
shotgain         = 1.0f;
shotwhiteBalance = 6500;
shot.exposure    = 30000;
shot.image = FCam::Image(2592, 1944, FCam::YUV420p);

// Move the lens to the closest focus position
lens.setFocus(lens.nearFocus(), lens.maxFocusSpeed());
while(lens.focusChanging()){;}

// Send the request to the Sensor
sensor.capture(shot);

// Get the frame
frame = sensor.getFrame();
```

Fig. 7 Capture a shot at near focus.

3.2 Tags

When using the FCam API there is no need to keep track of the state for each **`Device`**. Instead, each **`Frame`** that is returned by the **`Sensor`** is tagged with the parameter‘s of all devices that had been attached to the **`Sensor`**. Each **`Device`** that is in use has to be attached to the **`Sensor`** by calling `Sensor::attach()` before triggering the first capture. This allows the **`Sensor`** to know which devices to notify that a Frame capture has been completed.

Tags are parameter values that are added to a Frame instance. Each device class has an inner class to retrieve its corresponding tags from a **`Frame`**. Following our **`Flash`** and **`Lens`** examples, the **`Flash`** provides **`Flash::Tags`** and the **`Lens`** provides **`Lens::Tags`**. The **`Flash`** tags indicate the flash firing time relative to the start of the exposure, its duration, its brightness, and its peak time. If the flash was not fired, the tags will show a brightness of zero. Similarly, the **`Lens`** provides tags that indicate the initial and final focus positions, the focus speed, and the average focus setting for a **`Frame`**. If the lens did not move during the exposure, the three values for initial, final, and average focus position will all be the same. There are also tags for aperture and zoom settings.

It is important to stress that accurate tagging and **`Frame`** parameters makes a big difference in computational photography applications. Knowing the states of the camera during the exposure allows plugging this information into our algorithms or making a decision about the usefulness of the **`Frame`** we have just captured. For example, one might decide to discard a **`Frame`** if the lens moved during the exposure of the shot.

3.3 Application: Second-curtain flash synchronization

The richness of the API and its ability to synchronize the exposure with external devices can be exemplified with second-curtain flash synchronization. Using the F2 Frankencamera and two Canon flash units, one doing low-intensity strobing, while the other emits a second-curtain high-intensity flash at the end of the exposure, it is possible to produce the effect shown in Figure 8. A long exposure captures the path of the cards as they fly into the air, and the final bright flash freezes the cards to their final positions in the image.

4 Automating capture parameter setting

Early photographers had full control of every stage of photography, and they had to make explicit selections of all the variables affecting the creation of a photo. They had to estimate the amount of light in the scene and how that should be taken into account in selection of the lenses, aperture setting, or exposure time. Some of the

Fig. 8 Second-curtain flash: one flash strobes to illuminate the path of the flying card, while the other one freezes the motion at the end of the exposure. Image by David Jacobs.

exposure problems could be still treated during the interactive film development and printing stages. Modern cameras make photography much easier as they have automated most of these decisions. Before the actual image is taken, the camera measures and tries some of the parameters. This process typically consists at least of these three tasks: auto exposure, auto focus, and auto white balance, also known collectively as 3A. Video is controlled continuously: the camera analyses the previous frames, and based on the analysis the exposure, focus, or white balance values are slowly and continuously modified for the following frames.

Traditional camera control APIs completely automate these tasks and do not allow the user to modify them. Sometimes the user can override the precise camera control parameter values, but it is not possible to provide different metrics or algorithms for determining those values automatically. The default 3A produces values that in most cases provide a good image, but is optimized for the average situation, not for the current application. For example, in a security application the camera should make sure that the faces of the people remain recognizable, or the register plates of the cars can be deciphered, but it does not matter if the sky is completely saturated. FCam, on the other hand, allows you to implement your own parameter setting algorithms that are suitable for your needs.

In this section we discuss the default 3A algorithms provided by FCam, together with some advanced algorithms.

4.1 Auto exposure

The auto exposure algorithm determines how much light should be collected to create an image so that it is not too dark and does not saturate. There are several parameters that affect the exposure, the most obvious one being the duration of the exposure. Other parameters include the amount of gain applied in the conversion of analog sensor signal to digital, and the size of the aperture in the lens system. Since the size of the aperture affects also other parameters such as depth of field, it is usually kept fixed by the auto exposure routines. In dark conditions it is better to increase the exposure time to collect more light, but on the flip side this allows both the camera and objects in the scene to move, which causes blur. By increasing the analog gain, also known as the ISO value, one can shorten the exposure time and still get a sufficient large signal, but by amplifying the signal, the noise is amplified as well, and the likelihood of saturating the light representation increases, so the limits for modifying gain are fairly narrow. The gain and exposure time are usually multiplied together, and this product is called exposure.

