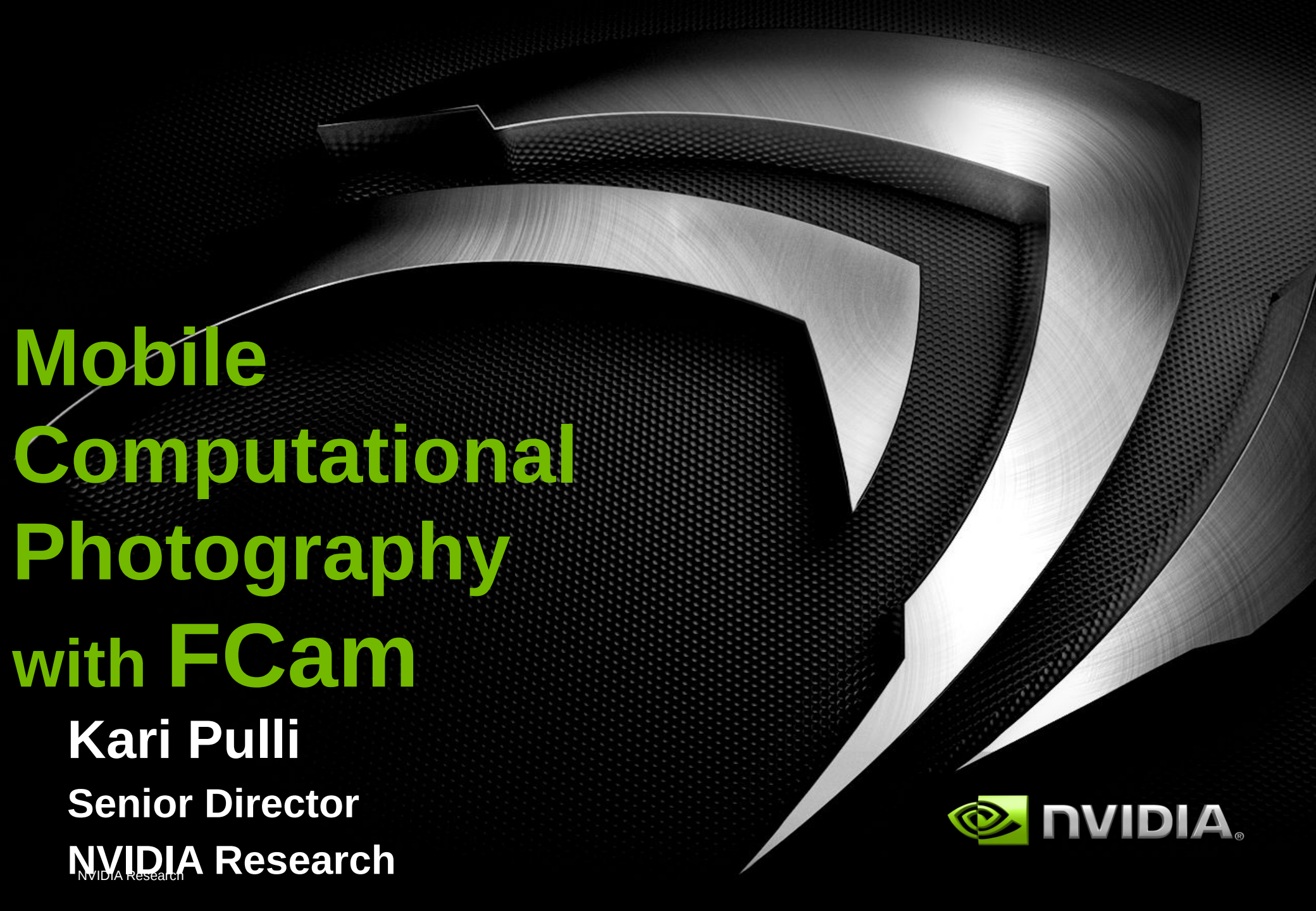


Mobile Computational Photography with FCam

Kari Pulli

Senior Director

NVIDIA Research

Computational photography

- Try to overcome the limitations of normal cameras
  - Usually: Take several images
    - combine, compute, and tease out more information
  - Sometimes also change the camera itself
- Mostly in the lab
  - researchers, professionals, hard-core hobbyists
  - camera on a tripod, situation is static or at least controlled
- Use high-end cameras
  - big optics and sensors -> high image quality
- Processing later, offline on a PC

Mobile Computational Photography

- From labs to everybody
  - camera phones are consumer products
- Camera phones pose new challenges
  - small optics and sensors -> high noise
  - handheld
- Online computational photography
  - interactive loop between
    - user, computation, and imaging
  - immediate feedback (do I need to recapture?)
  - instant gratification, immediate sharing

Viewfinder Alignment

- A pixel-accurate alignment algorithm that runs at 320x240 at 30fps on a Nokia N95 Camera-Phone
- Low-noise viewfinding
  - Align and average a moving window of previous frames
- Panorama capture
  - Automatically take new images when the view has moved to a new location
- …

Computational photography

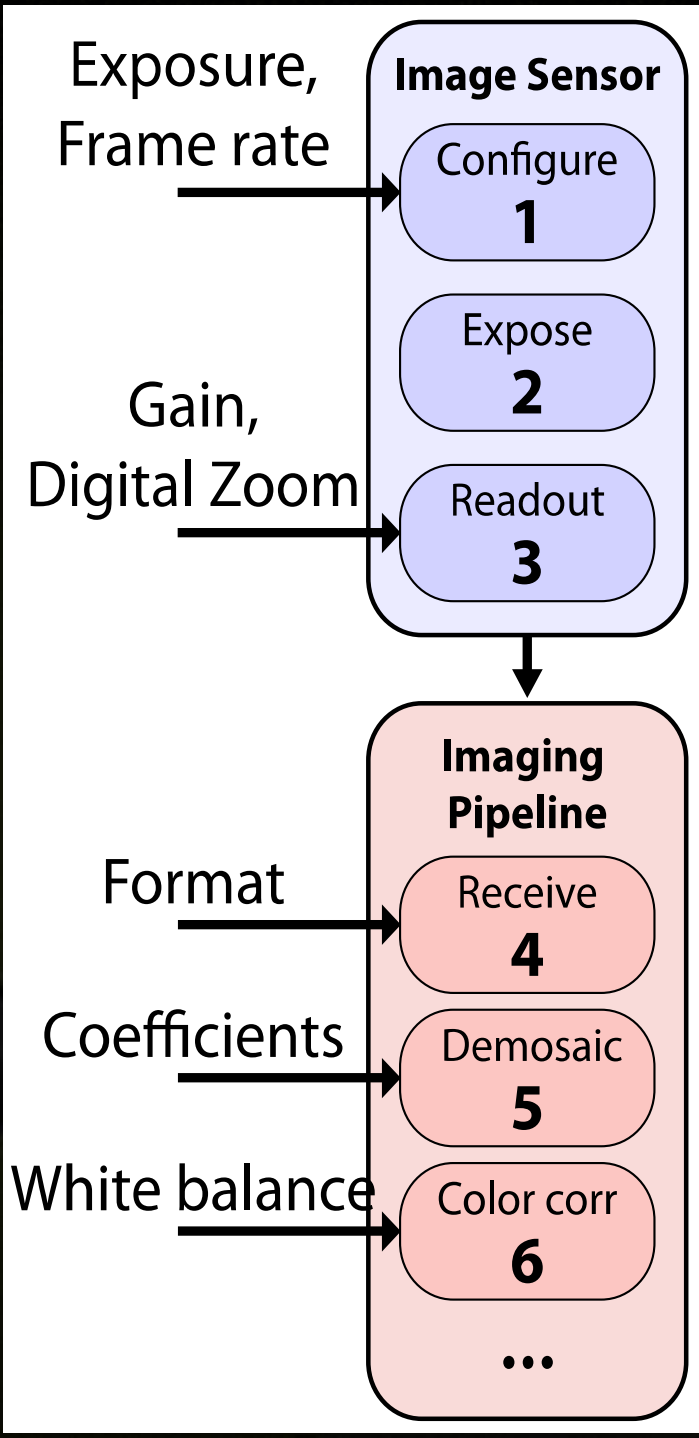
```
for (...) {
  Change camera settings
  Take picture
}
Combine the pictures
```

1: Platform is closed

- No control over
  - exposure time
  - white balance
  - focus
  - frame rate
  - image format/resolution
  - post-processing pipeline parameters
  - metering algorithm
  - autofocus algorithm
- “Real” cameras can’t be reprogrammed at all

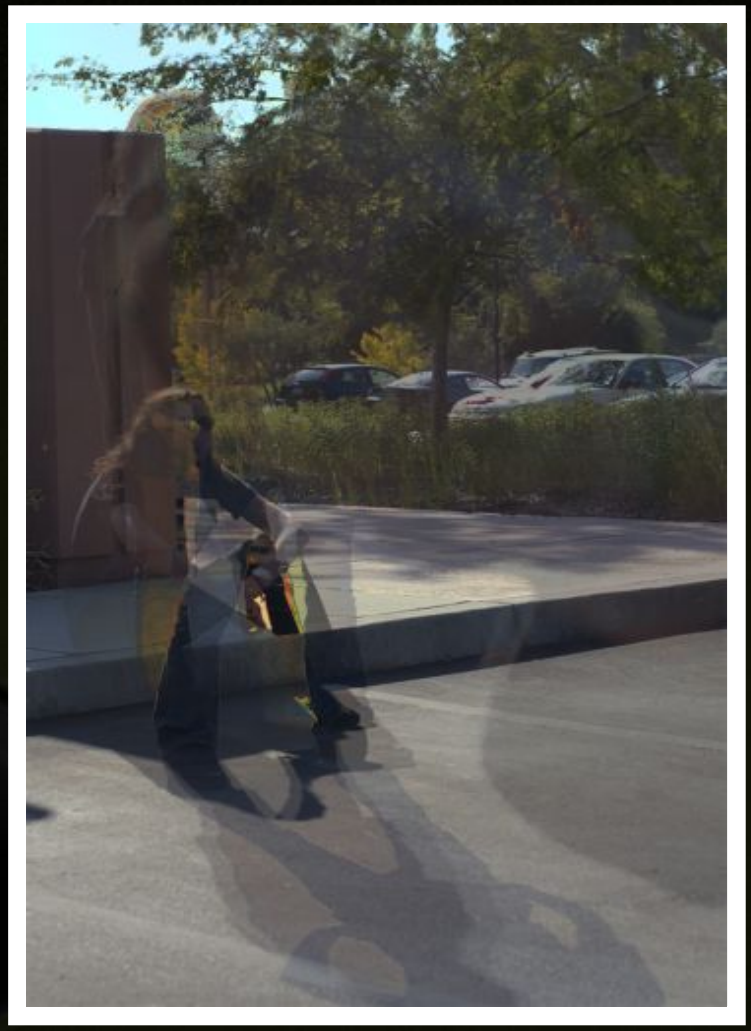
2: Wrong sensor model

- Real image sensors are pipelined
  - while one frame exposing
  - next one is being prepared
  - previous one is being read out
- Viewfinding / video mode:
  - pipelined, high frame rate
  - settings changes take effect sometime later
- Still capture mode:
  - need to know which parameters were used
  - reset pipeline between shots slow



So What?

- So the user has to wait another couple of seconds
- How bad is that?

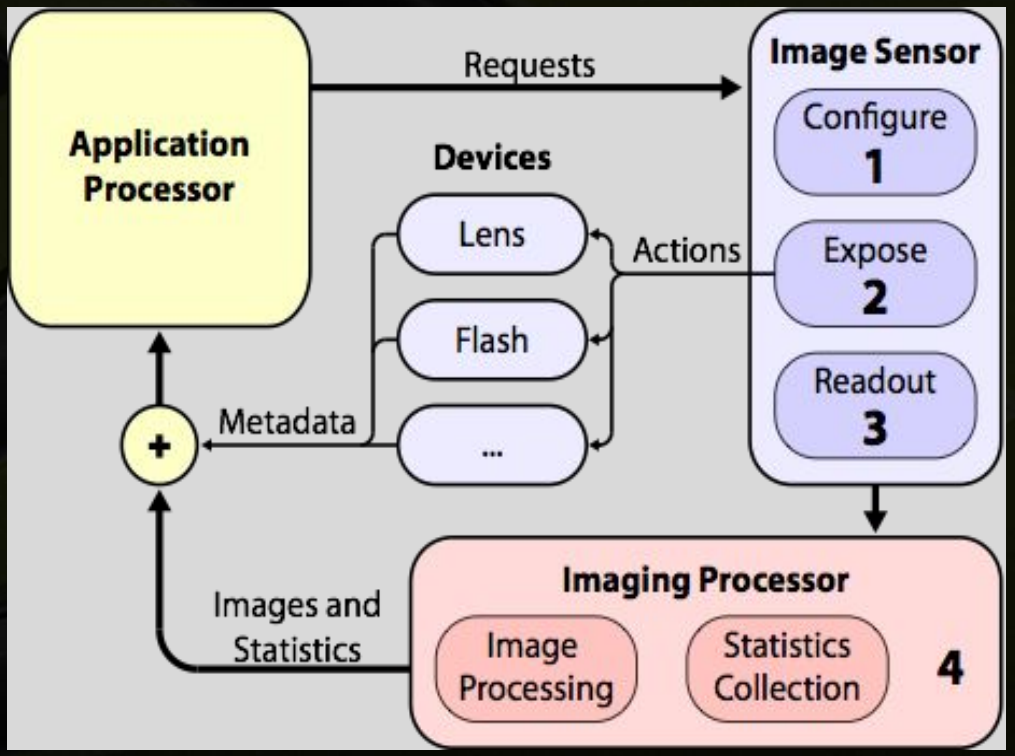


Computational Photography

```
for (...) {
  Change camera settings
  Take picture
}
Combine the pictures
```

The FCam Architecture

- A software architecture for programmable cameras
  - that attempts to expose the maximum device capabilities
  - while remaining easy to program



Sensor

- A pipeline that converts requests into images
- No global state
  - state travels in the requests through the pipeline
  - all parameters packed into the requests

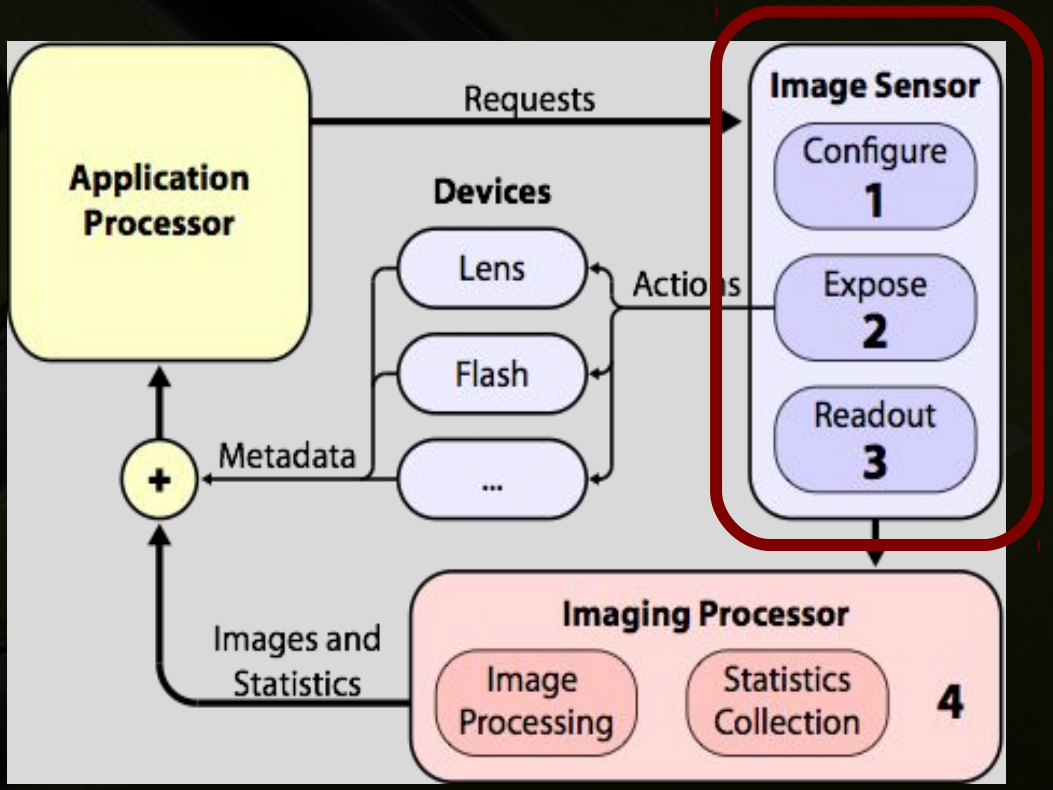
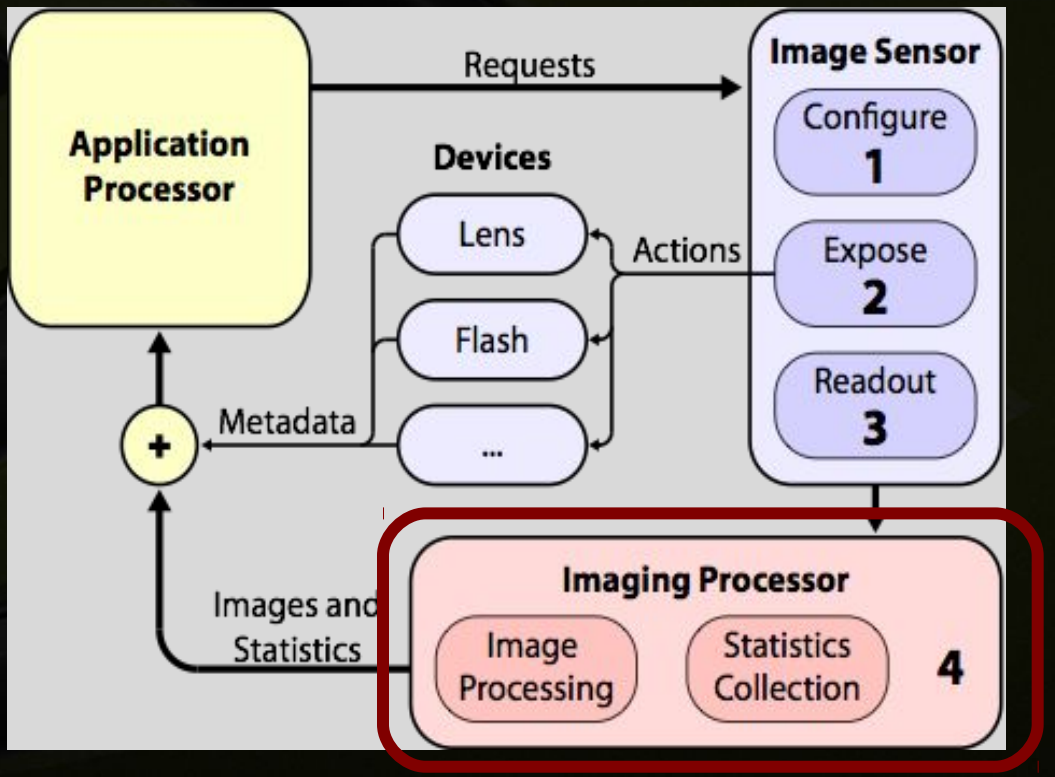


