FCam for Multiple Cameras

Alejandro Troccoli, Dawid Pajak, and Kari Pulli
NVIDIA Research, Santa Clara, CA, USA

ABSTRACT

The Frankencamera (FCam) architecture and API enables precise control over the camera in computational photography applications. We present an extension to FCam API for systems equipped with multiple cameras. The proposed extension allows for an enumeration of cameras and their corresponding properties, such as position or orientation. In addition, we explicitly support camera synchronization, either through hardware mechanisms or software primitives. If hardware synchronization is available, cameras can be grouped together under a concept of a multi-sensor. Otherwise, multiple camera streams are scheduled asynchronously and synchronized using our software control primitives.

Keywords: Computational photography, Camera control, API

1. INTRODUCTION

Computational photography is an active area of research that enhances digital imaging by combining non-traditional sensing with computation to produce a new image that a traditional camera would not be able to capture in a single shot. Examples of computational photography applications include high-dynamic-range (HDR) imaging,[1] flash no-flash imaging,[2] panoramic stitching, and extended depth of field.[3] For a long time, computational photography applications were limited to research-like environments, with cameras fixed to a tripod and connected through a cable to a host computer responsible for all the processing. However, recent advances in design and performance of system on a chip (SoC) solutions in mobile devices enabled these applications on portable platforms.

The Frankencamera architecture and FCam API[4] defined a programmable camera and programming model for computational photography. Adams et al.[4] showed two implementations of the Frankencamera concept. The first one is based on a custom-built F2 camera, while the second one uses Nokia N900 mobile phone running a modified software stack. Gelfand et al.[5] implemented metering for multi-exposure on mobile devices using the

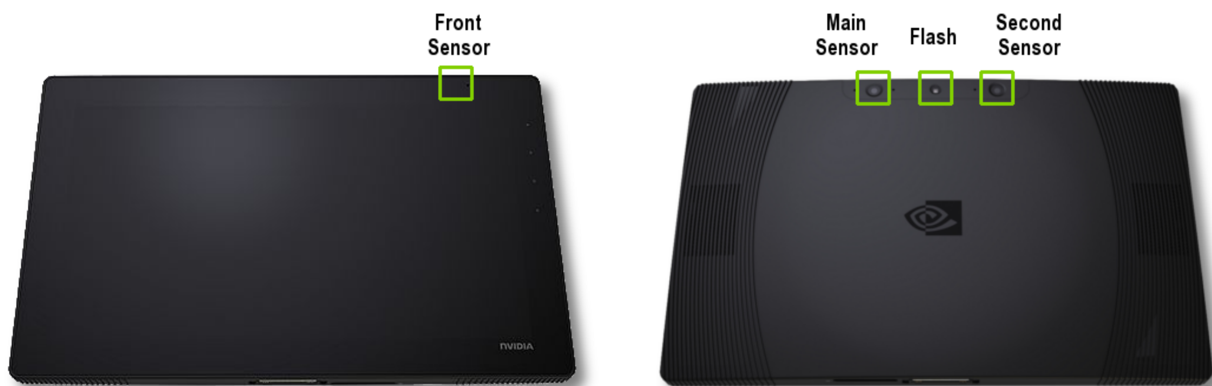

Figure 1. **The Tegra 3 prototype board.** The Tegra 3 prototype board has 3 sensors, the main sensor and the secondary sensor are identical and can be used for stereo image capture. There is a flash unit between the two sensors. The third sensor faces the user and is on the same side as the display.

Nokia N900 FCam API implementation. In this work we extend the implementation of FCam API to support NVIDIA Tegra 3 prototype tablet.

The original FCam API addressed the access to single-sensor devices. However, most camera phones have at least two cameras, one facing the user and another, forward-facing camera. In addition, some mobile devices, such as the LG Optimus 3D phone and the NVIDIA Tegra 3 developer board shown in Figure 1, have a pair of cameras supporting stereo imaging. In this paper we amend the API and develop an extension of FCam API to support camera systems with multiple sensors.

2. THE FRANKENCAMERA ARCHITECTURE

The Frankencamera architecture, illustrated in Figure 2, defines an abstraction for the data and control flows, as well as for synchronization between the applications processor, the image sensor, the image processor, and external devices such as the lens or the flash. This abstract architecture was set up under a set of design goals that would allow for an easy programming of computational photography applications. Among these goals, the most salient ones are: the ability to manipulate the sensor, lens, and camera image processor settings on a per-frame basis, preferably at a video rate or as fast as the hardware allows, and the ability to label each output frame with the actual settings used to capture the images.