FCam provides a sample auto exposure function that allows the user to set two numbers for an exposure target. The first number is a percentage P, and the second number is target luminance value Y. For example, values $P = 0.995, Y = 0.9$ mean that the system tries to find an exposure value so that 99.5% of the pixels have a value that is at most 0.9 (1.0 means the pixel is saturated). These values mean metering for highlights, so that the details in bright areas remain visible. Setting $P = 0.1, Y = 0.1$ can be interpreted so that at most 10% of the pixels should have a value 0.1 or less, metering the image so that details in the shadows remain visible.

What makes choosing the perfect exposure value difficult is the inconvenient fact that the dynamic range of the sensor is quite narrow, so it is often impossible to take a single image in which details both in the dark and bright areas remain visible, as discussed earlier with HDR imaging. A typical heuristic for capturing an HDR burst is to meter for one normal image, and then choose a fixed number of images that are taken with increasing and decreasing exposure settings. For example, if the auto exposure routine gives an exposure duration of 20ms, the bracketing heuristic could choose durations of 1.25, 5, 20, 80, and 320 milliseconds for the five shots in the burst. A better heuristic would find first the shortest exposure so that no pixel is (or only a few pixels are) saturated, and then increase the exposure times as long as there are pixels that remain very dark. However, neither heuristic adapts well to the actual distribution of the light in the scene.

Gallo et al. [7] developed a metering method for HDR imaging that attempts to take the smallest number of images while still accurately capturing the scene data. An advantage of taking only a few shots is that the capture takes shorter amount of time, leaving the scene objects less time to move around. Also, a burst containing

fewer images can be processed faster, and there is a smaller chance to create spurious artifacts, especially when objects are moving.

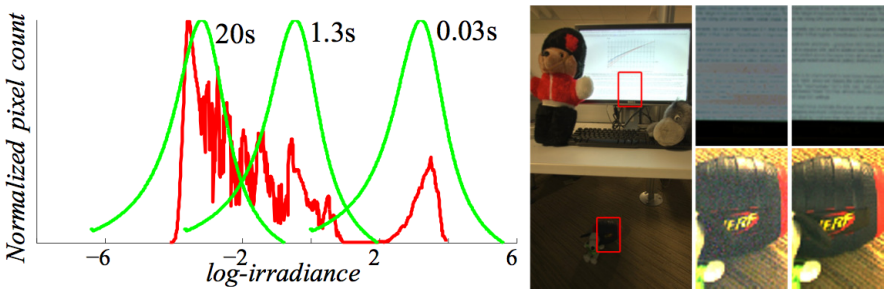

Fig. 9 HDR metering: luminance histogram (red) with three chosen exposures (green); office scene; noisy HDR with 5 bracketed images; better HDR with 3 better-metered images. Image by Orazio Gallo.

Figure 9 illustrates the method of Gallo et al. [7]. The red curve on the left shows the light histogram for the office scene shown in the middle. The areas marked by red rectangles in the office scene are shown enlarged on the right. Gallo's method selects only three images with exposure times of 0.03, 1.3, and 20 seconds, producing the detail HDR images on the first column on the right, while an auto-bracketing method with five images produces more noisy result, which is illustrated in the second column from the right. The trick is that Gallo first estimates the whole luminance histogram, and then places the images such that they capture in their most sensitive region the parts of the histogram that are strongly represented in the scene. The method was implemented on an NVIDIA Tegra 3 developer board using FCam on Android.

4.2 Auto focus

Another important parameter to choose is the focus distance. Sometimes it is desirable that as much of the image as possible remains sharp, but at other times only the object at the center of attention needs to be sharp, while the unimportant background should be somewhat blurred. Most automated focus routines try to maximize the sharpness either over the whole image or over its center.

FCam provides a sample auto focus implementation. It uses a simple sharpness measure evaluated either over the whole image or a user-specified window. The sharpness measure is a sum of absolute differences of the intensity of neighboring pixels. The idea is that, if the image is blurred, the neighboring pixels have quite similar intensity values, while textured surfaces that are in focus have pixels with higher variance in their intensity values. The sample implementation simply sweeps

the lens and gathers sharpness statistics, estimating the single lens position that maximizes the sharpness within the evaluation window.

Fig. 10 Three images (left) were taken, focused in objects in foreground (top), middle ground (middle), and background (bottom), and combined into a single image that is sharp everywhere (right). Image by Daniel Vaquero.

Vaquero et al. [13] implemented a method that computes an all-in-focus image. If some of the scene objects are very close and others are far, it may be impossible to take a single image in which everything is in focus. However, if one captures several images focused at different depths, one can then afterwards combine them by selecting pixels separately from different images based on their sharpness estimate, as illustrated in Figure 10.