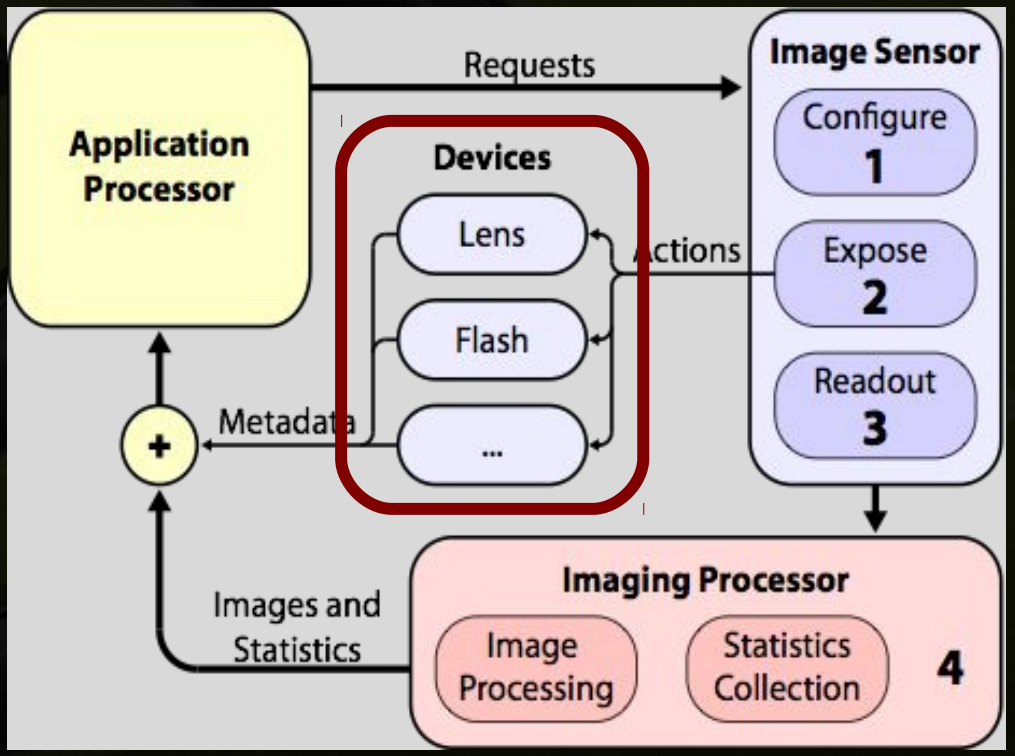
Image Signal Processor (ISP)

- Receives sensor data, and optionally transforms it
  - untransformed raw data must also be available
- Computes helpful statistics
  - histograms, sharpness maps



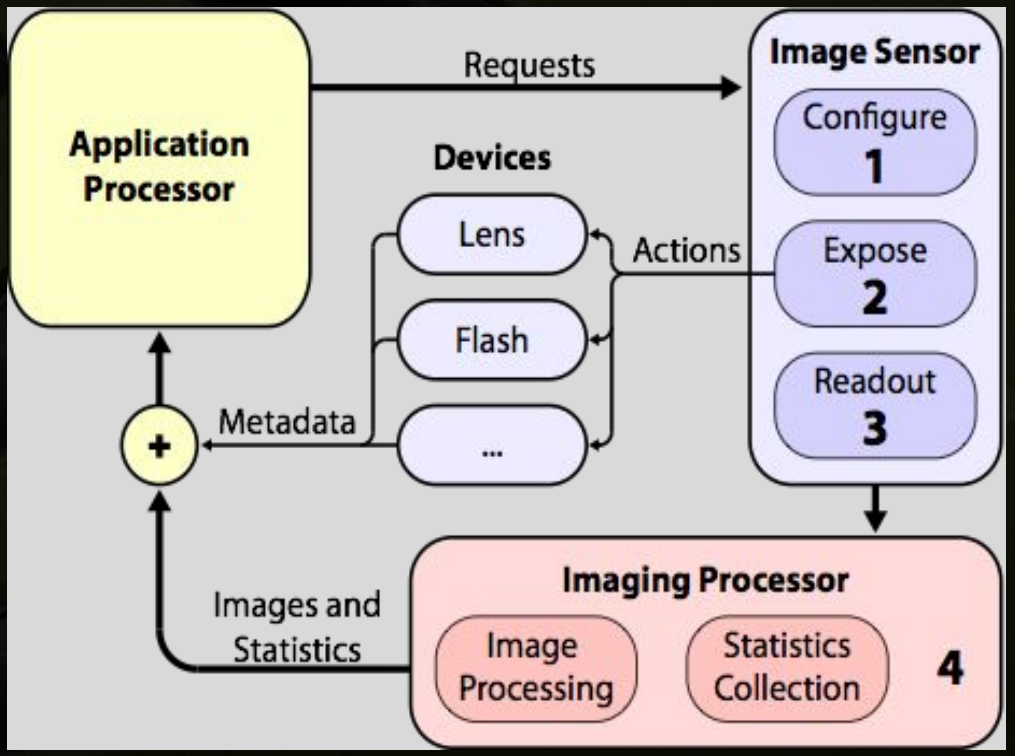
Devices

- Devices (like the Lens and Flash) can
  - schedule Actions
    - to be triggered at a given time into an exposure
  - Tag returned images with metadata



Everything is visible

- Programmer has full control over sensor settings
  - and access to the supplemental statistics from ISP
- No hidden daemon running autofocus/metering
  - nobody changes the settings under you



Implementations

HDR / Exposure Fusion on N900

Natasha Gelfand, Andrew Adams, Sung Hee Park, Kari Pulli
Multi-exposure Imaging on Mobile Devices
ACM Multimedia, Florence, Italy, October 25-29, 2010

Simple HDR Burst

```
#include <FCam/N900.h>
…
```

Simple HDR Burst

```
#include <FCam/N900.h>
...
Sensor sensor;
Shot   shortReq, midReq, longReq;
Frame  short, mid, long;
```

Simple HDR Burst

```
#include <FCam/N900.h>
...
Sensor sensor;
Shot   shortReq, midReq, longReq;
Frame  short, mid, long;

shortReq.exposure =  10000; // microseconds
midReq.exposure   =  40000;
longReq.exposure  = 160000;
shortReq.image = Image(sensor.maxImageSize(), RAW);
midReq.image   = Image(sensor.maxImageSize(), RAW);
longReq.image  = Image(sensor.maxImageSize(), RAW);
```

Simple HDR Burst

```
#include <FCam/N900.h>
...
Sensor sensor;
Shot   shortReq, midReq, longReq;
Frame  short, mid, long;

shortReq.exposure =  10000; // microseconds
midReq.exposure   =  40000;
longReq.exposure  = 160000;
shortReq.image = Image(sensor.maxImageSize(), RAW);
midReq.image   = Image(sensor.maxImageSize(), RAW);
longReq.image  = Image(sensor.maxImageSize(), RAW);

sensor.capture(shortReq);
sensor.capture(midReq);
sensor.capture(longReq);
```

Simple HDR Burst

```
#include <FCam/N900.h>
...
Sensor sensor;
Shot   shortReq, midReq, longReq;
Frame  short, mid, long;

shortReq.exposure =  10000; // microseconds
midReq.exposure   =  40000;
longReq.exposure  = 160000;
shortReq.image = Image(sensor.maxImageSize(), RAW);
midReq.image   = Image(sensor.maxImageSize(), RAW);
longReq.image  = Image(sensor.maxImageSize(), RAW);

sensor.capture(shortReq);
sensor.capture(midReq);
sensor.capture(longReq);

short = sensor.getFrame();
mid   = sensor.getFrame();
long  = sensor.getFrame();
```

HDR Viewfinder with metering

```
#include <FCam/N900.h>
...



...
while(1) {









}
```

HDR Viewfinder with metering

```
#include <FCam/N900.h>
...
vector<Shot> hdr(2);
hdr[0].exposure = 40000;
hdr[1].exposure = 10000;
...
while(1) {









}
```

HDR Viewfinder with metering

```
#include <FCam/N900.h>
...
vector<Shot> hdr(2);
hdr[0].exposure = 40000;
hdr[1].exposure = 10000;
...
while(1) {
  sensor.stream(hdr);




}
```

HDR Viewfinder with metering

```
#include <FCam/N900.h>
...
vector<Shot> hdr(2);
hdr[0].exposure = 40000;
hdr[1].exposure = 10000;
...
while(1) {
  sensor.stream(hdr);

  Frame longExp  = sensor.getFrame();
  Frame shortExp = sensor.getFrame();









}
```

HDR Viewfinder with metering

```
#include <FCam/N900.h>
...
vector<Shot> hdr(2);
hdr[0].exposure = 40000;
hdr[1].exposure = 10000;
...
while(1) {
  sensor.stream(hdr);

  Frame longExp  = sensor.getFrame();
  Frame shortExp = sensor.getFrame();

  hdr[0].exposure = autoExposeLong(longExp.histogram(),
                                   longExp.exposure());
  hdr[1].exposure = autoExposeShort(shortExp.histogram(),
                                    shortExp.exposure());


}
```

HDR Viewfinder with metering

```
#include <FCam/N900.h>
...
vector<Shot> hdr(2);
hdr[0].exposure = 40000;
hdr[1].exposure = 10000;
...
while(1) {
  sensor.stream(hdr);

  Frame longExp  = sensor.getFrame();
  Frame shortExp = sensor.getFrame();

  hdr[0].exposure = autoExposeLong(longExp.histogram(),
                                   longExp.exposure());
  hdr[1].exposure = autoExposeShort(shortExp.histogram(),
                                    shortExp.exposure());

  overlayWidget.display( blend(longExp, shortExp) );
}
```

Metering

- FCam provides a 30 fps stream of 640x480 frames with individually controlled exposure time and gain
- Alternate long and short exposures

1-10% of pixels bright (> 239)

1-10% of pixels dark (< 16)

Viewfinder

- Viewfinder preview of the HDR result @ 30 fps
- Show per-pixel average of the long and short exposures

Our viewfinder result has more detail than any single image

Single image with exposure at the geometric mean of inputs loses contrast

Firing the flash

```
...
Shot flashShot;
flashShot.exposure = 100000;  // 0.1 sec
...
```

Firing the flash

```
...
Shot flashShot;
flashShot.exposure = 100000;  // 0.1 sec
...
Flash flash;

Flash::FireAction fire(&flash);
```

Firing the flash

```
...
Shot flashShot;
flashShot.exposure = 100000;  // 0.1 sec
...
Flash flash;

Flash::FireAction fire(&flash);

fire.duration   = 1000; // 1 ms
fire.brightness = flash.maxBrightness();
fire.time       = flashShot.exposure - fire.duration;
```

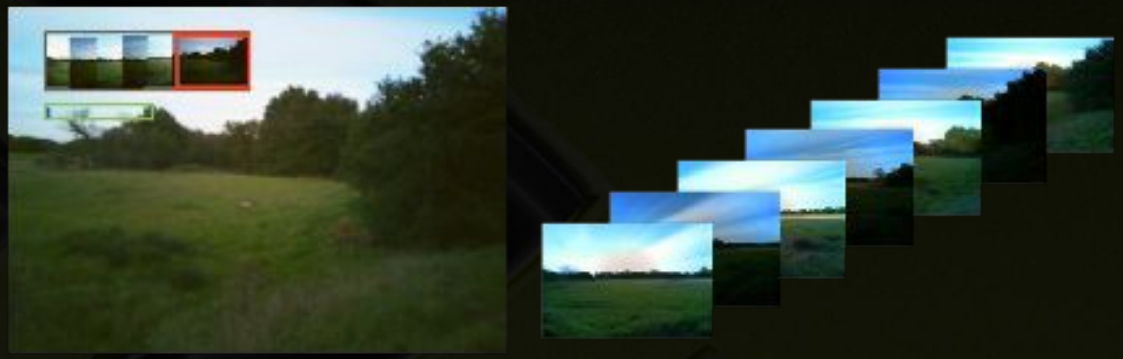
Firing the flash

```
...
Shot flashShot;
flashShot.exposure = 100000;  // 0.1 sec
...
Flash flash;

Flash::FireAction fire(&flash);

fire.duration   = 1000; // 1 ms
fire.brightness = flash.maxBrightness();
fire.time       = flashShot.exposure - fire.duration;

flashShot.addActions(fire);
sensor.capture(flashShot);
Frame flashFrame = sensor.getFrame();
```

Double flash

HDR Panorama Capture

- Alternates exposures to extend dynamic range



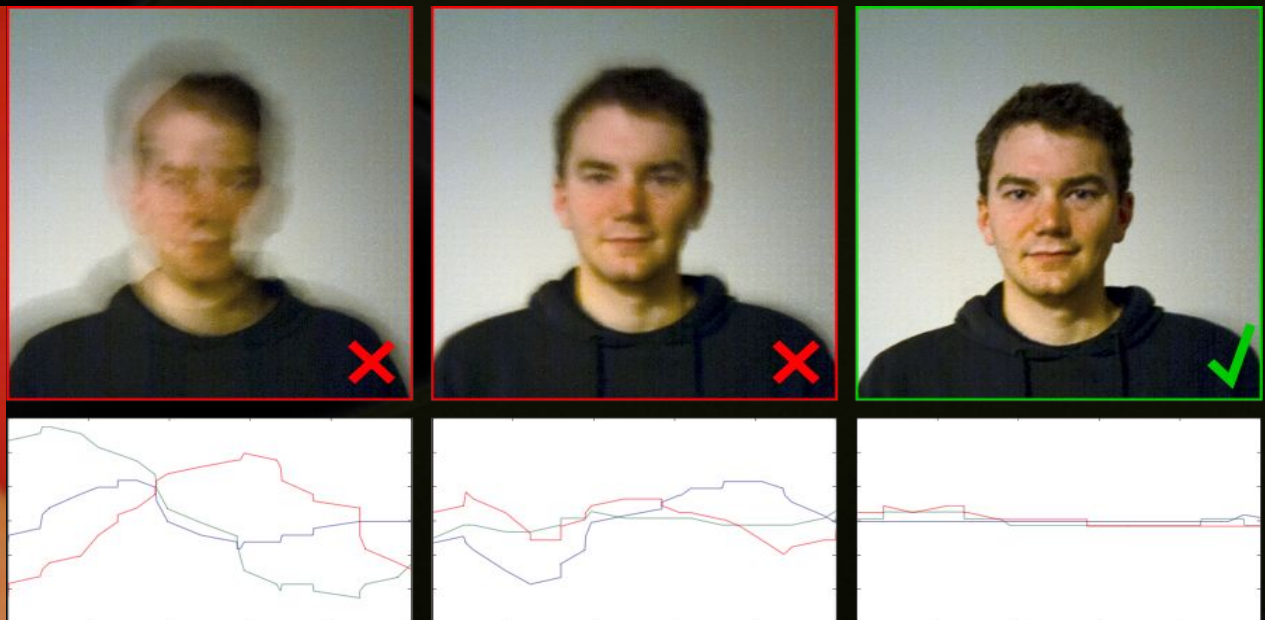
capture interface

individual images

extended dynamic range panorama

Lucky Imaging: Hand-held long exposures

- Holding camera steady for a long exposure is difficult
  - but sometimes you get lucky and hold it steady for a while
- We attached a 3-axis gyro to the N900
  - estimate if a captured image suffers from handshake
    - keep capturing if it does



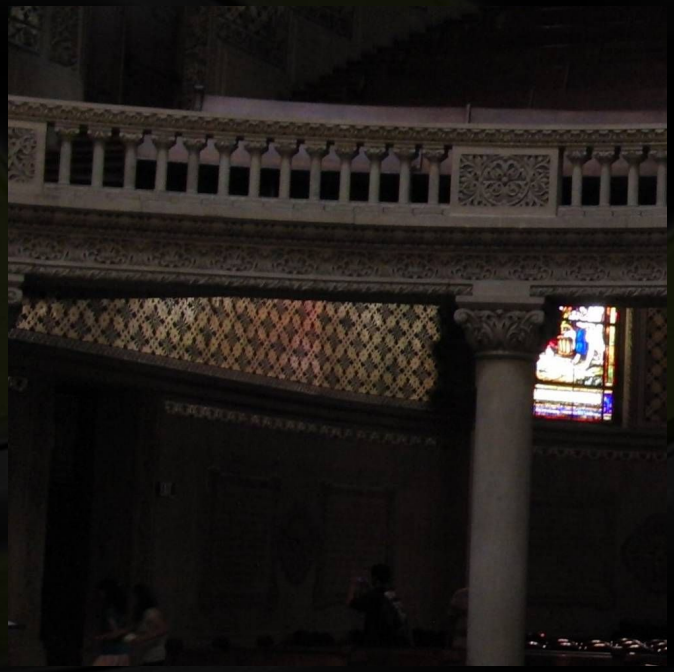
Low-light imaging on small cameras is challenging

- **Quality trade-offs**
  - **Increase ISO sensitivity: amplifies the noise**
  - **Increase exposure time: motion blur (hand-shake, objects)**

Marius Tico, Kari Pulli
Image Enhancement Method via Blur and Noisy Image Fusion
IEEE International Conference on Image Processing (ICIP'09)

Solution: Two images, combine the best aspects

Short exposure: dark, noisy, bad colors



Long exposure: good colors but blurry

NVIDIA Research

All-in-Focus Imaging

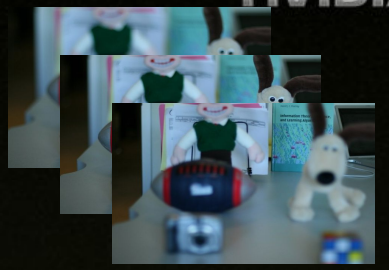
Images focused at different distances
(focal stack)

All-in-Focus Imaging

Contrast-based Passive Autofocus

Sharp!

Generalized Autofocus for focal stacks



- **Find the minimal set of images for an all-in-focus composite**
  - **The choice of this set depends on the scene**

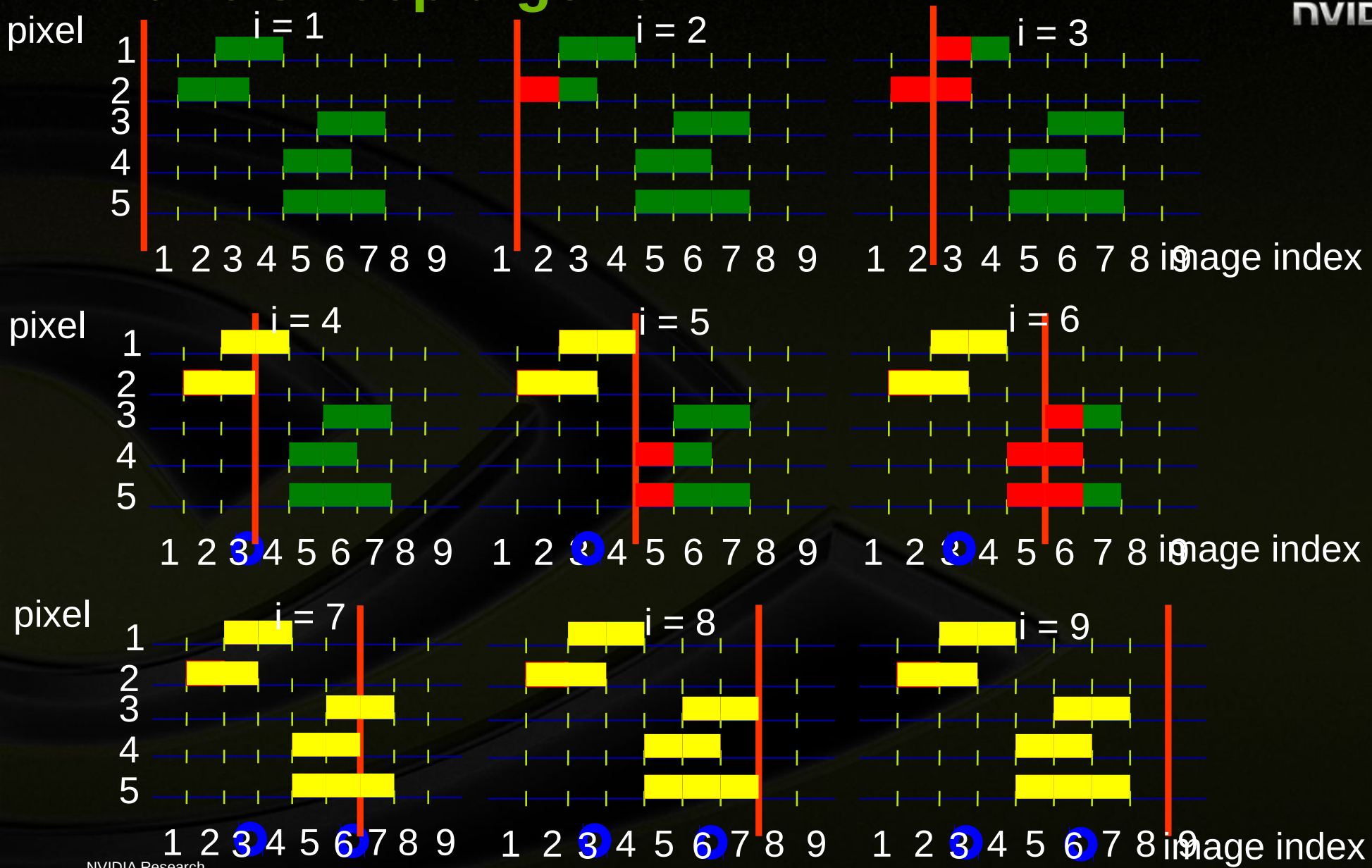
- **Minimizing the number of images is important**
  - **Faster capture**
  - **Less sensitive to motion**
  - **Requires less memory and processing power**