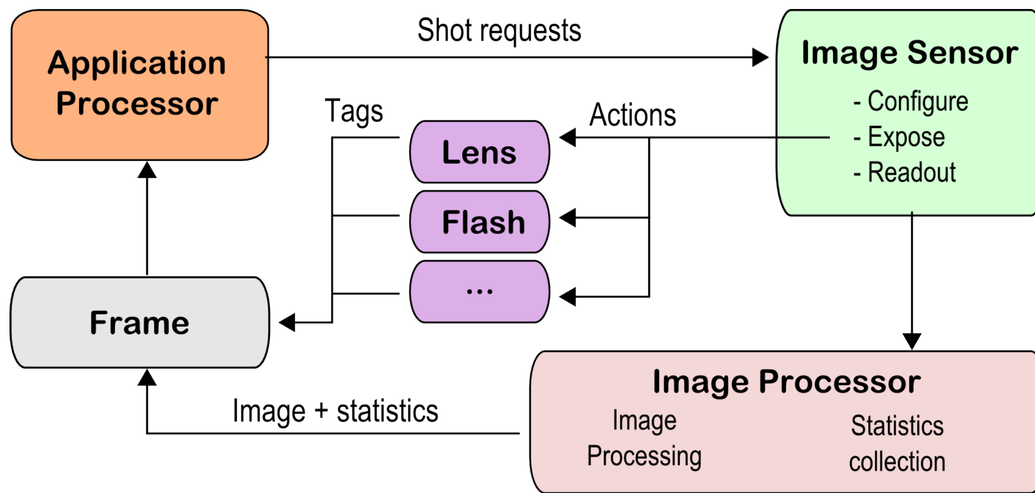

Figure 2. **The Frankencamera architecture.** The application processor generates shot requests that are sent to the image sensor. When the sensor is exposed any actions related to the shot are executed on the lens, flash or other devices. The image processor accepts the image data, computes statistics and performs any requested image processing tasks. Finally, the image, the statistics plus tags from the devices are combined into a frame. Adapted from Eino-Ville Talvala.[6]

In the architecture, the application processor provides the computing resources to operate and manipulate each of the camera components. It should also have spare computing power to handle the output images and do the necessary post-processing. In case of HDR imaging, the application processor would need to provide the resources for combining individual exposures into an HDR image and tonemapping it for display. Computing resources for these tasks could be in the form of CPU-cores, other units in the application processor such as the graphics processing unit (GPU), or dedicated image processing hardware. Examples of actual application processors that have been used in Frankencamera implementations are NVIDIA's Tegra 3 and Texas Instrument's OMAP 3.

The image sensor is an abstraction of the physical imaging sensor. It is typically a pipelined component that transforms requests into frames. Whereas in most previous camera APIs the sensor has a global state, in FCam each request contains a set of parameters that together define the hardware state to produce a frame. Therefore the state is associated with frame requests, and travels from the frame through the different stages of the image capturing and processing pipeline. The frame request parameters include the actual imaging sensor configuration

settings, like exposure and gain, post-processing parameters handled by the imaging processor like white balance, color tone adjustment and output format; and a list of actions for external devices that should be synchronized with the exposure. Each frame the comes out of the sensor is queued to be later retrieved asynchronously by the application, and will include the actual configuration parameters used in its capture and the request used to generate it.

The imaging processor takes the output of the image sensor and performs the requested post-processing steps. It converts the raw image into the desired format, performing demosaicking, white balancing, and color tone correction if requested. In addition, it computes statistics on the raw image pixels such as a histogram and a sharpness map over a programmable region to assist in metering and autofocus.

Devices such as lens or flash are components that are controlled independently of the image sensor and add functionality to the camera. Quite often these devices must be coordinated with the image sensor to produce the desired result. For example, a flash should be synchronized with respect to the exposure. Devices provide actions that the application programmer can attach in the shot request. These actions have a time line relative to the beginning the the frame exposure, and require precise knowledge of latencies for correct scheduling and coordination.