The benefits of auto focusing for focal stacks are similar to metering for HDR stacks. Even though one could simply take an image focused at every depth, it is better to only take those images that actually bring new information, yielding a faster capture time, faster processing time, and fewer chances of creating processing artifacts.

4.3 Auto white balance

The human visual system quickly adapts to the color of ambient illumination and mostly discounts it, allowing good color perception under varying lighting conditions. This is much more difficult to do for a camera, and may result in images with a strong color tint, and appear both unnatural and very different from how a humans perceive the same situation. One reason for this is that the camera has a much smaller field of view than people and thus cannot as accurately estimate the color of

the ambient illumination. Another reason is that the mechanisms of color constancy in human perception are still not completely understood.

The FCam sample auto white balance implementation uses a simple heuristic called the gray world assumption. The idea is that many scenes have many objects that do not have a color that differs from some shade of gray, including white, black, and anything in between. FCam further simplifies the assumption so that it attempts to balance the amount of blue and red light, as they correspond psycho-physically to cold and warm colors, while green does not have as strong perceptual effect. The sensor is pre-calibrated with two color correction matrices, one to correct a scene with blueish tint and another to correct a scene with reddish tint. The relative amounts of blue and red light in the captured image determine how these two color correction matrices are interpolated before they are applied to correct the image colors.

4.4 Building your own camera application

It is easy to write your own custom camera application using FCam. A basic camera application streams frames continuously, and for each captured **`Frame`** the user is provided with statistics that allow 3A to be performed, either by using the sample implementation or the user's own, more sophisticated heuristics. The streaming **`Shot`** parameters are updated and the **`Frame`** displayed to the user. These functionalities could be implemented using the simple example code shown in Figure 11.

While the FCam API can take care of the camera control aspect of the camera application implementation, the display and UI are system-dependent. On the N900 platform the Qt framework is used for the UI and display. On the Tegra 3 platform the Android framework is used instead. An Android UI is built as a Java component that sets up the Android views. The FCam API is a native API, and therefore requires using the Java Native Interface (JNI) for the communication between the Java Virtual Machine and the native library that contains the FCam code. The UI generates events that are passed to the native camera implementation that runs on FCam. We have built a sample application called FCameraPro depicted in Figure 12. This application can be modified without much effort to try out new algorithms and techniques.

5 FCam platforms

The Frankencamera architecture and FCam API began as a joint research project between Nokia Research Center and Stanford University Graphics Lab. Nokia was at the time working on a family of smartphones that ran Linux (the version was earlier called Maemo, later renamed to Meego) and that used the TI OMAP 3 processor. Mistral had also made OMAP 3 boards available even for hobbyists, and the project

```
FCam::Tegra::Sensor    sensor;
FCam::Tegra::Shot      shot;
FCam::Tegra::Frame     frame;
FCam::Tegra::Lens      lens;
FCam::Tegra::AutoFocus autoFocus(&lens);

// Viewfinder resolution
shot.image = FCam::Image(1280, 720, FCam::YUV240p);

// Enable the histogram generation and sharpness computation
shot.histogram.enabled = true;
shot.sharpness.enabled = true;

// Attach the lens device to sensor
sensor.attach(&lens);

// Send a streaming request to the sensor
sensor.stream(shot);

while(1) {
    // Wait for the frame
    frame = sensor.getFrame();

    // Display the frame
    display(frame);

    // Do 3A
    FCam::autoExpose(&shot, frame);
    FCam::autoWhiteBalance(&shot, frame);
    FCam::autoFocus(frame, &shot);

    // run the shot with the updated parameters
    sensor.stream(shot);
}
```

Fig. 11 A basic camera implementation.

chose OMAP 3 and Linux as the common HW and SW platform. This led in parallel to two related implementations: the Nokia N900 used the standard hardware that the phone shipped with and allowed much more flexible use of that hardware than the camera stack that came with the phone, and the Stanford Frankencamera V2 (F2) used the Mistral OMAP 3 board together with a Birger lens controller that accepts Canon EOS lenses. The N900 allowed a relatively cheap mass-marketed FCam solution, while F2 provided an extensible research platform that allowed experimentation with different optics and hardware choices.