Daniel Vaquero, Natasha Gelfand, Marius Tico, Kari Pulli, Matthew Turk
Generalized Autofocus
IEEE Workshop on Applications of Computer Vision (WACV) 2011

Approach

1. Lens sweep from near focus to far focus
   - capture metering stack of low-resolution images
2. Analyze sharpness
3. Plane-sweep algorithm to determine the minimal set of required images, runs in linear time

Plane sweep algorithm



Approach

1. Lens sweep from near focus to far focus
   - capture metering stack of low-resolution images
2. Analyze sharpness
3. Plane-sweep algorithm to determine the minimal set of required images, runs in linear time
4. Recapture minimal set in high resolution
5. Perform all-in-focus fusion

Implementation

- Nokia N900 + FCam + enblend

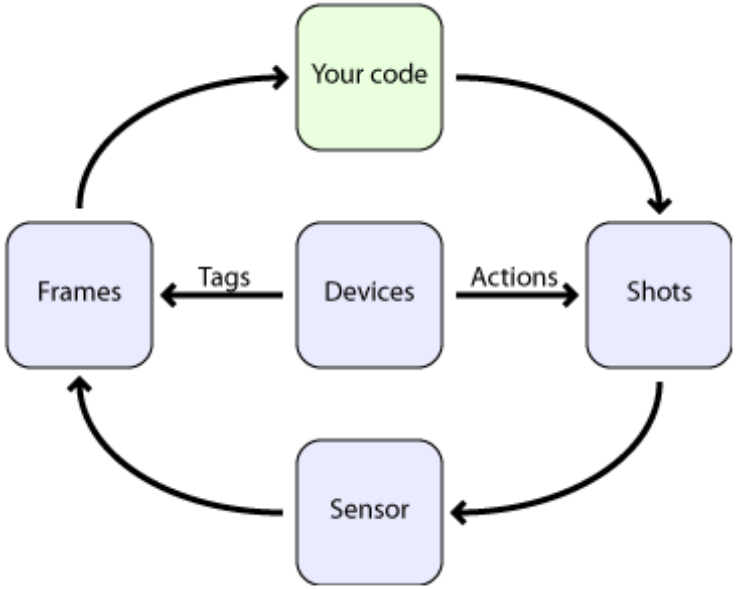
FCam: Open Source Project

http://fcam.garage.maemo.org/apiDocs.html

Main Page | Related Pages | Namespaces | Classes | Files

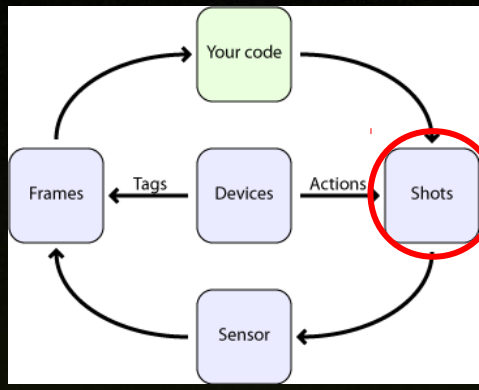
FCam: An API for controlling computational cameras.

The **FCam** API provides mechanisms to control various components of a camera to facilitate complex photographic applications.



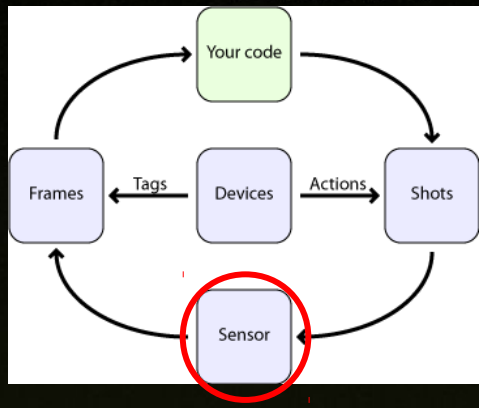
To use the **FCam**, you pass **Shots** to a **Sensor** which asynchronously returns **Frames**. A **Shot** completely specifies the capture and post-processing parameters of a single photograph, and a **Frame** contains the resulting image, along with supplemental hardware-generated statistics like a **Histogram** and **SharpnessMap**. You can tell **Devices** (like **Lenses** or **Flashes**) to schedule **Actions** (like **firing the flash**) to occur at some number of microseconds into a **Shot**. If timing is unimportant, you can also just tell **Devices** to do their thing directly from your code. In either case, **Devices** add tags to returned **Frames** (like the position of the **Lens** for that **Shot**). Tags are key-value pairs, where the key is a string like "focus" and the value is a **TagValue**, which can represent one of a number of types.

Shot specifies capture & post-process



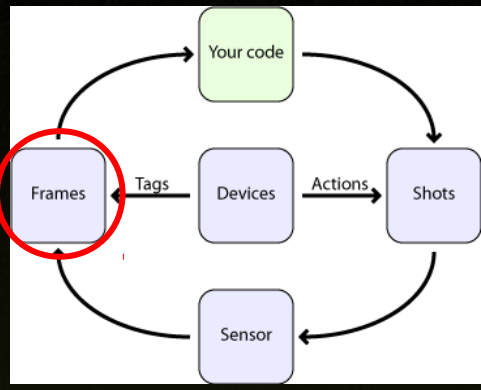
- Sensor parameters
  - analog gain (~= ISO)
  - exposure time (in microseconds)
  - total time (to set frame rate)
  - output resolution
  - format (raw or demosaicked [RGB, YUV])
  - white balance (only relevant if format is demosaicked)
  - memory location where to place the Image data
  - unique id (auto-generated on construction)
- Configures fixed-function statistics
  - region for Histogram
  - region and resolution for Sharpness Map

A Shot is passed to a Sensor



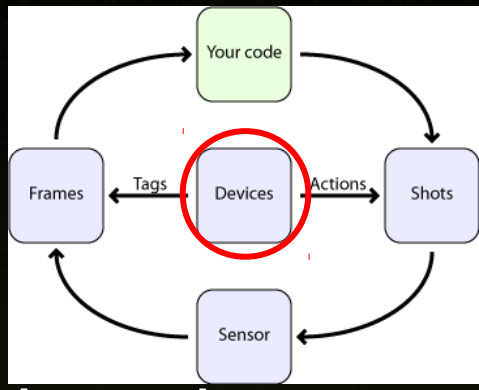
- Sensor manages a Shot queue in a separate thread
  - Sensor::capture()
    - just sticks a Shot on the end of the queue
  - Sensor::stream()
    - adds a copy of Shot to queue when the queue becomes empty
- Change the parameters of a streaming Shot
  - just alter it and call stream again with the updated Shot
- You can also specify a burst = vector of Shots
  - e.g., to capture quickly a full HDR stack, or for HDR viewfinder

Sensor produces Frames



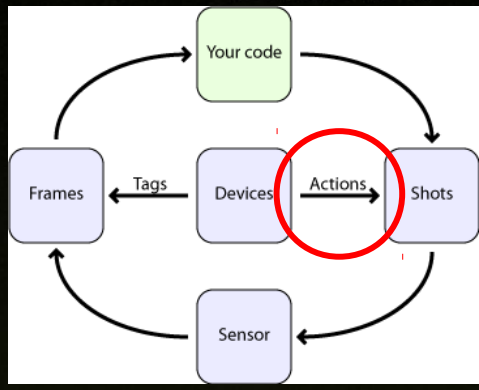
- Sensor::getFrame() is the only blocking call
- A Frame contains
  - image data and statistics
  - the precise time the exposure began and ended
  - the actual and requested (Shot) parameters
  - Tags from Devices (in Frame::tags() dictionary)
- Exactly one Frame for each Shot
  - If Image data is lost or corrupted
    - a Frame is still returned
      - with Image marked as invalid
      - statistics may be valid

Devices



- Lens
  - focus
    - measured in diopters: d * f = 1m
      - 20D => f = 5cm, 0D => f = inf
    - the lens starts moving (at specified speed) in the background
  - focal length (zooming factor) (fixed on N900)
  - aperture (fixed on N900)
- Flash
  - fire with a specified brightness and duration
- Other Devices can be created
  - FCam example 6 creates a Device for playing the click sound

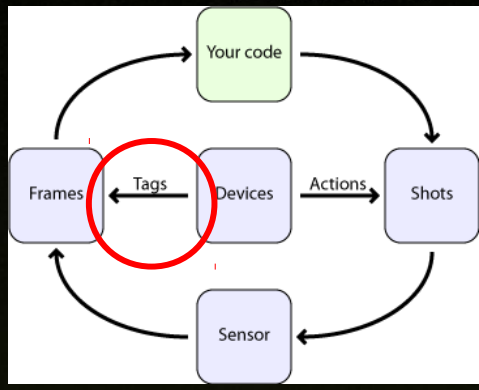
Actions allow Devices to coordinate



- Devices may have a set of Actions, with
  - start time w.r.t. image exposure start
  - Action::doAction() to initiate the action
  - a latency field
    - indicates the delay between the method call and the action begin

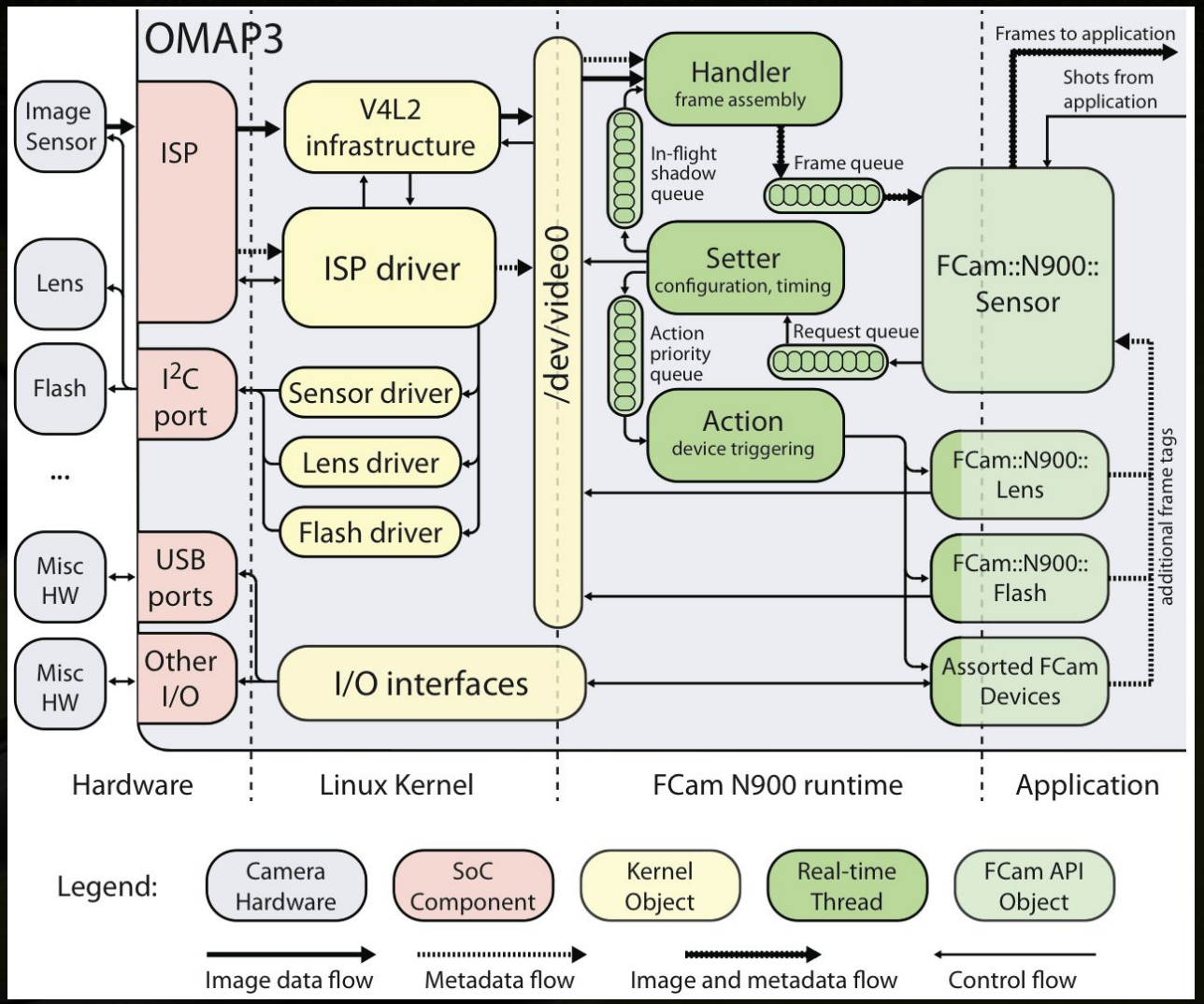
- Shots perform Actions during the exposure
  - with predictable latency Actions can be precisely scheduled
    - e.g., the timing of Flash in second-curtain sync must be accurate to within a millisecond

Tags



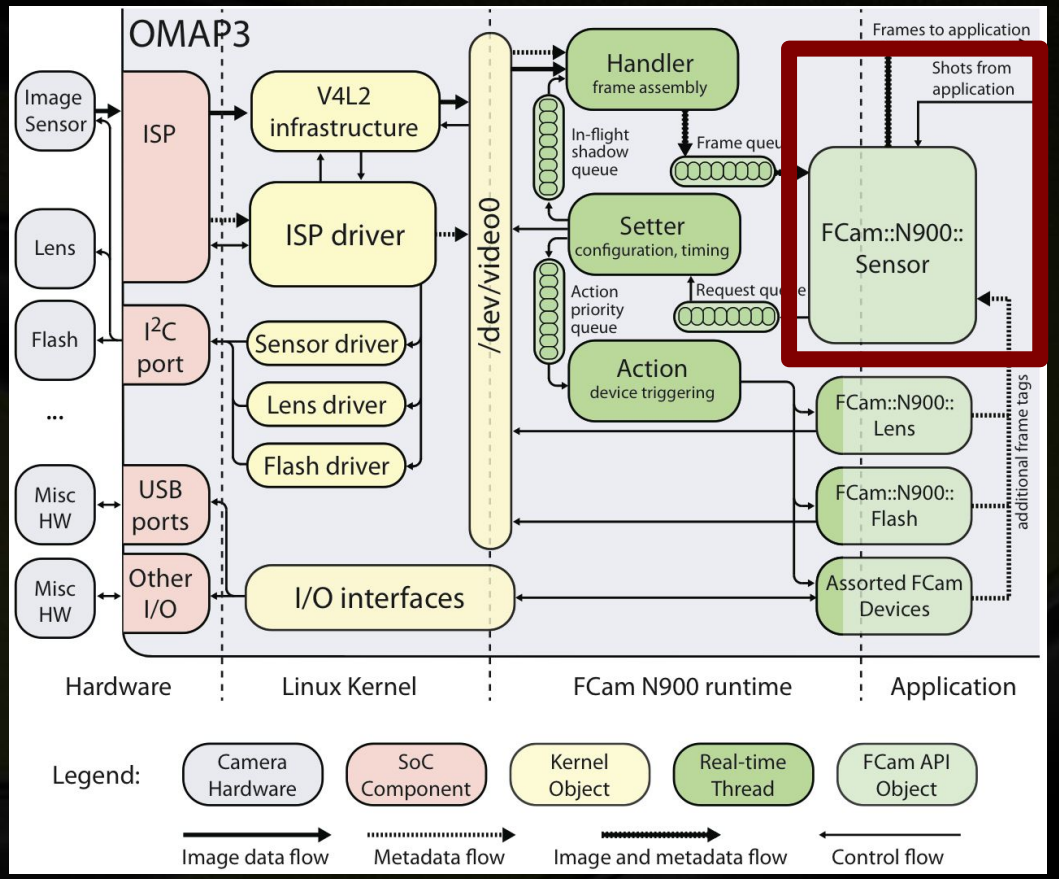
- Frames are tagged with metadata
  - after they leave the pipeline
  - Devices need to keep a short state history
    - match with time stamps
- Lens and Flash tag each Frame with their state
  - writing an autofocus algorithm becomes straightforward
    - the focus position of the Lens is known for each Frame
- Other appropriate uses of Tags
  - sensor fusion

N900 implementation of FCam



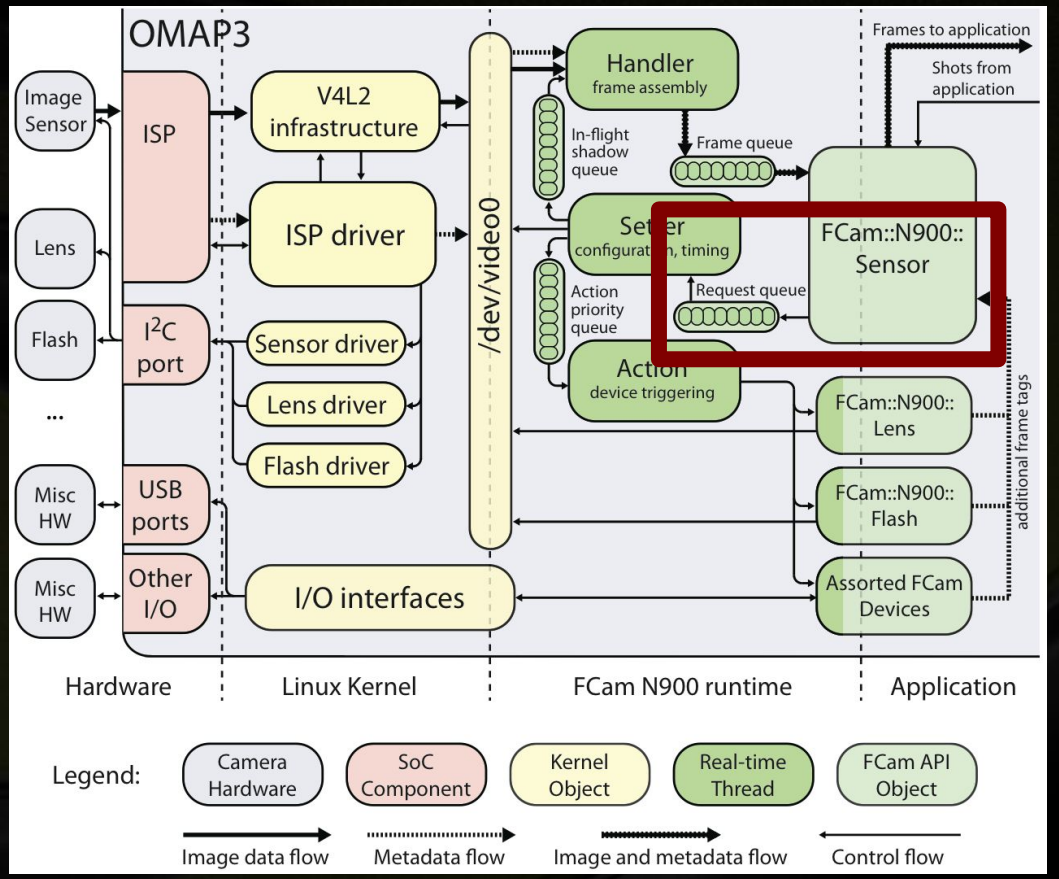
FCam image capture on N900 (simplified)