In multiple-sensor systems, each sensor has its own internal pipeline and abstract imaging processor.[4] The imaging processor could be implemented as a separate hardware unit or a single time-shared unit. In this paper we present different multiple-sensor implementations, their corresponding synchronization mechanism, and possible applications. The first case we consider is that of a stereo imaging system where the two imaging sensors can be synchronized by hardware to trigger the exposures concurrently. In addition, we consider the case where the imaging sensors are allowed to run asynchronously but some level of software synchronization is desired and define a set of software objects for this purpose. A mobile programmable system with multiple sensors is the NVIDIA Tegra 3 prototype device shown in Figure 1. Recently Heinzle et al.[7] presented a computational stereo camera system where the programmable control loop allows for setting physical properties of camera arrangement.

3. ARCHITECTURE FOR MULTIPLE SENSORS

A large number of mobile devices currently available in the market have more than one imaging sensor. Among these systems, most have two sensors, one facing the user and one pointing away from the user. Typically, the user-facing sensor is used for video chat, since it captures the user's face while she is looking at the screen. A few other mobile devices have 3 sensors: an outward-looking stereo pair and a user-facing sensor. It is not difficult to imagine a computational photography application that feeds from more than one sensor at the same time. In addition, synchronized dual sensors can be used to capture stereo pairs and perform 3D reconstruction tasks. However, the previous version of the FCam API did not address these scenarios in detail, leaving the door open for future implementations. We came up with the following requirements when extending the Frankencamera architecture and FCam API for multiple sensors.

1. The API needs to enumerate the sensors that are available, and report their geometry and other relevant properties. In enumerating the sensors one should be able to identify each physical sensor unequivocally. Terms like front-facing and back-facing, left and right could lead to ambiguities; to avoid this, we propose that each sensor is assigned a position and orientation with respect to a global coordinate system.

2. The API needs to properly handle system-wide devices, such as multiple flashes, and per-sensor devices, such as lenses. In a system with multiple sensors it is likely that lenses attached to each sensor have different properties. A lens and a sensor are tightly coupled, so it makes sense to refer to a lens as the lens attached to a particular sensor; on the other hand, a flash and a sensor are not coupled, the same flash may be shared by multiple sensors.

3. The API needs the ability to handle a group of similar synchronized sensors; we call such a group a sensor array. The individual sensors in the sensor array can be synchronized by the system, leaving the details on how to achieve this up to the actual implementation. A pair of hardware-synchronized sensors for stereo capture is an example of a sensor array of size 2.

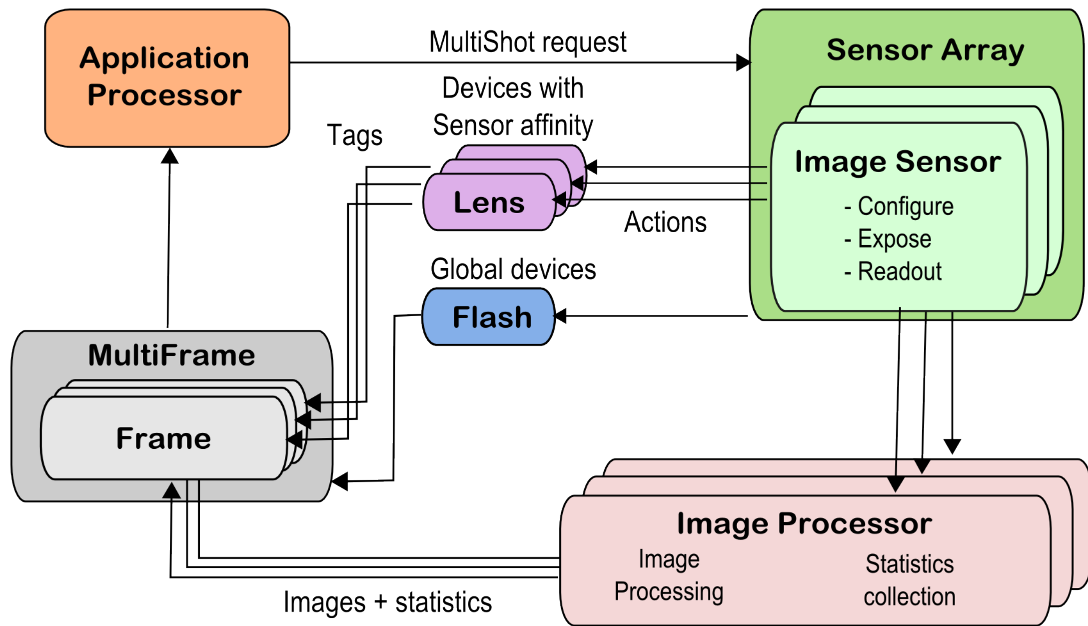

Figure 3. **The sensor array.** The sensor array consists of a set of identical sensors that can be synchronized by the system. A request is made in the form of a multi shot which contains individual shots for each sensor and the corresponding actions for the devices that have sensor affinity. Actions for system devices are added to the multi shot. Each sensor produces a frame with its own statistics and tags, all frames are combined in a multi frame if the system packs the frames together.