The FCam in N900 was not "officially" supported by the Nokia product program, it was a "community effort" maintained by the Nokia Research Center. However, the follow-up product N9 now provides official support for the FCam API. Currently,

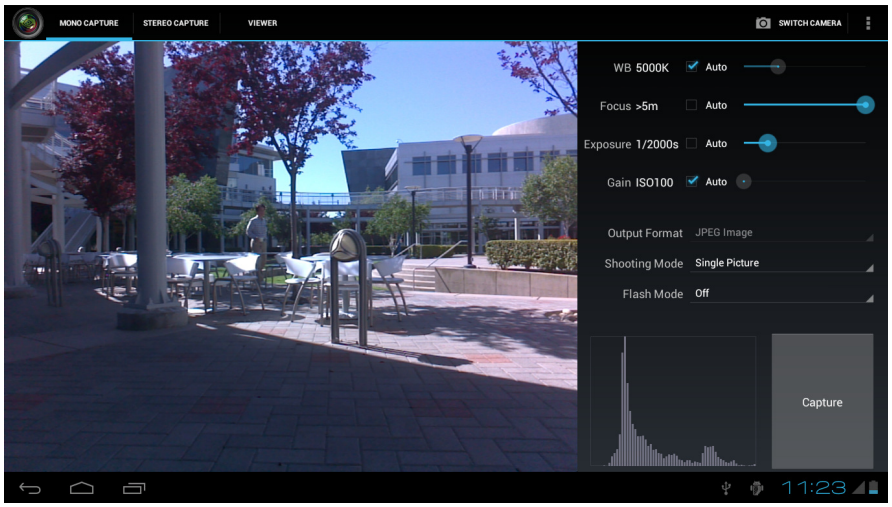

Fig. 12 FCameraPro: A custom camera application built using FCam.

most active FCam development happens on NVIDIA's Tegra-3-powered development tablets running Android.

5.1 The FCam runtime

At the core of any of the FCam implementations is the FCam runtime. The runtime is made of a set of components that take FCam API calls, configure the camera subsystem for execution, and return captured frames. In Figure 13 we show the block-diagram of the FCam implementation running on the NVIDIA Tegra 3 prototype board. The runtime is made of the FCam API objects and runs a number of threads. The first is a **`Setter`** thread that manages the incoming requests queue, programs the hardware, and computes the absolute time at which actions should be executed. Secondly, an **`Action`** thread manages the action queue; it wakes up for each scheduled action and launches its execution. It is important that the work an **`Action`** launches on execution is bounded, otherwise the thread could miss the execution deadline for the following **`Action`**. If necessary, an **`Action`** could spawn a new thread to achieve completion. Finally, a **`Handler`** thread receives callbacks from the camera driver with image data and metadata, assembles these data into a **`Frame`** instance, and delivers it to the **`Sensor`** output queue. On the left side of Figure 13 is the camera hardware, the NVIDIA Tegra 3 SoC (system-on-chip), the Linux kernel drivers and the NVIDIA camera driver. The NVIDIA camera driver takes parameter requests and assembles commands to configure the ISP or calls the corresponding kernel device driver, according to the request.

Having given the basic components of an FCam implementation, we now enumerate the steps necessary to convert an application request into image data:

1. The application makes a capture request into the **`Sensor`** passing a **`Shot`**.
2. The **`Sensor`** takes the **`Shot`** and places it in the request queue that it shares with the **`Setter`** thread.
3. At the next indication that the camera subsystem is ready to be configured, the **`Setter`** thread takes the first element of the request queue. For each **`Action`** in the **`Shot`** it computes its execution time and schedules it in the action priority queue. It also sends commands to the NVIDIA camera driver to configure the image sensor and ISP with the requested parameters.
4. The **`Action`** thread wakes up and executes any **`Action`** that is synchronized with the current **`Shot`**. Each **`Action`** will trigger a command into the NVIDIA camera driver.
5. The NVIDIA camera driver abstracts the underlying camera hardware. It receives commands and programs the corresponding kernel device drivers.
6. When the image data and metadata are ready, the NVIDIA camera driver delivers them to the **`Handler`**.
7. The **`Handler`** assembles a **`Frame`** and puts it into the frame output queue.
8. When the frame output queue receives a new **`Frame`** the **`Sensor`** delivers it to the application.

To further expand on the pipeline aspects of the FCam runtime, we now turn our attention to the timeline of events that are needed for proper configuration of the image sensor. An image sensor might require state changes to be precisely timed. For example, an image sensor could have a dual set of registers, and the system writes into one of them the parameters that will become active at the frame reset. Or it could be the case that the change to a particular register is applied immediately.

The image sensor in the Nokia N900 is a Toshiba ET8EK8 rolling shutter CMOS. The sensor requires that the exposure time and frame duration be programmed one frame ahead. The sensor emits a vertical synchronization (VSync) interrupt that is used to synchronize the FCam runtime. At the VSync interrupt the FCam runtime sets up the exposure time and frame duration for the following frame and the sensor gain for the current one, as shown in Figure 14. A similar timeline is implemented on the Tegra 3 Prototype running the Omnivision 5650 CMOS sensor.

5.2 Porting FCam

As we have seen from the implementation details, porting the FCam API to a new platform requires deep knowledge of the underlying OS and camera stack. It is also necessary that some of the system drivers be flexible enough to accommodate all the parameters that the FCam runtime needs to set. Finally, it is important that consistent latencies can be computed in order to schedule actions correctly.