1. Request comes in from client

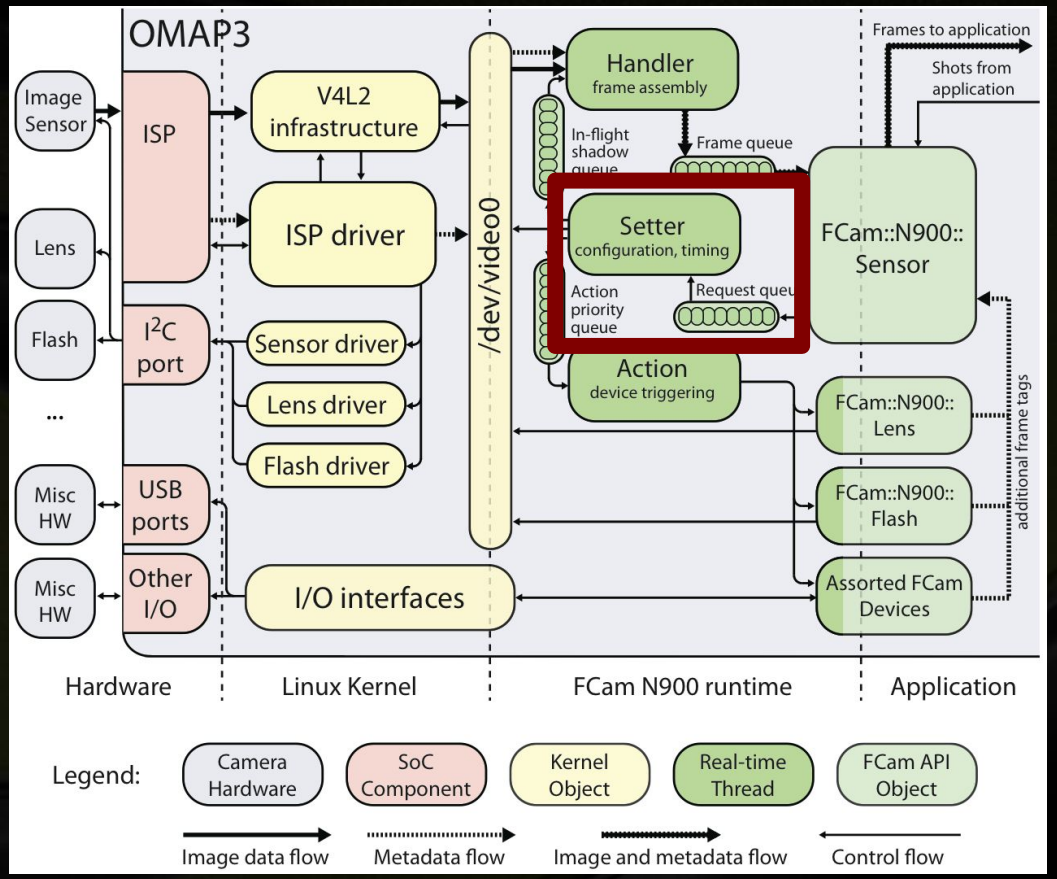


FCam image capture on N900 (simplified)

1. Request comes in from client
2. Request is put into request queue

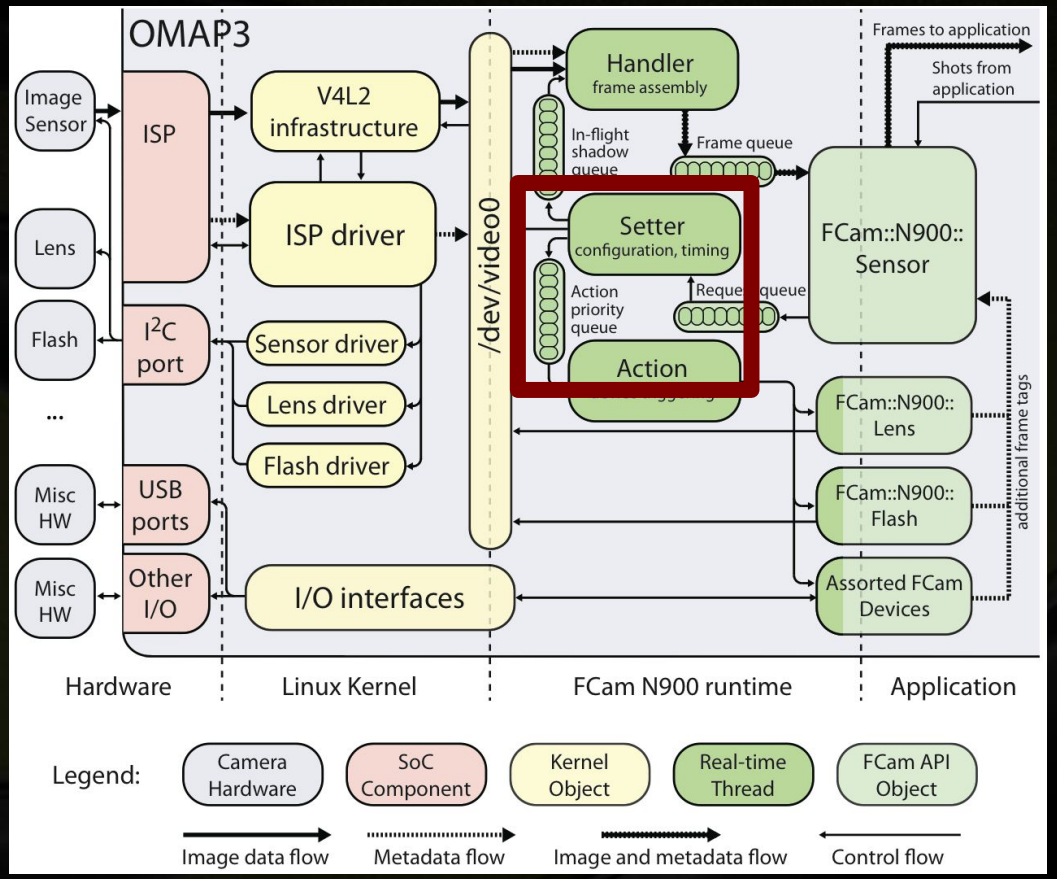


FCam image capture on N900 (simplified)



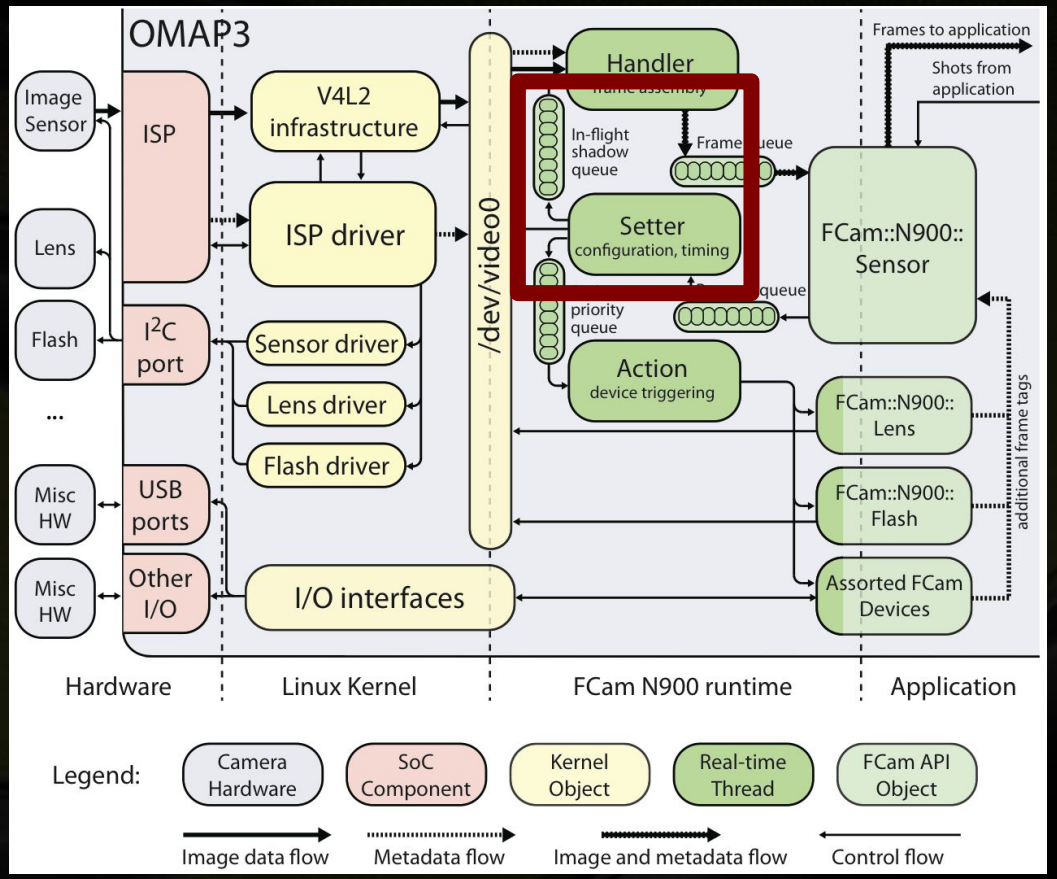
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue

FCam image capture on N900 (simplified)



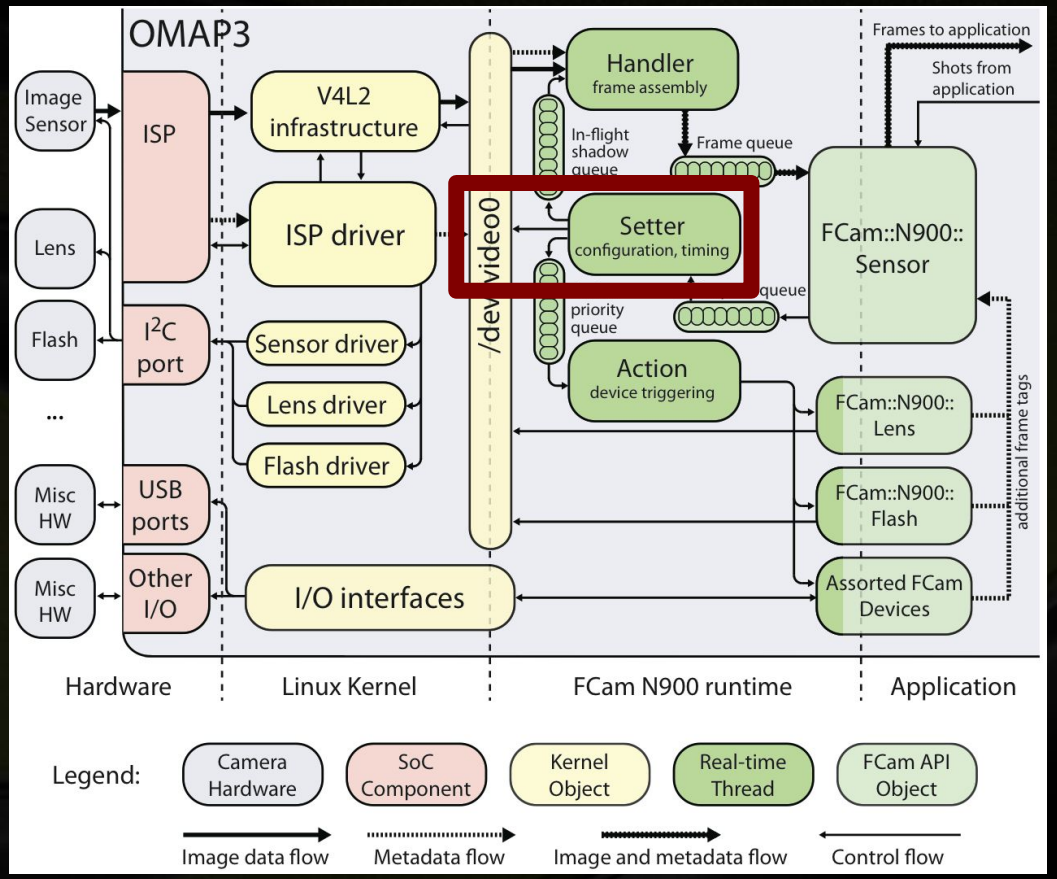
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue

FCam image capture on N900 (simplified)



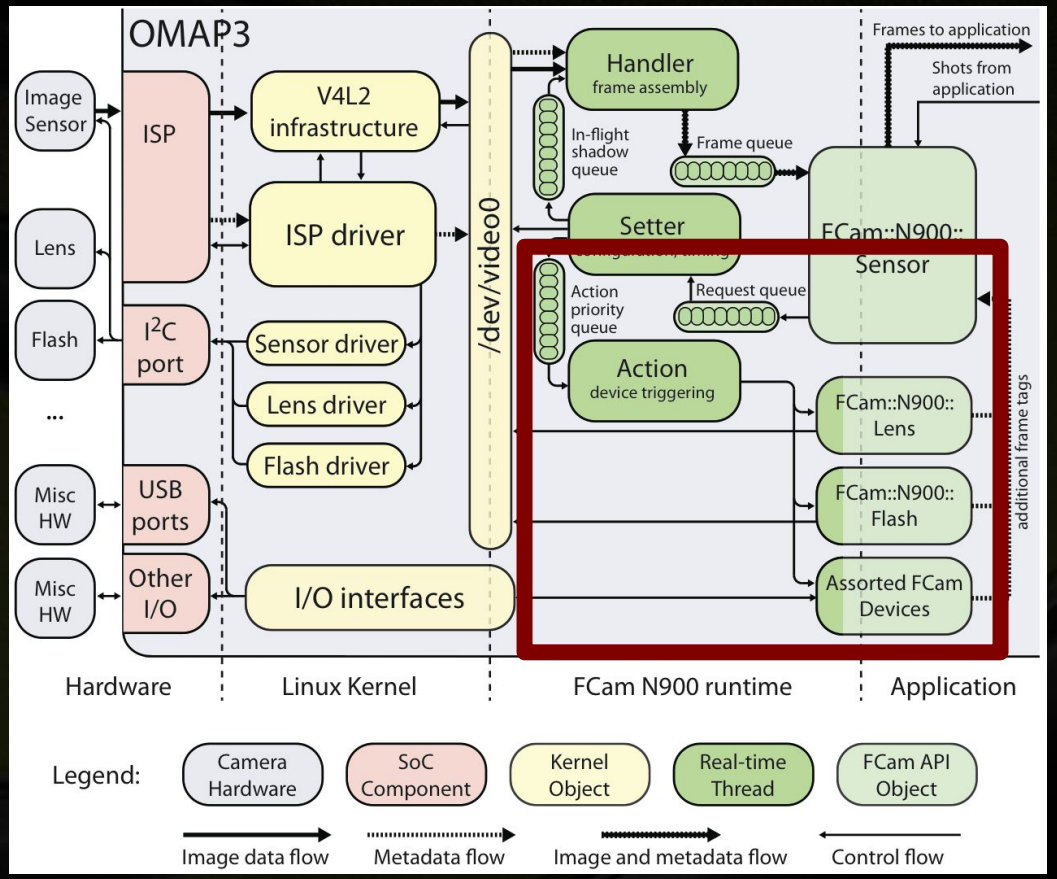
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue
5. Setter computes ETA for the image data from ISP and puts request info into in-flight shadow queue

FCam image capture on N900 (simplified)



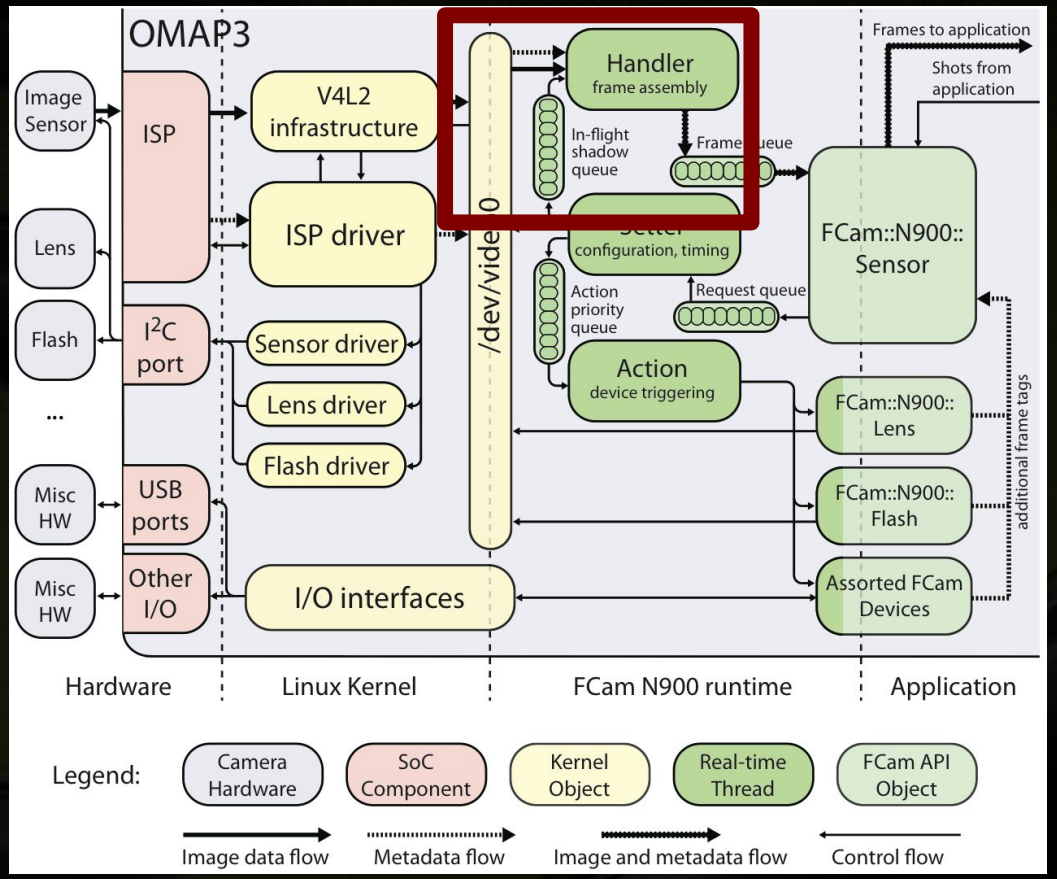
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue
5. Setter computes ETA for the image data from ISP and puts request info into in-flight shadow queue
6. Setter sets the sensor parameters according to the request

FCam image capture on N900 (simplified)



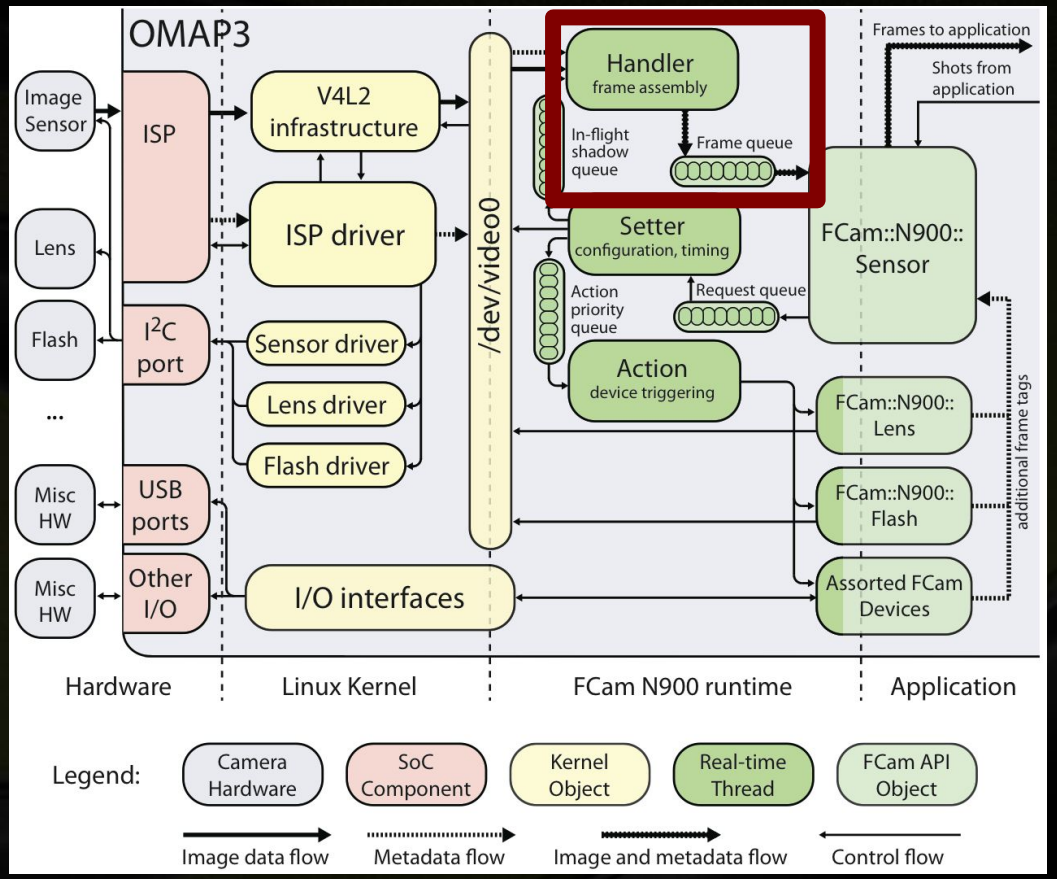
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue
5. Setter computes ETA for the image data from ISP and puts request info into in-flight shadow queue
6. Setter sets the sensor parameters according to the request
7. Actions are triggered from the action queue at correct time by the Action thread and handled by Devices