4. The API needs synchronization primitives that can be inserted in sensor imaging stream to synchronize exposure of several otherwise asynchronous sensors.

To satisfy the unequivocal identification requirement 1, we define a coordinate system like the one used in the OpenGL graphics API: the negative z-axis is aligned with the viewing direction of the "main" camera; typically this would mean that the positive z-axis points toward the user on a smart phone or a tablet. The y-axis points up in the default orientation of the device when used for imaging (often the landscape orientation for either a smart phone or a tablet), and the x-axis completes a right handed coordinate system, usually pointing to the user's right. The origin of the coordinate system is ideally at the optical center of the main camera of the system. The location information can be used to find the baseline of a stereo camera, for example, and the orientation helps in disambiguating between the user-facing and outward-facing sensors.

In addition, we expand the concept of a device with an affinity. A device with sensor affinity needs to know about the sensor it is coupled with. An example of such a device is a lens. For devices that have sensor affinity we will need to refer to them as the device associated with the particular sensor. On the other hand, a device that has camera affinity is not tightly coupled with any particular sensor. The flash is the most salient example. Other devices with system affinity are the gyroscope and accelerometer. If appropriate, devices with system affinity should have their geometry defined to indicate their position and orientation. A flash device will have a position and orientation, which will allow the user to determine if the flash is facing the same direction as the sensor is. Figure 3 shows the flow of requests, actions, and frames between the application processor and the sensor array.

We add a **sensor array** concept to satisfy requirement 3. A sensor array provides a unified view of two or more identical sensors that can have exposures synchronized. As such, requests for captures to a sensor array may be constrained to frames with identical resolution, output format, and frame rate. In addition, all requests will share the same queue and will not require any other synchronization mechanism. To make sensor arrays interesting, the user should be free to provide different request parameters for particular sensors where possible. For example, one could desire to have the two sensors in a stereo pair capture the scene with different exposure

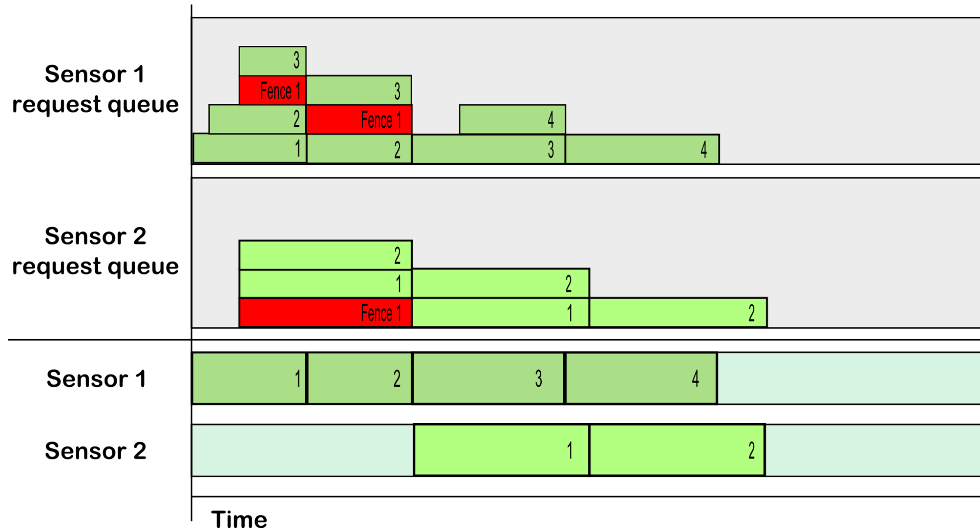

Figure 4. **Software synchronization using fences.** Synchronization fences can be introduced in a sensor request queue to synchronize the exposures. In this example, Shot 3 on Sensor 1 is synchronized with Shot 1 in Sensor 2. The fence halts the processing of requests in a queue until all queues have reached the fence.

settings. Even more, devices that have sensor affinity should have the ability to be controlled independently for each sensor. We could perform the focal sweep for autofocusing in opposite directions in order to focus the camera twice as fast compared to a single camera. A sensor array does not preclude addressing each sensor of the array independently to stream frames asynchronously. Each sensor in the array will be enumerated as a sensor and can be individually controlled. However, if the option of using hardware synchronization is desired the sensors should be addressed as an array of sensors.