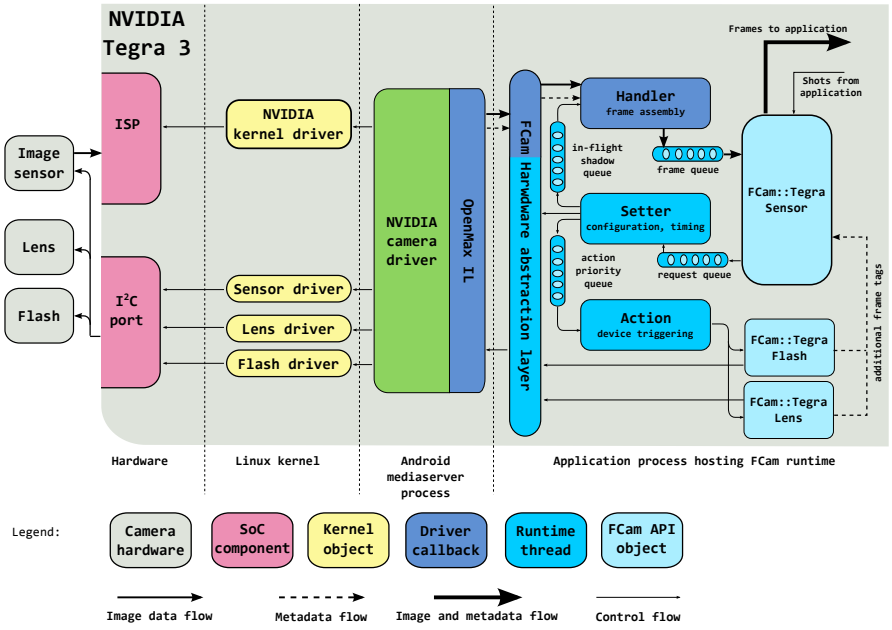

Fig. 13 A block-diagram of the FCam implementation on the Tegra 3 Prototype. Adapted from the original block-diagram by Eino-Ville Talvala [11].

The N900 implementation required the modification of the Video For Linux 2 (V4L2) kernel driver. Once the changes were done, the FCam runtime was implemented calling the device drivers directly. However, not all platforms allow for user applications to call functions running in hardware device drivers. Porting the FCam API to the Tegra 3 platform required tweaks at different levels of the software stack because only system processes are allowed to access the camera drivers in Android. User applications need to connect to the Android mediaserver process to send requests to the camera hardware.

As camera APIs evolve, it is expected these will become more flexible and enable high-throughput computational photography applications.

6 Image processing for FCam applications

FCam is meant for camera control, not for intensive image processing. For that there are other tools and APIs. In this section we describe three: OpenCV computer vision library, OpenGL ES 2.0 graphics API, and NEON intrinsics (NEON is a SIMD-type co-processor for ARM CPUs).

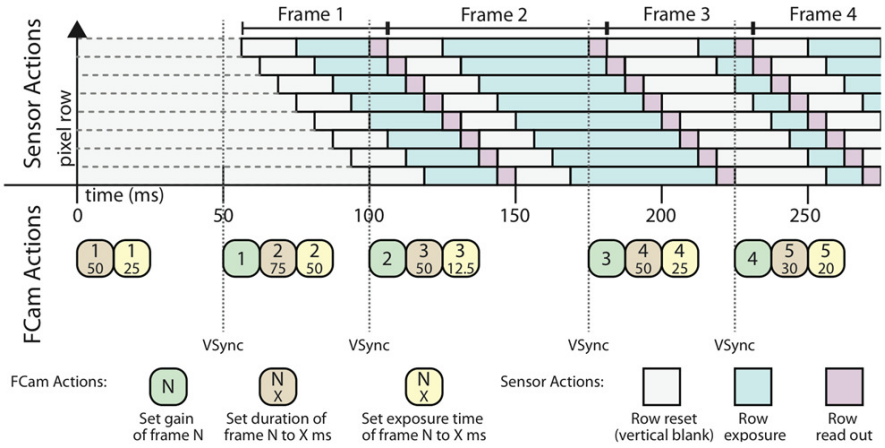

Fig. 14 Timeline of events for configuration of the image sensor. From Eino-Ville Talvala's dissertation [11].