FCam image capture on N900 (simplified)



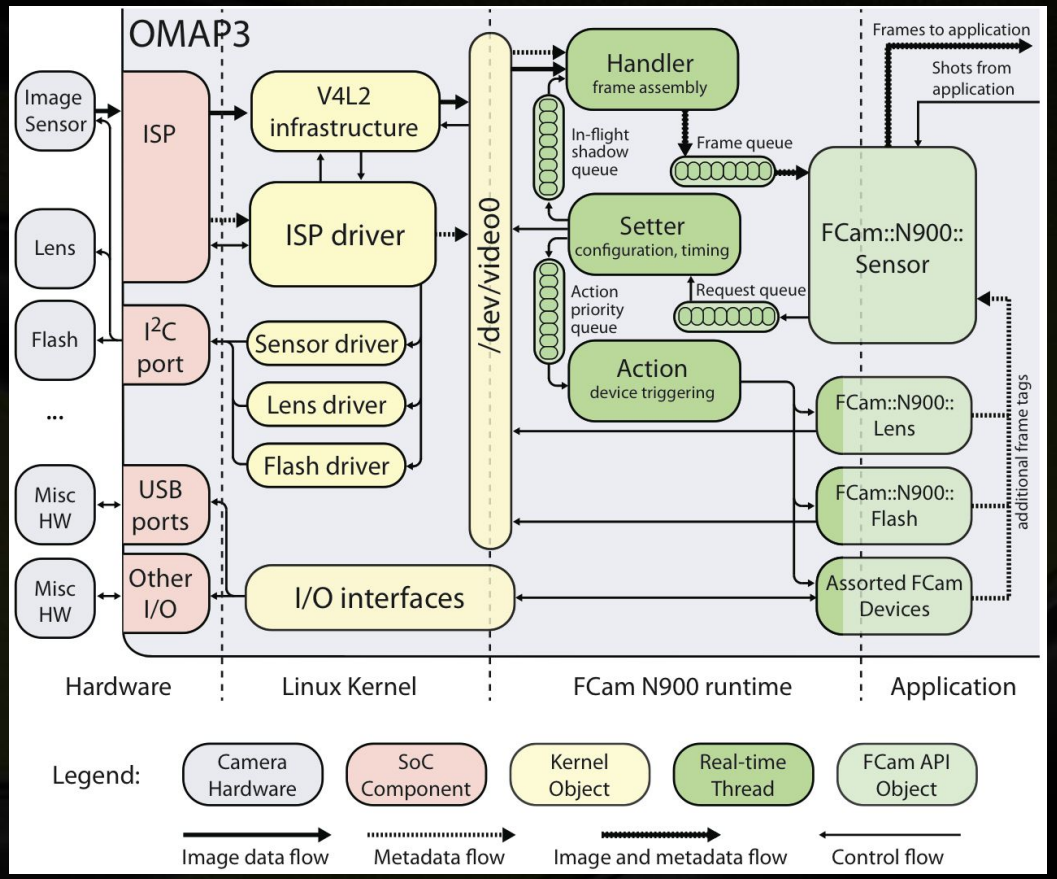
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue
5. Setter computes ETA for the image data from ISP and puts request info into in-flight shadow queue
6. Setter sets the sensor parameters according to the request
7. Actions are triggered from the action queue at correct time by the Action thread and handled by Devices
8. Handler thread reads incoming image data and metadata, connects them with the corresponding request in in-flight queue, and gets Tags from Devices

FCam image capture on N900 (simplified)



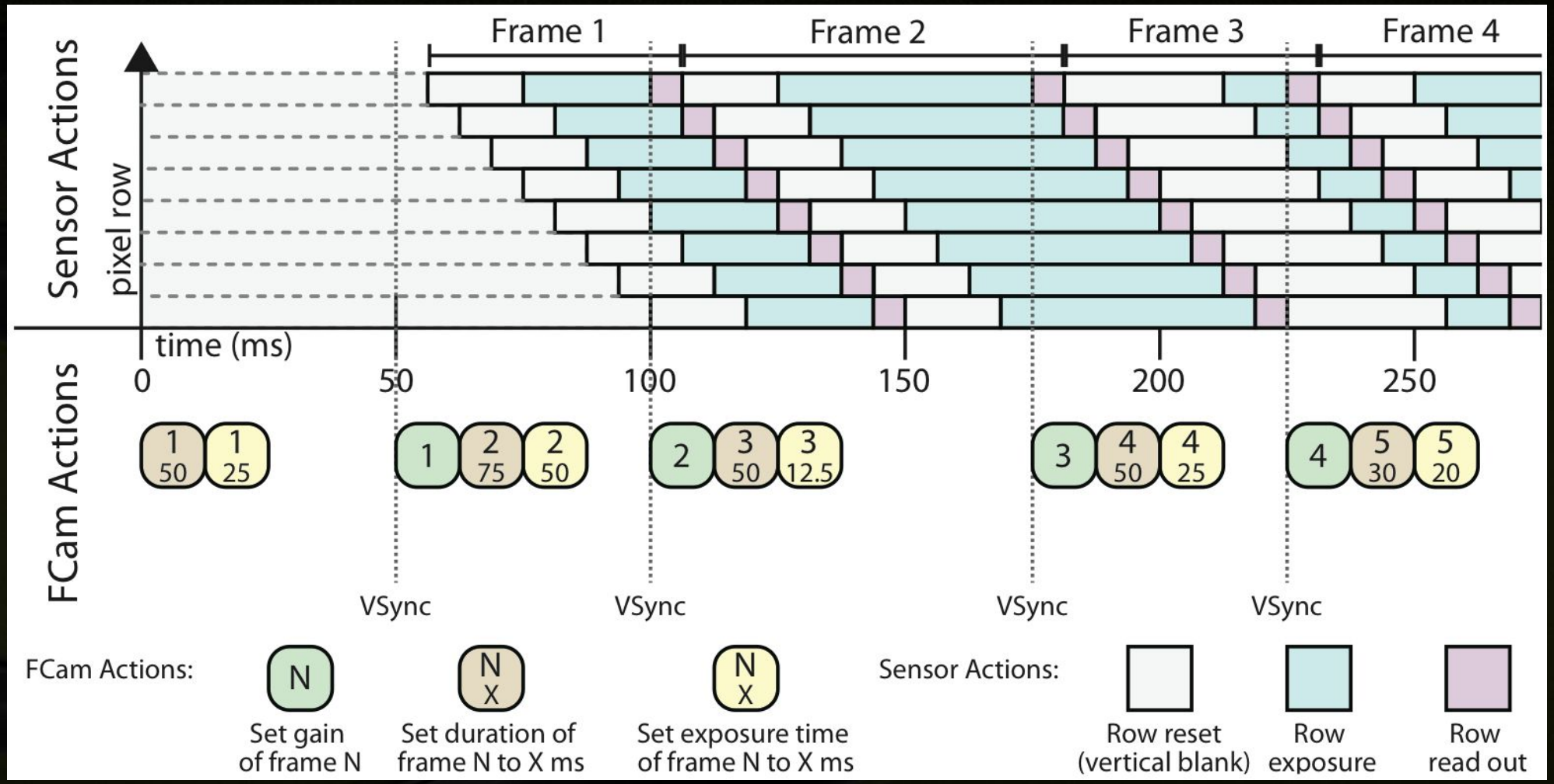
1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue
5. Setter computes ETA for the image data from ISP and puts request info into in-flight shadow queue
6. Setter sets the sensor parameters according to the request
7. Actions are triggered from the action queue at correct time by the Action thread and handled by Devices
8. Handler thread reads incoming image data and metadata, connects them with the corresponding request in in-flight queue, and gets Tags from Devices
9. Handler puts the assembled Frame object into Frame queue for client

FCam image capture on N900 (simplified)

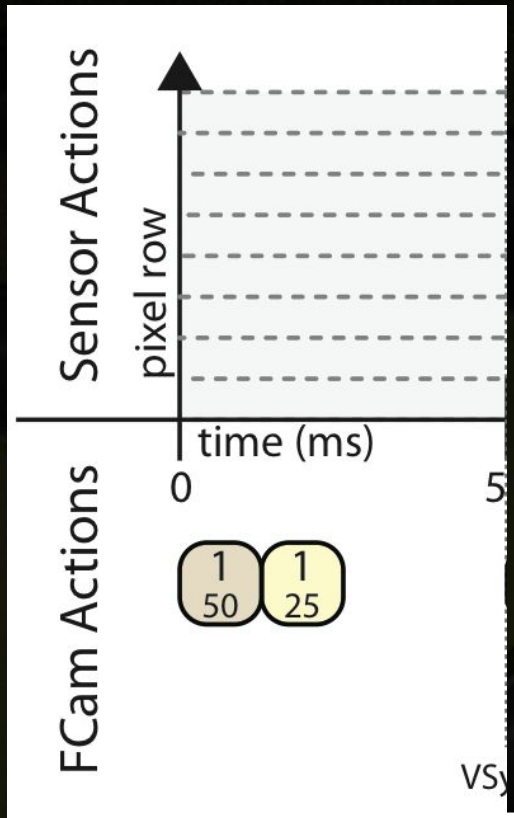


1. Request comes in from client
2. Request is put into request queue
3. Setter reads request from queue
4. Setter computes timing for possible actions and puts actions in queue
5. Setter computes ETA for the image data from ISP and puts request info into in-flight shadow queue
6. Setter sets the sensor parameters according to the request
7. Actions are triggered from the action queue at correct time by the Action thread and handled by Devices
8. Handler thread reads incoming image data and metadata, connects them with the corresponding request in in-flight queue, and gets Tags from Devices
9. Handler puts the assembled Frame object into Frame queue for client

N900 implementation of FCam



N900 implementation of FCam



FCam Actions:

N — Set gain of frame N

N / X — Set duration of frame N to X ms

N / X — Set exposure time of frame N to X ms

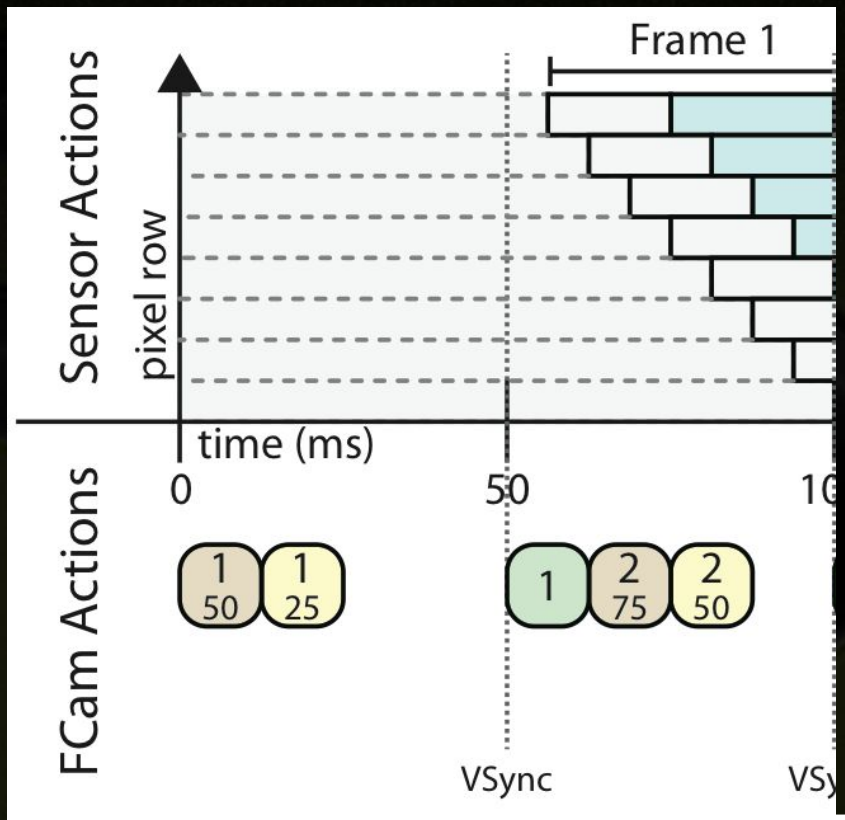
Sensor Actions:

Row reset (vertical blank)

Row exposure

Row read out

N900 implementation of FCam



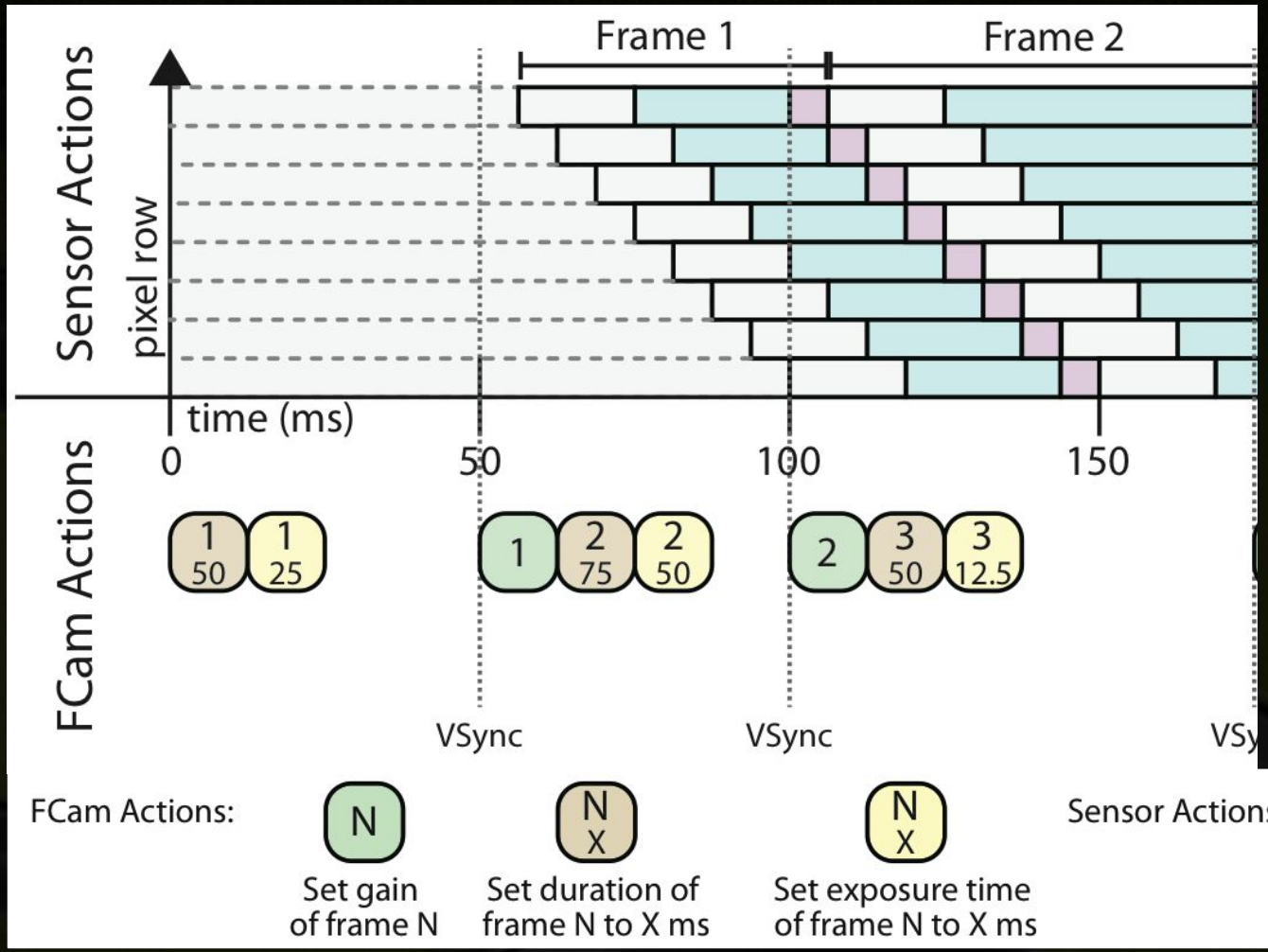
FCam Actions:

- N — Set gain of frame N
- N X — Set duration of frame N to X ms
- N X — Set exposure time of frame N to X ms

Sensor Actions:

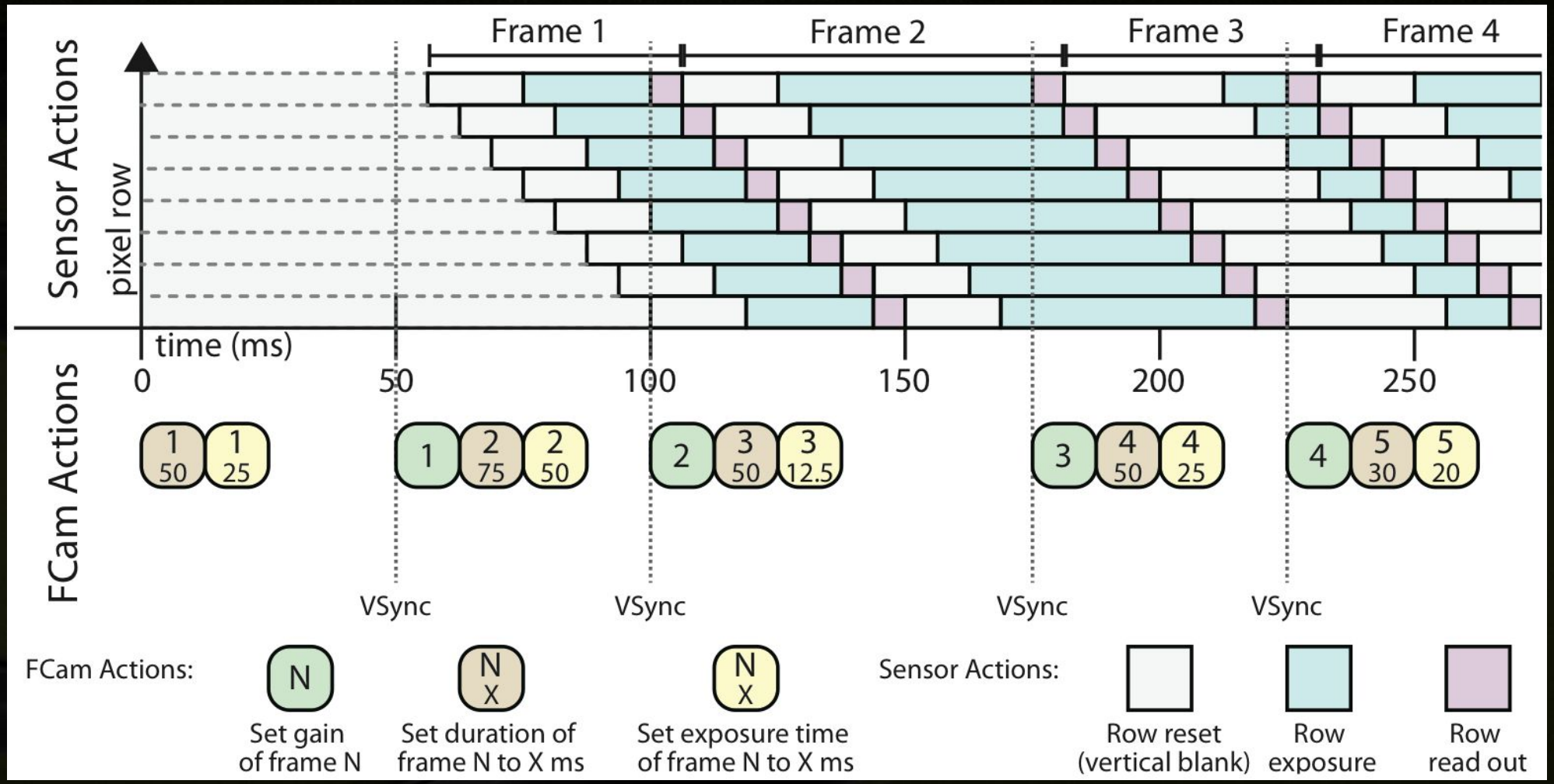
- Row reset (vertical blank)
- Row exposure
- Row read out