The last new element of the multiple sensor architecture is a synchronization object. The synchronization object is placed in a request queue in one of two conditions: a release condition or a wait condition. When the request queue is being processed, if a wait is found and the condition is not satisfied the sensor will be halted until the condition is satisfied. On the other hand, a release in a request queue will change the value of the synchronization object. On a release, any wait request is re-evaluated and if satisfied the halted sensor is allowed to proceed. Different kind of synchronization objects can be defined, the most useful one being the fence. A fence halts the processing of requests until all queues reach the fence, at which time all request queues are allowed to proceed in parallel. Fence synchronization is illustrated in Figure 6.

4. PROGRAMMING FOR MULTIPLE SENSORS

We now describe how to program with the new extensions added to the FCam API for handling multiple sensors. As a reminder, the basic components of the FCam API are sensors, devices, shots, and frames.[4]

Sensor. We extend the Sensor class with static members `numberOfSensors` and `properties` to enumerate the properties of each sensor. The properties structure includes the position and rotation of the sensor and maximum sensor resolution. The `Sensor` constructor takes an index to the sensor to construct. In addition we add `numberOfArrays` and `array` to enumerate the available sensor arrays and the included sensors. Figure 5 shows an example of sensor enumeration.

SynchronizationObject and Fence. We provide a new abstract class `SynchronizationObject` derived from `Device` with members `release` and `wait`. We also provide a `ReleaseAction` that can be added to a `Shot` to trigger the release of the object in coordination with the exposure of a frame. To introduce a wait on a queue we added the `waitForSynchornizationObject` member to the `Shot` class. A wait is not an action because actions are executed asynchronously from requests. Instead, when a Sensor is processing a request it looks for `SynchronizationObject` and if one is found it waits on its condition. We also provide the `Fence` class

```
int i = 0;
for(; i < Sensor::numberOfSensors(); ++i) {
    if (isFacingUser(Sensor::properties(i))
        break;
}

if (i == Sensor::numberOfSensors())
    // No user facing sensor found!
    exit(1);

// Construct the user facing sensor
Sensor sensor(i);
Lens lens(sensor);

...

bool isFacingUser(SensorProperties prop) {
    return (prop.rotation[2][2] > 0.1f);
}
```

Figure 5. **Sensor enumeration.** A minimal code example to demonstrate sensor enumeration and querying properties.

derived from `SyncrhonizationObject`. On construction it takes a positive integer with the number of queues to synchronize. When all queues reach a wait condition on the fence the requests are allowed to proceed. The `SynchronizationObject` is smart enough that it can be used safely in conjunction with the `stream` member of `Sensor`, meaning that it will remember its internal count over successive invocations of `release` and `wait`, such that each `wait` is paired with the correct number of `release` invocations.

```
// Open front and back sensor
Sensor backsensor(0), frontsensor(1);

// Create a fence for the two sensors
Fence fence(2);

Shot shot;
shot.waitForSynchronizationObject(fence);
...

backsensor.capture(shot);
frontsensor.capture(shot);
```

Figure 6. **Software synchronization.** A new Fence object is created and the shot is requested to wait on the object before proceeding.