6.1 OpenCV

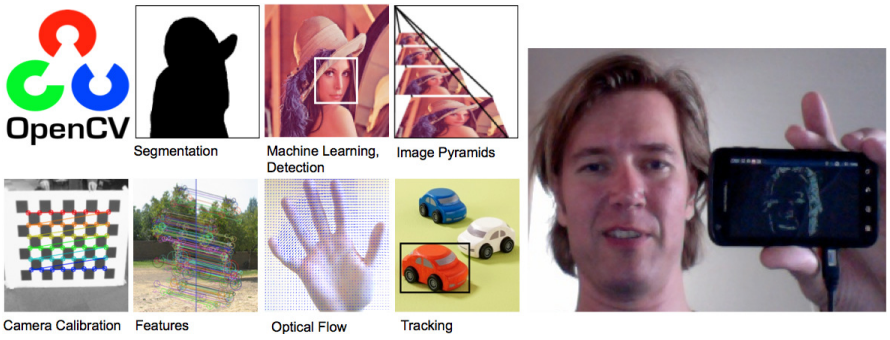

Fig. 15 OpenCV supports a large array of computer vision and image processing functions. The image on the right shows an OpenCV example running on an Android phone, doing a real-time edge detection on the input video stream coming from the camera.

OpenCV [3] is the de-facto standard computer vision API. It originated at Intel, and the original alpha version was released in 2000. After Intel stopped development of OpenCV, companies such as Willow Garage, Itseez, and NVIDIA have supported its development. It has over 500 algorithms for all types of computer vision and image processing tasks. It is available on most operating systems, including Windows,

Linux, MacOS, and Android. Figure 15 illustrates a subset of OpenCV functionality, and shows a sample OpenCV program running on an Android smartphone.

Originally OpenCV was developed and optimized for execution on Intel CPUs, but it has now been compiled for many different hardware platforms, including the ARM processor that powers most smartphones and tablets. A relatively recent development is the addition of the GPU module, that leverages the processing power of modern CUDA-capable graphics cards on desktop and laptop computers [10]. NVIDIA is also tailoring OpenCV so that it can use the hardware capabilities on its Tegra 3 mobile processor, which includes four ARM CPU cores, each with a NEON co-processor, and GPU supporting OpenGL ES 2.0.

OpenCV is a well-documented library that makes cross-platform vision or image processing applications easy. You can develop and test the application first on a desktop computer, and once the basic logic is working, easily port the application to a mobile device for further finetuning and optimizations.

6.2 OpenGL ES 2.0

In addition to the CPU, most computers have another powerful processor, the GPU (Graphics Processing Unit). The first generation of mobile graphics processors supported OpenGL ES 1.0 and 1.1, which had the traditional fixed-function graphics pipeline that makes the use of the GPU for anything other than traditional computer graphics cumbersome and inefficient. OpenGL ES 2.0 [9] increased the flexibility considerably by introducing segments called vertex and fragment shader, where the programmer can provide a compilable program. In particular, the fragment shader, which is run for each pixel, is a useful tool for image processing. The typical sequence is to upload the input image into a texture map, map the texture into a pair of triangles that cover as many pixels as the size of the output image, perform the image processing in the fragment shader, and finally read back the processed image to your own program.

6.3 NEON intrinsics

Most mobile devices such as smartphones and tablets use ARM CPUs, and most high-end mobile devices have also a co-processor called NEON [2]. NEON provides SIMD (Single Instruction, Multiple Data) architecture extension, allowing one instruction to operate on multiple data items in parallel. NEON extensions are particularly useful when you have to operate on several pixels in parallel, and can provide up to 10 times speed increase on some image processing algorithms. The NEON instructions can be accessed via C intrinsics, which provide similar functionality to inline assembly, and some additional features such as type checking and automatic register allocation, which make their use easier than inline assembly. The program-

mer needs to map the data to special NEON datatypes, then call the intrinsics that actually operate on the data, and finally map the processed data back to regular C data structures.

6.4 How should you choose which solution to use?

Each of the cited options for performing the image processing has its own limitations. If we list the choices in order of ease-of-use, OpenCV is probably the easiest to get started with. NEON is more flexible than OpenGL ES, which has some surprises such as limited floating point precision and limited storage precision for storing intermediate results. However, when one considers the speed of execution, and energy consumption, the order becomes the reverse. Pulli et al. [10] report measurements of several image processing algorithms implemented on ARM CPU, ARM with NEON instructions, and OpenGL ES. Use of GPU is more efficient both in time and energy than the other options, followed by NEON, and pure CPU remaining the last one. To make the developers' lives a bit easier, NVIDIA optimizes OpenCV for its Tegra mobile SoC so that the implementation internally uses multithreading (making use of up to four ARM cores), NEON intrinsics, and GPU via OpenGL ES, when it makes sense. Although the result is still not quite as optimal as if the programmer would hand-tune the whole application to these execution units, the user gets still a significant speedup compared to a naive implementation with relatively little programming effort.