N900 implementation of FCam



50 ms

N900 implementation of FCam



Ex 1: Take a photo (Sensor, Shot, Frame)


```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/* Macros for Android logging
 * Use: adb logcat FCAM_EXAMPLE1:*  *:S
 * to see all messages from this app
 */
#if !defined(LOG_TAG)
#define LOG_TAG "FCAM_EXAMPLE1"
#define LOGV(...) __android_log_print(ANDROID_LOG_VERBOSE, LOG_TAG, __VA_ARGS__)
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG  , LOG_TAG, __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO   , LOG_TAG, __VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN   , LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR  , LOG_TAG, __VA_ARGS__)
#endif
#include <android/log.h>

#include <FCam/Tegra.h>
namespace Plat = FCam::Tegra;
const int width  = 2560;
const int height = 1920;
const FCam::ImageFormat format = FCam::YUV420p;
const char output_image[] = "/data/fcam/example1.jpg";

/** \file */

/************************************************************/
/* A program that takes a single shot                       */
/*                                                          */
/* This example is a simple demonstration of the usage of   */
/* the FCam API.                                            */
/************************************************************/

void errorCheck();

int main(int argc, char ** argv) {

    // Make the image sensor
    Plat::Sensor sensor;

    // Make a new shot
    Plat::Shot shot1;
    shot1.exposure = 50000; // 50 ms exposure
    shot1.gain = 1.0f;      // minimum ISO

    // Specify the output resolution and format, and allocate storage for the resulting image
    shot1.image = FCam::Image(width, height, format);

    // Order the sensor to capture a shot
    sensor.capture(shot1);

    // Check for errors
    errorCheck();

    assert(sensor.shotsPending() == 1); // There should be exactly one shot

    // Retrieve the frame from the sensor
    Plat::Frame frame = sensor.getFrame();

    // This frame should be the result of the shot we made
    assert(frame.shot().id == shot1.id);

    // This frame should be valid too
    assert(frame.valid());
    assert(frame.image().valid());

    // Save the frame
    FCam::saveJPEG(frame, output_image);

    // Check that the pipeline is empty
    assert(sensor.framesPending() == 0);
    assert(sensor.shotsPending() == 0);

    return 0;
}

void errorCheck() {
    // Make sure FCam is running properly by looking for DriverError
    FCam::Event e;
    while (FCam::getNextEvent(&e, FCam::Event::Error) ) {
        printf("Error: %s\n", e.description.c_str());
        if (e.data == FCam::Event::DriverMissingError) {
            printf("example1: FCam can't find its driver. Did you install "
                   "fcam-drivers on your platform, and reboot the device "
                   "after installation?\n");
            exit(1);
        }
        if (e.data == FCam::Event::DriverLockedError) {
            printf("example1: Another FCam program appears to be running "
                   "already. Only one can run at a time.\n");
            exit(1);
        }
    }
}
```

Ex 2: Flash/No-flash (Device, Flash Action)

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <FCam/Tegra.h>

/* Macros for Android logging
 * Use: adb logcat FCAM_EXAMPLE2:*  *:S
 * to see all messages from this app
 */
#if !defined(LOG_TAG)
#define LOG_TAG "FCAM_EXAMPLE2"
#define LOGV(...) __android_log_print(ANDROID_LOG_VERBOSE, LOG_TAG, __VA_ARGS__)
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG  , LOG_TAG, __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO   , LOG_TAG, __VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN   , LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR  , LOG_TAG, __VA_ARGS__)
#endif
#include <android/log.h>

namespace Plat = FCam::Tegra;
const int width  = 2560;
const int height = 1920;
const FCam::ImageFormat format = FCam::YUV420p;
const char output_image_flash[] = "/data/fcam/example2_flash.jpg";
const char output_image_noflash[] = "/data/fcam/example2_noflash.jpg";

/**********************************************************/
/* Flash / No-flash                                       */
/*                                                        */
/* This example demonstrates capturing multiple shots with */
/* possibly different settings.                           */
/**********************************************************/

int main(int argc, char ** argv) {

        // Devices
    Plat::Sensor sensor;
    Plat::Flash flash;
    sensor.attach(&flash); // Attach the flash to the sensor

    // Make two shots
    std::vector<Plat::Shot> shot(2);

    // Set the first shot parameters (to be done with flash)
    shot[0].exposure = 10000;
    shot[0].gain = 1.0f;
    shot[0].image = FCam::Image(width, height, format);

    // Set the second shot parameters (to be done without flash)
    shot[1].exposure = 10000;
    shot[1].gain = 1.0f;
    shot[1].image = FCam::Image(width, height, format);

    // Make an action to fire the flash
    Plat::Flash::FireAction fire(&flash);

    // Flash on must be triggered at time 0 - duration is ignored.
    fire.duration = flash.minDuration();
    fire.time = 0;                              // at the start of the exposure
    fire.brightness = flash.maxBrightness();    // at full power

    // Attach the action to the first shot
    shot[0].addAction(fire);

    // Order the sensor to capture the two shots
    sensor.capture(shot);
    assert(sensor.shotsPending() == 2);    // There should be exactly two shots

    // Retrieve the first frame
    Plat::Frame frame = sensor.getFrame();
    assert(sensor.shotsPending() == 1);    // There should be one shot pending
    assert(frame.shot().id == shot[0].id); // Check the source of the request

    FCam::Flash::Tags tags(frame);
    LOGD("First frame flash settings:\n");
    LOGD(" brightness %f, start %d, duration %d, peak %d\n",
                tags.brightness, tags.start, tags.duration, tags.peak);

        // Write out file if needed
    FCam::saveJPEG(frame, output_image_flash);

    // Retrieve the second frame
    frame = sensor.getFrame();
    assert(frame.shot().id == shot[1].id); // Check the source of the request

    tags = FCam::Flash::Tags(frame);
    LOGD("Second frame flash settings:\n");
    LOGD(" brightness %f, start %d, duration %d, peak %d\n",
                tags.brightness, tags.start, tags.duration, tags.peak);

    // Write out file
    FCam::saveJPEG(frame, output_image_noflash);

    // Check the pipeline is empty
    assert(sensor.framesPending() == 0);
    assert(sensor.shotsPending() == 0);
}
```

Example 3

http://fcam.garage.maemo.org/examples.html

- Capture a photograph while the focus ramps from near to far
- Demonstrates
  - Device
  - Lens Action
  - Tag

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <FCam/N900.h>

// Select the platform
namespace Plat = FCam::N900;
// namespace Plat = FCam::F2;

/************************************************************/
/* Focus sweep                                              */
/*                                                          */
/* This example demonstrates moving the lens during an      */
/* exposure via the use of Lens::FocusAction. It also       */
/* shows how to use the metadata tagged by the devices.     */
/* Because the lens on the N900 zooms slightly when it      */
/* focuses, you'll also get a zoom-blur effect.             */
/************************************************************/
int main(int argc, char ** argv) {

    // Devices
    Plat::Sensor sensor;
    Plat::Lens lens;

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

    // First focus near with maximal speed
    lens.setFocus(lens.nearFocus(), lens.maxFocusSpeed());
    while (lens.focusChanging()); // Wait to be done


    // Now make a shot that will sweep the lens
    FCam::Shot shot1;

    FCam::Lens::FocusAction sweep(&lens);
    // Set the parameters of this action
    sweep.time = 0;
    sweep.focus = lens.farFocus();
    sweep.speed = lens.maxFocusSpeed()/4;
    // Calculate how long it takes to move the lens to the desired
    // location in microseconds
    float duration = 1000000.0f *
        (lens.nearFocus() - lens.farFocus()) / sweep.speed;

    printf("The lens will sweep from near to far in %f milliseconds\n",
           duration / 1000.0);

    // Set the shot parameter accordingly
    shot1.exposure = duration;
    shot1.gain = 1.0f;
    // Use a lower resolution to minimize rolling shutter effects
    shot1.image = FCam::Image(640, 480, FCam::UYVY);

    // Attach the action to the shot
    shot1.addAction(sweep);

    // Order the sensor to capture the shot
    sensor.capture(shot1);
    assert(sensor.shotsPending() == 1); // There should be exactly one shot

    // Retrieve the frame
    FCam::Frame frame = sensor.getFrame();
    assert(frame.shot().id == shot1.id); // Check the source of the request

    // Print out some metadata
    const FCam::Lens::Tags lensTags(frame);
    printf("Aperture        : %.4f\n", lensTags.aperture);
    printf("Initial focus   : %.4f\n", lensTags.initialFocus);
    printf("Final focus     : %.4f\n", lensTags.finalFocus);

    // Save the resulting file
    FCam::saveJPEG(frame, "/home/user/MyDocs/DCIM/example3.jpg");

    assert(sensor.framesPending() == 0);
    assert(sensor.shotsPending() == 0);
}
```

Ex 4: Streaming (autoExpose, AWB)


```cpp
#include <FCam/Tegra.h>
#include <FCam/AutoExposure.h>
#include <FCam/AutoWhiteBalance.h>

namespace Plat = FCam::Tegra;
const int width  = 640;
const int height = 480;
const FCam::ImageFormat format = FCam::YUV420p;
const char output_image[] = "/data/fcam/example4.jpg";


/************************************************************/
/* Autoexposure                                             */
/*                                                          */
/* This example shows how to request streams and deal with */
/* the incoming frames, and also uses the provided          */
/* auto-exposure and auto-white-balance routines.           */
/************************************************************/
int main(int argc, char ** argv) {

    // Make a sensor
    Plat::Sensor sensor;

    // Shot
    Plat::Shot stream1;
    // Set the shot parameters
    stream1.exposure = 33333;
    stream1.gain = 1.0f;

    // We don't bother to set frameTime in this example. It defaults
    // to zero, which the implementation will clamp to the minimum
    // possible value given the exposure time.

    // Request an image size and allocate storage
    stream1.image = FCam::Image(width, height, format);

    // Enable the histogram unit
    stream1.histogram.enabled = true;
    stream1.histogram.region = FCam::Rect(0, 0, width, height);

    // We will stream until the exposure stabilizes
    int count = 0;          // # of frames streamed
    int stableCount = 0;    // # of consecutive frames with stable exposure
    float exposure;         // total exposure for the current frame (exposure time * gain)
    float lastExposure = 0; // total exposure for the previous frame

    Plat::Frame frame;

    do {
        // Ask the sensor to stream with the given parameters
        sensor.stream(stream1);

        // Retrieve a frame
        frame = sensor.getFrame();
        assert(frame.shot().id == stream1.id); // Check the source of the request

        fcam_print("Exposure time: %d, gain: %f\n", frame.exposure(), frame.gain());

        // Calculate the total exposure used (including gain)
        exposure = frame.exposure() * frame.gain();

        // Call the autoexposure algorithm. It will update stream1
        // using this frame's histogram.
        autoExpose(&stream1, frame);

        // Call the auto white-balance algorithm. It will similarly
        // update the white balance using the histogram.
        autoWhiteBalance(&stream1, frame);

        // Increment stableCount if the exposure is within 5% of the
        // previous one
        if (fabs(exposure - lastExposure) < 0.05f * lastExposure) {
            stableCount++;
        } else {
            stableCount = 0;
        }

        // Update lastExposure
        lastExposure = exposure;

    } while (stableCount < 5); // Terminate when stable for 5 frames

    // Write out the well-exposed frame
    FCam::saveJPEG(frame, output_image);

    // Order the sensor to stop streaming
    sensor.stopStreaming();
    printf("Processed %d frames until stable for 5 frames!\n", count);

    // There may still be shots in the pipeline. Consume them.
    while (sensor.shotsPending() > 0) frame = sensor.getFrame();

    // Check that the pipeline is empty
    assert(sensor.framesPending() == 0);
    assert(sensor.shotsPending() == 0);
}
```

Ex 5: Streaming (AutoFocus)


```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <FCam/AutoFocus.h>
#include <FCam/Tegra.h>
#include <FCam/Tegra/AutoFocus.h>

/************************************************************/
/* Autofocus                                                */
/*                                                          */
/* This example shows how to request streams and deal with */
/* the incoming frames, and also uses the provided          */
/* autofocus routine.                                       */
/************************************************************/
int main(int argc, char ** argv) {

    // Devices
    FCam::Tegra::Sensor sensor;
    FCam::Tegra::Lens lens;
    sensor.attach(&lens); // Attach the lens to the sensor

    // Autofocus supplied by FCam API
    FCam::Tegra::AutoFocus autoFocus(&lens);

    // Shot
    FCam::Tegra::Shot stream1;
    // Set the shot parameters
    stream1.exposure = 50000;
    stream1.gain = 1.0f;

    // Request a resolution, and allocate storage
    stream1.image = FCam::Image(640, 480, FCam::YUV420p);

    // Enable the sharpness unit
    stream1.sharpness.enabled = true;

    // We will stream until the focus stabilizes
    int count = 0;        // # of frames streamed

    // Ask the autofocus algorithm to start sweeping the lens
    autoFocus.startSweep();

    // Stream until autofocus algorithm completes
    FCam::Frame frame;
    do {
        // Stream the updated shot
        sensor.stream(stream1);

        // Retrieve a frame
        frame = sensor.getFrame();
        assert(frame.shot().id == stream1.id); // Check the source of the request

        // The lens has tagged each frame with where it was focused
        // during that frame. Let's retrieve it so we can print it out.
        float diopters = frame["lens.focus"];
        fcam_print("Lens focused at %2.0f cm\n", 100/diopters);

        // The sensor has attached a sharpness map to each frame.
        // Let's sum up all the values in it so we can print out
        // the total sharpness of this frame.
        int totalSharpness = 0;
        for (int y = 0; y < frame.sharpness().height(); y++) {
            for (int x = 0; x < frame.sharpness().width(); x++) {
                totalSharpness += frame.sharpness()(x, y);
            }
        }
        fcam_print("Total sharpness is %d\n\n", totalSharpness);

        // Call the autofocus algorithm
                // Pass the shot too, it can be updated with
                // a focus stepping action.
        autoFocus.update(frame, &stream1);

        // Increment frame counter
        count++;
    } while (!autoFocus.idle());

    fcam_print("Autofocus chose to focus at %2.0f cm\n\n", 100/lens.getFocus());

    // Write out the focused frame
    FCam::saveJPEG(frame, "/data/fcam/example5.jpg");

    // Order the sensor to stop streaming
    sensor.stopStreaming();
    fcam_print("Processed %d frames until autofocus completed!\n", count);

    // There may still be shots in the pipeline. Consume them.
    while (sensor.shotsPending() > 0) frame = sensor.getFrame();

    // Check that the pipeline is empty
    assert(sensor.framesPending() == 0);
    assert(sensor.shotsPending() == 0);
}
```

Adding functionality: Devices

- Device is a base class for anything that adds Tags to Frames
- Derived classes must implement Device::tagFrame(Frame)
- Client application attaches Device to Sensor

```
class Kilroy : public FCam::Device {
public:
    void tagFrame(FCam::Frame f) {
        f[”mytag"] = std::string("Kilroy was here");
    }
};
/* ... */
FCam::N900::Sensor sensor;
Kilroy kilroy;
sensor.attach(&kilroy);
```

Adding functionality: Actions

- Action is a base class for anything synchronized to image exposure start
  - Flash fire, for example
- Derived classes must implement Action::doAction( )
  - actual action is performed here
  - doAction should return quickly, e.g., waking a QWaitCondition
- doAction is called at (exposure start time + time - latency)
  - Action should know its own latency (unit: microseconds)
- Client application adds Action objects to Shots
  - client sets Action firing time relative to exposure

FCamera SoundPlayer (simplified)


```
class SoundPlayer : public FCam::Device {
public:
    class SoundAction : public FCam::CopyableAction<SoundAction> {
    public:
        void doAction() { player->beep(); }

    protected:
        SoundPlayer * player;
    };

    void tagFrame(FCam::Frame) { /* No tags to add */ }

    /* Play a noise */
    void beep();
    /* Returns latency in microseconds */
    int getLatency();
};
```

Action::latency is set in SoundAction constructor

Implementation of Action::doAction must be fast, here player->beep() releases a semaphore which causes sound to play in another thread

FCamera SoundPlayer (simplified)

```
class SoundPlayer : public FCam::Device {
public:
    class SoundAction : public FCam::CopyableAction<SoundAction> {
    public:
        void doAction() { player->beep(); }

    protected:
        SoundPlayer * player;
    };

    void tagFrame(FCam::Frame) { /* No tags to add */ }

    /* Play a noise */
    void beep();
    /* Returns latency in microseconds */
    int getLatency();
};
```

Action::latency is set in SoundAction constructor

Implementation of Action::doAction must be fast, here player->beep() releases a semaphore which causes sound to play in another thread

Some devices (like Flash and Lens) add tags to the Frame, no tags to add in SoundPlayer

Ex 6: Synchronize sound (Device, Action)


```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <FCam/Tegra.h>

/** \file */

/* Macros for Android logging
 * Use: adb logcat FCAM_EXAMPLE6:*  *:S
 * to see all messages from this app
 */
#if !defined(LOG_TAG)
#define LOG_TAG "FCAM_EXAMPLE5"
#define LOGV(...) __android_log_print(ANDROID_LOG_VERBOSE, LOG_TAG, __VA_ARGS__)
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG  , LOG_TAG, __VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO   , LOG_TAG, __VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN   , LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR  , LOG_TAG, __VA_ARGS__)
#endif
#include <android/log.h>

#include "SoundPlayer.h"

namespace Plat = FCam::Tegra;
const int width  = 2560;
const int height = 1920;
const FCam::ImageFormat format = FCam::YUV420p;
const char output_image[] = "/data/fcam/example6.jpg";

// The native assetManager
extern AAssetManager* nativeAssetManager;


/*************************************************************/
/* Shutter sound                                             */
/*                                                           */
/* This example shows how to declare and attach a device,    */
/* and write the appropriate actions. In this example, the   */
/* camera will trigger two actions at the beginning of the   */
/* exposure: a flash, and a shutter sound.                   */
/* See SoundPlayer class for more information.               */
/*************************************************************/
int main(int argc, char ** argv) {

    if (nativeAssetManager == NULL)
    {
        LOGE("setAssets() was not called - can't get assetManager()\n");
        return 0;
    }

    // Devices
    Plat::Sensor sensor;
    Plat::Flash flash;

    // We defined a custom device to play a sound during the
    // exposure. See SoundPlayer.h/cpp for details.
    SoundPlayer audio(nativeAssetManager);

    sensor.attach(&flash); // Attach the flash to the sensor
    sensor.attach(&audio); // Attach the sound player to the sensor

    // Set the shot parameters
    Plat::Shot shot1;
    shot1.exposure = 50000;
    shot1.gain = 1.0f;
    shot1.image = FCam::Image(width, height, format);

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

    // Action (Sound)
    SoundPlayer::SoundAction click(&audio);
    click.time = 0; // Start at the beginning of the exposure
    click.setAsset("camera_snd.mp3");

    // Attach actions
    shot1.addAction(fire);
    shot1.addAction(click);

    // Order the sensor to capture a shot.
    // The flash and the shutter sound should happen simultaneously.
    sensor.capture(shot1);
    assert(sensor.shotsPending() == 1); // There should be exactly one shot

    // Retrieve the frame from the sensor
    Plat::Frame frame = sensor.getFrame();
    assert(frame.shot().id == shot1.id); // Check the source of the request

    // Write out the file
    FCam::saveJPEG(frame, output_image);

    // Check that the pipeline is empty
    assert(sensor.framesPending() == 0);
    assert(sensor.shotsPending() == 0);
}
```

Previous course projects

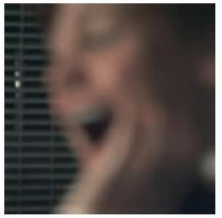
graphics.stanford.edu/courses/cs448a-10/

CS 448A - Computational photography

A cutaway view showing some of the optical and electronic components in the Canon 5D, a modern single lens reflex (SLR) camera. In the first part of this course, we'll take a trip down the capture and image processing pipelines of a typical digital camera.