In order to handle sensor arrays we provide a new set of classes **MultiSensor, MultiShot, MultiFrame,** and **MultiImage**. A `MultiSensor` is constructed by passing a sensor array index. It acts like a regular `Sensor` except that it takes requests to capture and stream in the form of instances of the `MultiShot` class. A `MultiShot` contains multiple `Shot` requests, one for each sensor in the array. In addition, if for a particular capture we want to use a subset of the sensors in the array we can disable the unwanted sensors and the requests for the disabled

sensors will be ignored. For devices with sensor affinity, the corresponding actions are added to the individual sensor shots. For devices with system affinity, on the other hand, the actions are added to the `MultiShot` request.

```
MultiSensor sensor(0);   // 0 is the array of 2 sensors in the Tegra board
Flash flash;

sensor.attach(&flash);

MultiShot shot;
shot[0].exposure = 50000;  // Configure the shot for the left camera
shot[0].gain     = 1.0f;
shot[1] = shot[0]; // Right camera shot with same parameters

// The sensor array will allocate the proper image for the given sensor array
shot.image = sensor.allocateImage(640, 480);

// Capture no-flash stereo frame
sensor.capture(shot);

Flash::FireAction fire(&flash);
fire.duration   = flash.minDuration();
fire.time       = 0;
fire.brightness = flash.maxBrightness();

shot.addAction(fire); // Add an action for a system device
shot.image = sensor.allocateImage(640, 480);

// Capture flash stereo frame
sensor.capture(shot);

MultiFrame noflashframe = sensor.getFrame();
MultiFrame flashframe   = sensor.getFrame();

// Get the left no flash image
Image leftNoFlash       = noflashframe[0].image();
```

Figure 7. **Flash/No Flash stereo capture.** The multi shot request is issued to the sensor array to trigger a stereo capture. The first request is captured without flash, the second one adds a flash action.

5. DISCUSSION

Having introduced the new architecture and its programming model we now discuss new applications that are enabled by the API extensions and that we seek to explore in the near future.

5.1 Fast focusing and continuous focusing

Traditional focusing requires sweeping the lens from the far to the near plane looking for the position that provides the best contrast for a given region of the image. Moving the lens is a time consuming task. On a system with multiple sensors that plane sweep space can be partitioned among the sensors, reducing the time of the autofocus sweep. For example, in the Tegra 3 prototype board of Figure 1 we can use the stereo camera pair to partition the sweep space in two. In a similar manner, one could use one sensor to continuously find the

best focus for the scene. Every time the focus changes, the primary sensor would update its focus position. This approach can provide continuous auto focus without the visual disruptive effects that a focus sweep produces.

5.2 Face tracking and metering

Face tracking can be a computationally intensive task that usually requires down-sampling the image to a smaller resolution. We could program a pair of stereo sensors to run asynchronously at different resolutions. A low resolution stream would could be used for face tracking and metering to keep the face well exposed and focused. The newly computed settings could then be used on a high resolution stream.

5.3 Stereo capture

Stereo capture is simple to achieve with the new programming model as already shown in Figure 7. A pair of stereo images can be the starting point for dense and sparse real-time 3D reconstruction algorithms on a mobile device.

5.4 Picture in picture video chat

By running the user facing and forward facing sensors asynchronously, one could compose a picture-in-picture video feed with the contents of both streams.

6. CONCLUSIONS

We have presented an extension of the FCam API for multiple sensors. We have added sensor enumeration and identification through an unambiguous position and orientation, the concept of a sensor array for handling system synchronized sensors, and software synchronization when hardware synchronization is not available. We showed examples of how the new extensions allow for multi-sensor programming, and we discussed computational photography applications that are enabled by the new extensions.

REFERENCES

[1] Debevec, P. E. and Malik, J., "Recovering high dynamic range radiance maps from photographs," *Proceedings of SIGGRAPH*, 369–378 (1997).

[2] Eisemann, E. and Durand, F., "Flash photography enhancement via intrinsic relighting," *ACM Transactions on Graphics* **23**(3), 673–678 (2004).

[3] Cossairt, O., Zhou, C., and Nayar, S. K., "Diffusion Coded Photography for Extended Depth of Field," *ACM Transactions on Graphics* **29**(4) (2010).

[4] 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., and Levoy, M., "The Frankencamera: an experimental platform for computational photography," *ACM Transactions on Graphics* **29**(3) (2010).

[5] Gelfand, N., Adams, A., Park, S. H., and Pulli, K., "Multi-exposure imaging on mobile devices," *Proc. of the international conference on Multimedia*, 823–826 (2010).

[6] Talvala, E.-V., *The Frankencamera: building a programmable camera for computational photography*, PhD thesis, Stanford University (2011).

[7] Heinzle, S., Greisen, P., Gallup, D., Chen, C., Saner, D., Smolic, A., Burg, A., Matusik, W., and Gross, M., "Computational stereo camera system with programmable control loop," *ACM Transactions on Graphics* **30**(4) (2011).