7 FCam in teaching

One of the design goals of FCam was that it should be simple to use, and this feature makes it also an excellent tool for projects in university courses on computational photography and other related topics. In fact, an inspiration for FCam was a 2008 Stanford University course on Mobile Computational Photography (taught by Marc Levoy, Andrew Adams, and Kari Pulli). The students did the course projects using standard Symbian camera APIs, and that API was too restrictive to implement really interesting computational photography projects. Two years later, in winter 2010 FCam was ready for a new version of the same course (taught by Marc Levoy, Fredo Durand, and Jongmin Baek). The first homework for the students was to implement their own auto focus routine on a Nokia N900. That is a task in which professional engineers invest several months if not years, and would normally be too cruel a task for just getting started on programming a camera. The fact that all the students could finish the assignment in a week, and that some even delivered a better solution than the one the camera phone shipped with, shows that with good tools great things can be achieved.

After the first course, different universities have used FCam in their courses on a couple of dozens top schools in North and South America, Europe, and Asia. We next describe two representative projects from those courses.

7.1 A borrowed flash

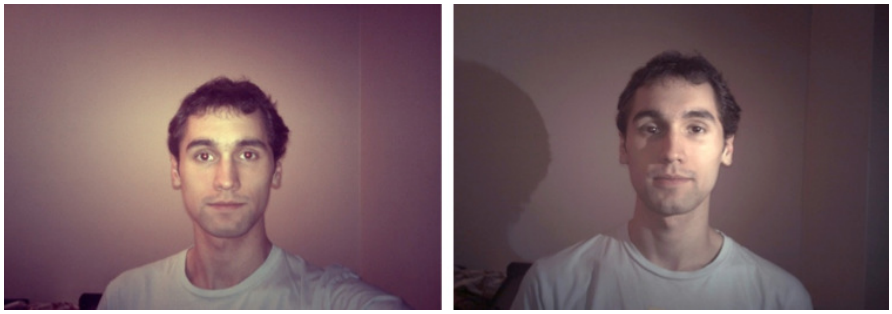

Fig. 16 A borrowed flash. The left image shows the results when the flash is too close to the camera: the eyes appear red. On the right image the Nokia N900 communicated with a second N900 so that the flash of the second camera illuminated the target while the first one took the image. Image by Michael Barrientos and David Keeler.

During the first FCam-based course, at Stanford in 2010, students Michael Barrientos and David Keeler decided to address the problem of red eyes due to flash. If the flash is close to the camera, the light enters the eye, is colored by the blood vessels feeding the retina, and is reflected back to the camera, and the eyes appear red, as illustrated in Figure 16 left. One red eye reduction technique briefly flashes a light, tricking the pupils to contract, which reduces the red eye phenomenon significantly. Another way is to move the light source further away from the camera.

In a camera phone there is not much room to move the flash more than a few centimeters away from the camera — that is, if the flash is still to remain in the same device. This project utilized the synchronization capabilities of the FCam API to borrow the flash from another device. When the main device is ready to take a photo, it signals the other device that intent over the Bluetooth wireless connection. The students were able to synchronize the two cameras accurately enough so that the flash on the second camera went off exactly as the first camera took the image, producing the image in Figure 16 right, where they eyes are not red. This project was implemented on a Nokia N900.

7.2 Non-photorealistic viewfinder

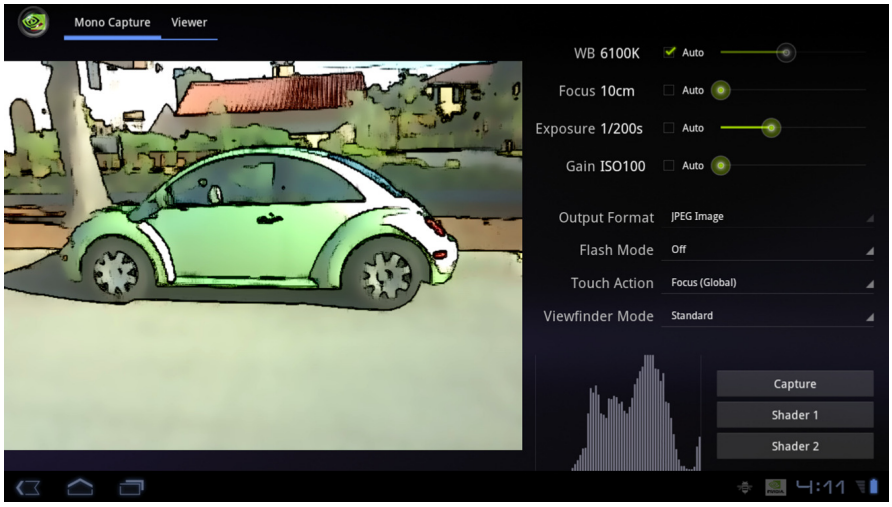

Fig. 17 A non-photorealistic viewfinder on NVIDIA Tegra 3 tablet. An OpenGL ES 2.0 fragment shader filters the viewfinder frames in real time to give it a live video cartoon look. Image by Tony Hyun Kim and Irving Lin.