This is the Stanford Frankencamera, an experimental open-source camera we are building in our laboratory. It's bigger, heavier, and uglier than the Canon camera, but it runs Linux, and its metering, focusing, demosaicing, denoising, white balancing, and other post-processing algorithms are programmable. We'll eventually be distributing this camera to researchers worldwide.

This is the Nokia N900, the first in a new generation of Linux-based cell phones. It has a 5-megapixel camera and a focusable Carl Zeiss lens. More importantly, it runs the same software as our Frankencamera, so it's programmable right down to its autofocusing algorithm. Each student taking this course for credit will be loaned an N900.



In the second part of the course, we'll consider problems in photography and how they can be solved computationally. One such problem is misfocus. By inserting a microlens array into a camera, one can record light fields. This permits a snapshot to be refocused after capture.

Most digital cameras capture movies as well as stills, but handshake is a big problem, as exemplified by the home video above. Fortunately, stabilization algorithms are getting very good; look at this experimental result. We'll survey the state-of-the-art in this evolving area.

One of the leading researchers in the new field of computational photography is Fredo Durand of MIT. Fredo is on sabbatical at Stanford and has agreed to co-teach this course and advise student projects. You'll be seeing a lot of Fredo this quarter.

Quarter
Winter, 2010
Units
3-4 (same workload) (+/NC or letter grade)
Time
Tue/Thu 2:15 - 3:30
Place
392 Gates Hall (graphics lab conference room)
Course URL
http://graphics.stanford.edu/courses/cs448a-10/
Newsgroup
su.class.cs448a

Instructors
Marc Levoy, Fredo Durand, Jongmin Baek (teaching assistant)
Office hours
Marc Levoy: Tue/Thu, 4:15 - 5:30 (Gates 366)
Fredo Durand: by appointment (Gates 362)
Jongmin Baek: Fri 10:00 - 11:45 / Thu 3:45 - 5:30 (Gates 360)
Prerequisite
An introductory course in graphics or vision, or CS 178, good programming skills
Televised?
No

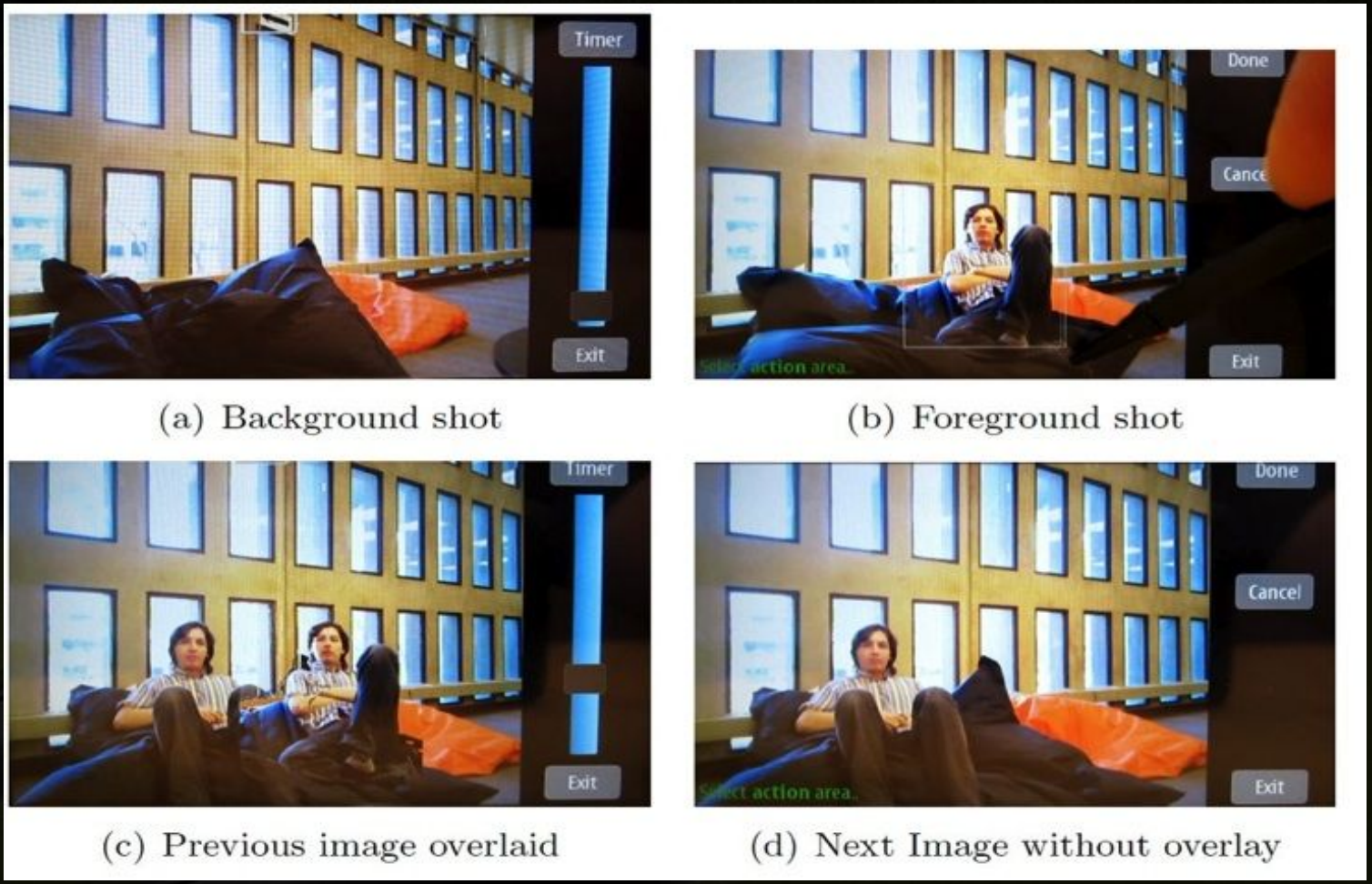
Remote flash over Bluetooth

- Send a device action to another N900 over Bluetooth

Blur-free low-light photography

- Short/long exposure fusion using blind deconvolution

Interactive Photomontage



(a) Background shot (b) Foreground shot

(c) Previous image overlaid (d) Next Image without overlay

Interactive Photomontage

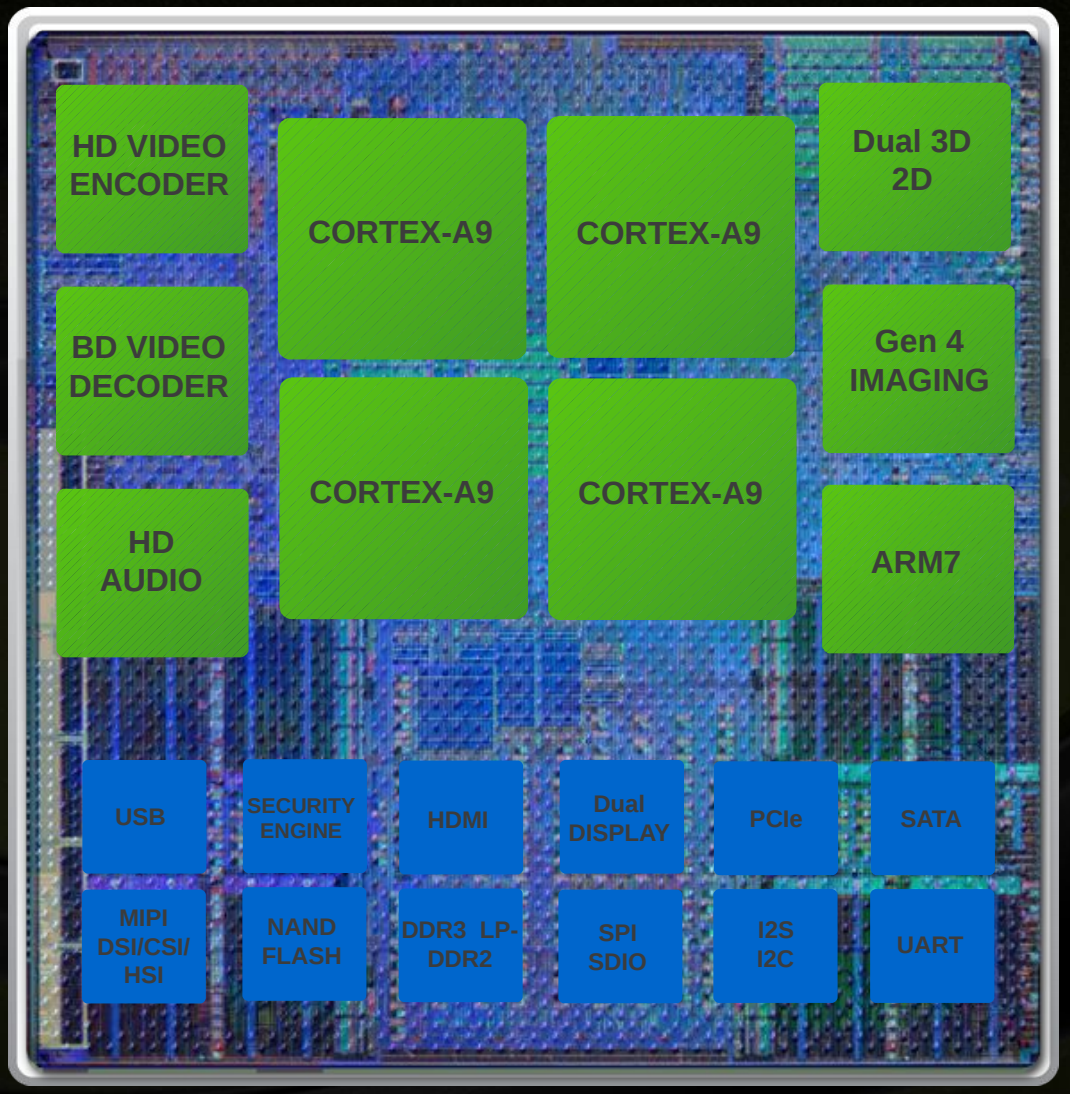

Timer
Exit

Painted aperture for portraits

(a) Using 8 images for blur.

(b) Using 16 images for blur.

Tegra 3 – Dedicated Processing



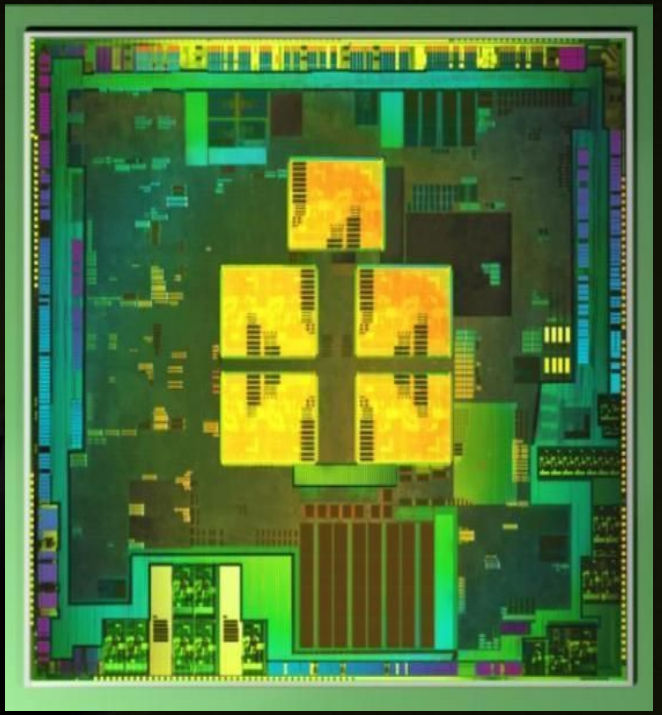
NVIDIA Research

CPU	3X Performance *Quad Cortex-A9 up to 1.5GHz, ULP Mode* *NEON*
POWER	**20x Lower Power** ***ULP Mode***
VIDEO	**4X Complexity** ***1080i/p High Profile***
GRAPHICS	**4X Performance** ***Dual Pixel Pipe, Tiled Compositor***
MEMORY	**3X bandwidth** ***DDR3L up to 1600 data rate***
IMAGING	**Better noise reduction & color rendition** ***Gen 4 ISP***
AUDIO	**HD Audio** ***7.1 channel surround***
STORAGE	**2 – 6X faster**

Tegra 3

The World's First Mobile Quad Core with 5th Companion Core for Low Power

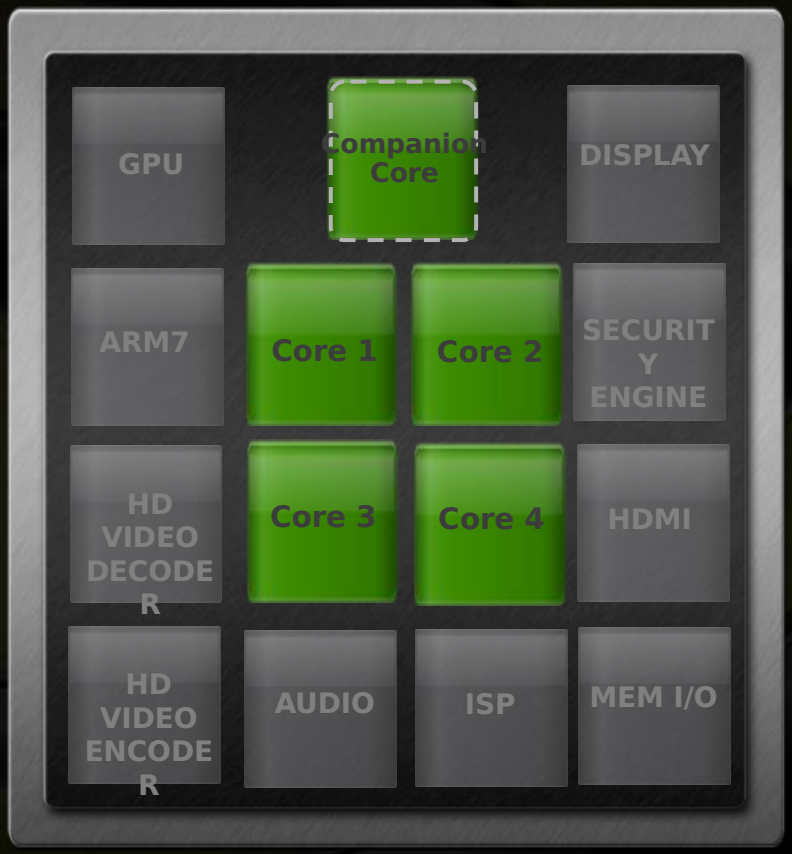
- 5x Tegra 2
- Lower Power
- 3x Faster GPU,
  - with Stereo 3D
- PC-Class CPU



4 Cores + a Companion Core

Patented architecture for Lowest Power and Highest Performance

5 CPU Cores



- 5th “Companion” core - low power for active standby, music, and video
- Four performance cores for max burst when needed
- Each core automatically enabled and disabled based on workload
- Companion core is OS transparent

Scalable CPU Cores –

Max CPU Performance, Max Battery Life

Increasing performance

Use case	Cores Enabled	Frequency	ULP
Multi-threaded App, Various Perf	1-4	Variable	Off
Single Threaded App, High Perf	1	Max	Off
OS Maintenance (Standby)	ULP	<500 MHz	On
Phone / Audio / Video			

Unused Cores are Powered Off

Android SDK

- The main Android SDK is based on Java
  - the language is very similar to Sun's original Java
    - UI libraries are different from Java SE
    - a different VM (called Dalvik)
- http://developer.android.com/
  - one-stop-shop for Android tools and documentation
- The easiest way to install all the needed tools is
  - http://developer.nvidia.com/tegra-android-development-pack
  - JDK, SDK, NDK, Eclipse, …
  - plus Tegra tools

Android and Native Code

- Native code adds some complexity
  - but unlocks peak performance (games, image processing)
  - makes porting existing (C/C++) code easier
  - reduces portability
- A set of cross-compiler toolchains
  - gcc for ARM
  - hosted on Windows, Linux, or Mac
- JNI, the Java/Native Interface
  - connecting native code to Java
- Gingerbread (Android 2.3) introduced NativeActivity
  - avoids the need for any Java code

FCam Android app anatomy

- Android `Activity` calls a native C++ method `run()`
  - `run()` launches a thread and calls the example's `main()`
- When the thread exits, the `Activity` is notified
- The JNI C framework code is provided
  - only need to implement `main`

Java: FCam Android Activity

```
public class Example extends Activity
{
    [......]
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        [......]
        // invoke the native code - which will launch its own thread.
        run();
    }
    /* onCompletion is called from the native code when the example code is done */
    public void onCompletion()
    {
        hNotificationHandler.sendEmptyMessage( COMPLETED );
    }
    /* A native method that runs the fcam API code and that is included in the
     * 'fcam_example' native library, which is packaged with this application. */
    public native int run();
```

C++: Android FCam JNI

```
static JavaVM  * gJavaVM;
static jclass    exampleJClass;
static jobject   exampleInstance;
static fields_t  fields;
static pthread_t fcamthread;
```