During the winter of 2012 version of the Stanford course (taught by Jongmin Baek, David Jacobs, and Kari Pulli), students Tony Hyun Kim and Irving Lin developed a non-photorealistic camera application. The application was developed for the Tegra 3 prototype board that the students used to implement their assignments. Output frames are post-processed using OpenGL ES before being displayed. Two shaders were written to give the non-photorealistic feeling: a bilateral filtering shader and an edge detection shader. The bilateral filter creates a flat, cartoonish rendition of the viewfinder image, and the edge detector further enhances the edges between different regions, providing more of a hand-drawn feeling. FCam was used to control flash to help separate foreground from background. The resulting application runs at an interactive frame rate on the NVIDIA Tegra 3 GPU. A screenshot is shown in Figure 17. Such an effect could be easily added to a camera application in a commercial device.

8 Conclusions

We have presented the FCam API and its applications to mobile computational photography. As we discussed, traditional camera APIs provide little control over the

camera subsystem. By treating the camera subsystem as a pipeline in which its state is associated with a request, the FCam API proves powerful for computational photography applications because:

1. it provides deterministic and well defined control over image bursts,
2. it allows for novel imaging effects by providing tight synchronization with the flash, lens, and other devices, and
3. it enables the user to build her own auto control algorithms targeting specialized applications.

In our discussions we highlighted each of these qualities by showing relevant applications. We showed how to use the per-frame control to program an HDR imaging application, how to use the synchronization capabilities to implement a borrowed flash, and how to extend the traditional metering algorithms for efficient HDR capture and all-in-focus image capture. In addition, a complete camera application can be written using the FCam API and enhanced with the image processing capabilities of today's mobile phones and tablets. The API is simple enough for university students to tackle computational photography projects. The non-photorealistic preview application, developed by students in a Computational Photography course, highlights how we can integrate camera control with image processing on the GPU to produce new stylized images.

The example applications and code snippets we have presented use a single camera; however, the number of mobile devices that have two or more cameras is rapidly increasing, opening the door for new API extensions. In [12] we have started to address multiple camera enumeration and synchronization in FCam.

Acknowledgements

The main architects behind the Frankencamera architecture and FCam API were Eino-Ville Talvala and Andrew Adams. The project leaders were Marc Levoy, Mark Horowitz, and Kari Pulli.

There have been several FCam courses at various conferences, presented by the authors, together with Marius Tico, Timo Ahonen, and Andrew Adams.

The authors would like to thank Orazio Gallo, David Pajak, Jongmin Baek, and David Jacobs for their invaluable comments and suggestions that helped improve the quality of the manuscript.

References

1. Adams, A., Talvala, E.V., Park, S.H., Jacobs, D.E., Ajdin, B., Gelfand, N., Dolson, J., Vaquero, D., Baek, J., Tico, M., Lensch, H.P.A., Matusik, W., Pulli, K., Horowitz, M., Levoy, M.: The Frankencamera: An Experimental Platform for Computational Photography. ACM Transactions on Graphics **29**(3) (2010)

2. ARM: Introducing NEON Development. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/ch01s04s02.html (2009)
3. Bradski, G., Kaehler, A.: Learning OpenCV: Computer Vision with OpenCV Library. O'Reilly Media (2008)
4. Cossairt, O., Zhou, C., Nayar, S.K.: Diffusion Coded Photography for Extended Depth of Field. ACM Transactions on Graphics **29**(4) (2010)
5. Debevec, P.E., Malik, J.: Recovering high dynamic range radiance maps from photographs. In: Proceedings of SIGGRAPH, pp. 369–378 (1997)
6. Eisemann, E., Durand, F.: Flash photography enhancement via intrinsic relighting. ACM Transactions on Graphics **23**(3), 673–678 (2004)
7. Gallo, O., Tico, M., Manduchi, R., Gelfand, N., Pulli, K.: Metering for Exposure Stacks. In: Eurographics (2012)
8. Mertens, T., Kautz, J., Van Reeth, F.: Exposure fusion. In: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications (2007)
9. Munshi, A., Ginsburg, D., Shreiner, D.: OpenGL ES 2.0 Programming Guide. Addison-Wesley Professional (2008)
10. Pulli, K., Baksheev, A., Kornyakov, K., Eruhimov, V.: Realtime Computer Vision with OpenCV. ACM Queue **10**(4) (2012)
11. Talvala, E.V.: The Frankencamera: building a programmable camera for computational photography. Ph.D. thesis, Stanford University (2011)
12. Troccoli, A., Pajak, D., Pulli, K.: FCam for multiple cameras. In: Proc. SPIE 8304 (2012)
13. Vaquero, D., Gelfand, N., Tico, M., Pulli, K., Turk, M.: Generalized Autofocus. In: IEEE Workshop on Applications of Computer Vision (WACV) (2011)