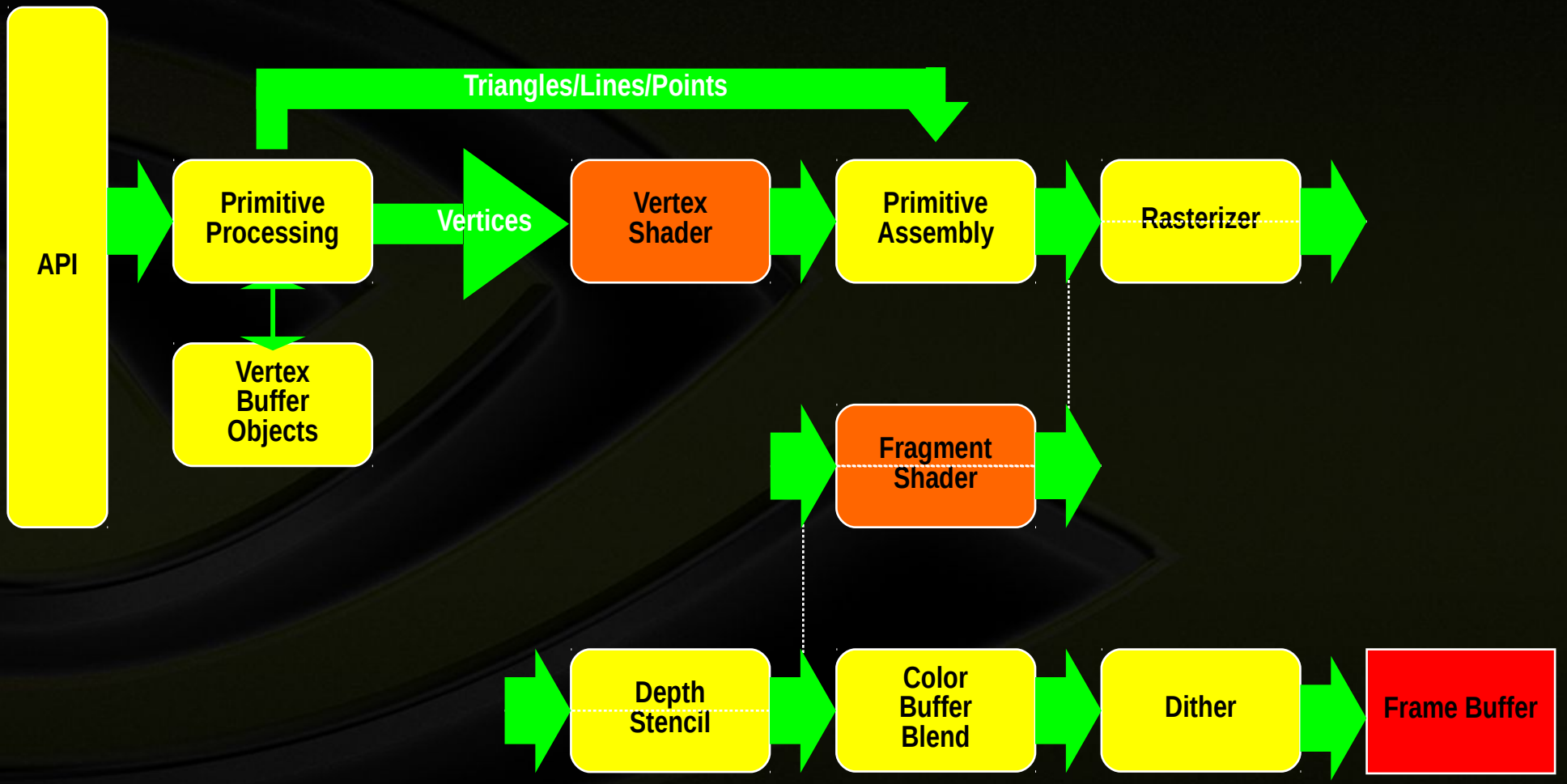
```
void *fcam_thread_(void *arg)
{
    jniAttachThread();
    main( 0, NULL );
    notifyCompletion();
    deleteGlobalRefs();
    jniDetachThread();
    pthread_exit( NULL );
    return NULL;
}
```

```
JNIEXPORT jint JNICALL Java_com_nvidia_fcam_example1_Example_run (JNIEnv *env,
                                                                 jobject thiz)
{
    // Register the Java based instance, class and methods we need to later call.
    exampleInstance     = env->NewGlobalRef( thiz );
    exampleJClass       = (jclass)env->NewGlobalRef( env->GetObjectClass(thiz) );
    fields.completionCb = env->GetMethodID( exampleJClass, "onCompletion", "()V" );
    // Launch the fcam thread
    pthread_create( &fcamthread, NULL, fcam_thread_, NULL );
}
```

3D Graphics with OpenGL ES 2.0

NVIDIA Research

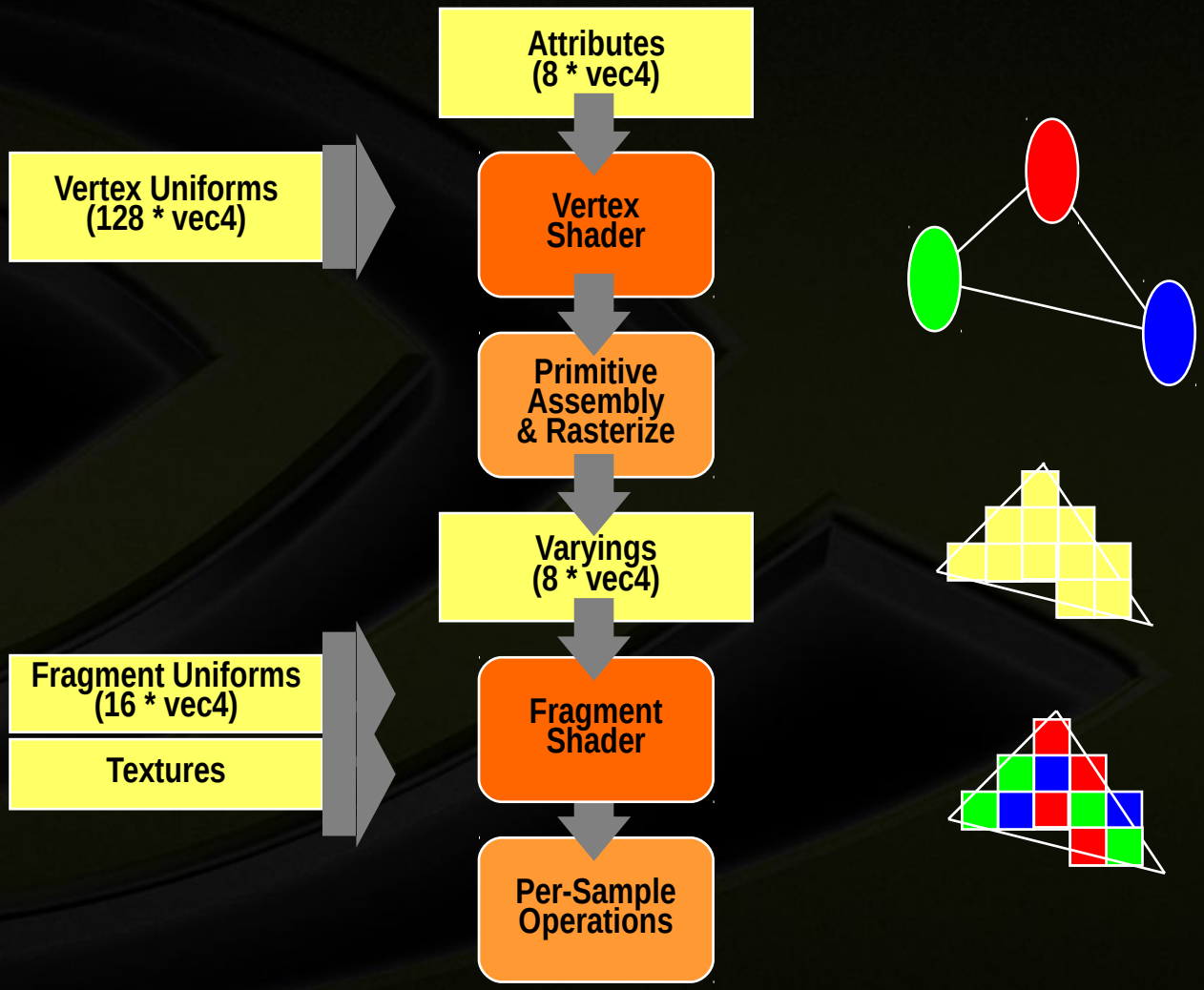
OpenGL ES 2.0 Programmable pipe



OpenGL ES 2.0 is a very bare-bones API

- Setup
- Input of per-object constants (uniforms)
  - no matrix calculation support in the API
    - do it on CPU with other utility APIs
- Input of per-vertex data (attributes)
  - no special property types
    - normals, texture coordinates, …
  - it's up to the shaders to interpret what the numbers mean
- And the shaders of course
  - sources as strings, compiled and linked on the fly
  - connect CPU variables with shader variables

Programmer's model



OpenGL ES 2.0 on Android NDK

- The following example comes from
  - http://developer.nvidia.com/tegra-android-development-pack
  - using its app_create.sh script

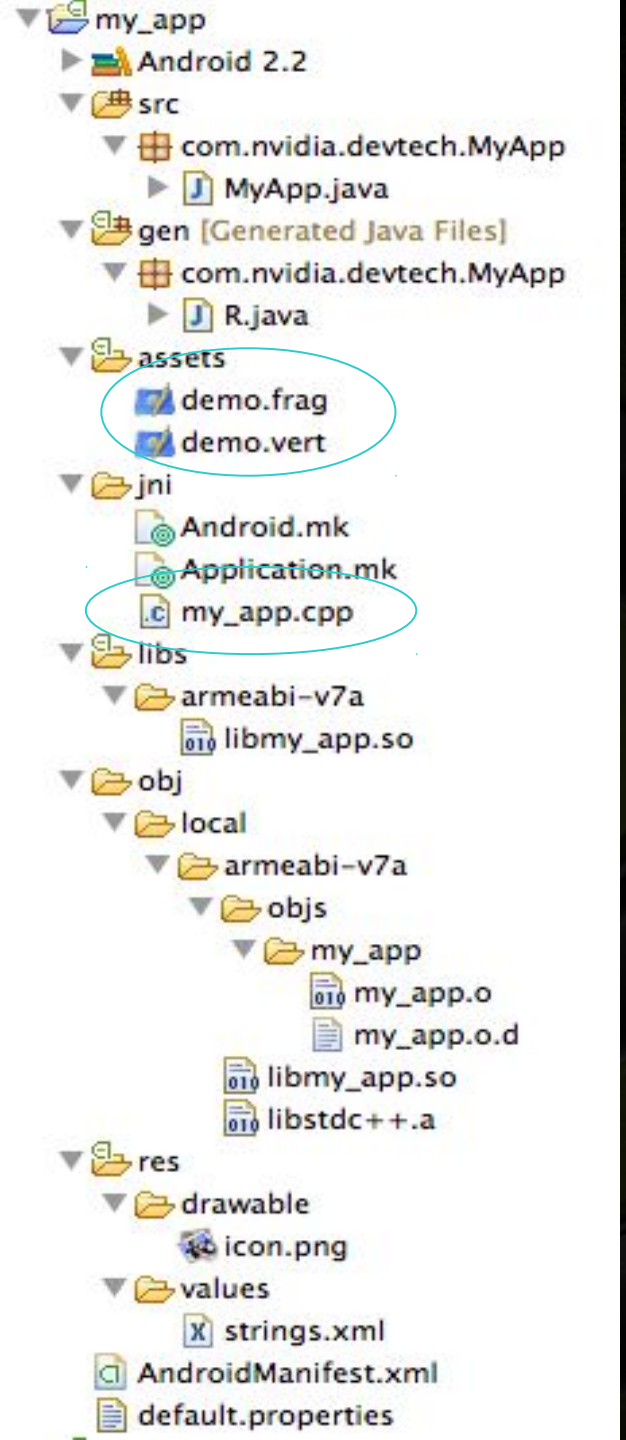
Tegra pack examples

- Import all sample projects to Eclipse workspace
  - set up environment variables, etc., following instructions on the pdf file
  - build all, try on a device
    here is es2_globe

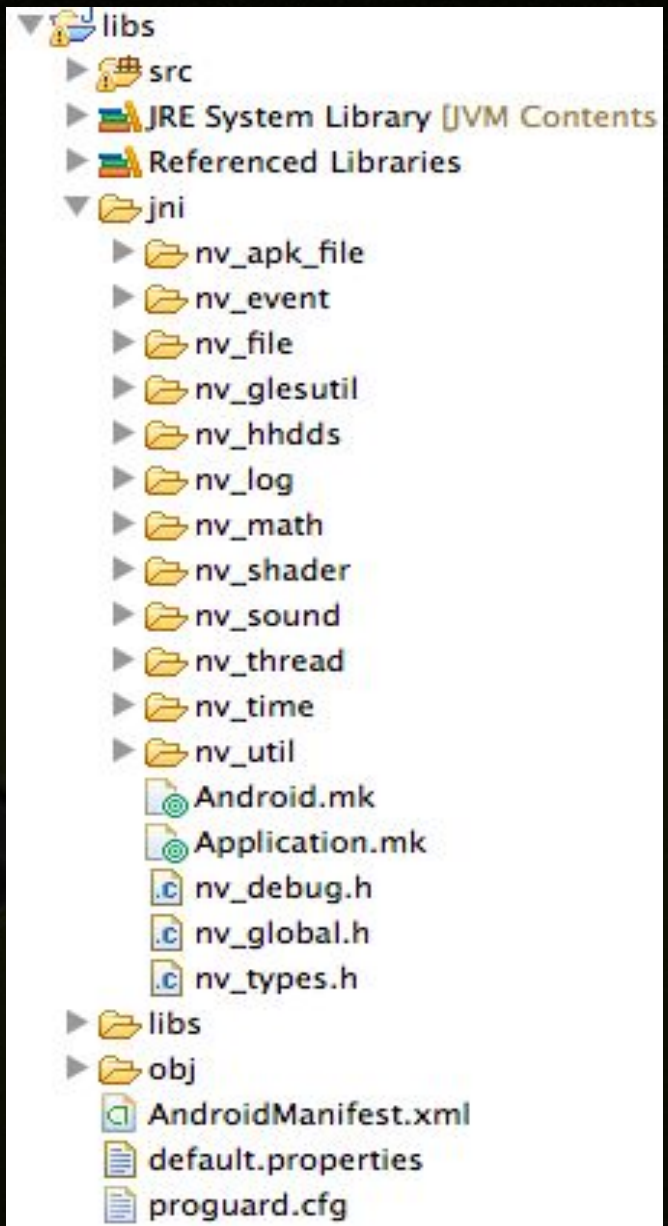
- Create your own sample app

```
karip@mac:~/NVPACK/TDK_Samples/Android_NVIDIA_samples_2_20110315/tools/app_create$
./app_create.sh my_app MyApp basic
INFO: Java/native app being placed in apps
creating destination directory tree ../../apps/my_app
copying files to destination tree
```

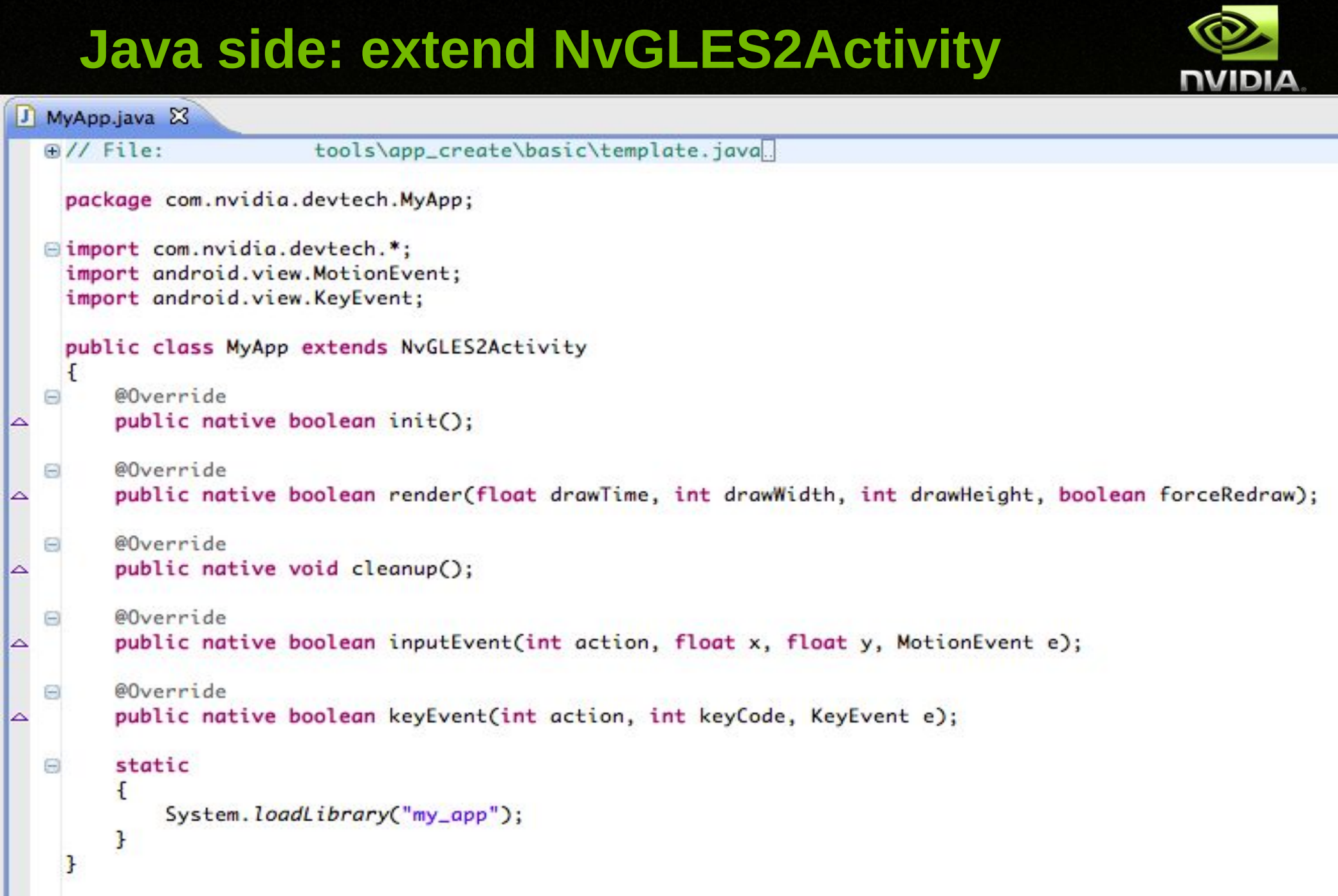
- Build and try it



- The project
- Also need a library project for various utilities



Java side: extend NvGLES2Activity



Matching part from C side

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv *env;

    NVThreadInit(vm);

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        DEBUG("Failed to get the environment using GetEnv()");
        return -1;
    }
    JNINativeMethod methods[] = {
        { "init",       "()Z",                                  (void *) init },
        { "render",     "(FIIZ)Z",                              (void *) render },
        { "inputEvent", "(IFFLandroid/view/MotionEvent;)Z", (void *) inputEvent },
        { "keyEvent",   "(IILandroid/view/KeyEvent;)Z",     (void *) keyEvent },
        { "cleanup",    "()V",                                  (void *) cleanup }
    };
    jclass k = (env)->FindClass ("com/nvidia/devtech/MyApp/MyApp");
    (env)->RegisterNatives(k, methods, 5);

    NVTimeInit();

    return JNI_VERSION_1_4;
}
```

Match the C functions,
with their types,
to Java functions

Initialize shaders and attributes

```
#define ROOT_3_OVER_2 0.8660254f
#define ROOT_3_OVER_6 (ROOT_3_OVER_2/3.0f)
static GLfloat s_vert[6] = { 0.5f, -ROOT_3_OVER_6,
                            -0.5f, -ROOT_3_OVER_6,
                             0.0f,  ROOT_3_OVER_2 - ROOT_3_OVER_6 };
static GLfloat s_col [12] = { 1.0, 0.0, 0.0, 1.0,
                              0.0, 1.0, 0.0, 1.0,
                              0.0, 0.0, 1.0, 1.0};

jboolean init(JNIEnv*  env, jobject  thiz)
{
    nv_shader_init();
    DEBUG("GL_VENDOR: %s", glGetString(GL_VENDOR));
    DEBUG("GL_VERSION: %s", glGetString(GL_VERSION));
    DEBUG("GL_RENDERER: %s", glGetString(GL_RENDERER));

    s_prog = nv_load_program("demo");

    s_angle = 0;
    s_translationX = 0;
    s_translationY = 0;

    return JNI_TRUE;
}
```

Vertex position and color data

NV utilities help in getting started

Really simple shaders

demo.vert

```
uniform vec2 rot;          // rot.xy = {cos(theta), sin(theta)}
uniform vec2 translation; // translation.xy
uniform float aspect;
attribute vec2 pos_attr;
attribute vec4 col_attr;
varying vec4 col_var;
void main()
{
    gl_Position.x =  rot.x * pos_attr.x - rot.y * pos_attr.y + translation.x;
    gl_Position.y = (rot.y * pos_attr.x + rot.x * pos_attr.y + translation.y) * aspect;
    gl_Position.z = 0.0;
    gl_Position.w = 1.0;

    col_var = col_attr;
}
```

Per object consts

Per vertex data

Outputs from vertex to fragment shader

demo.frag

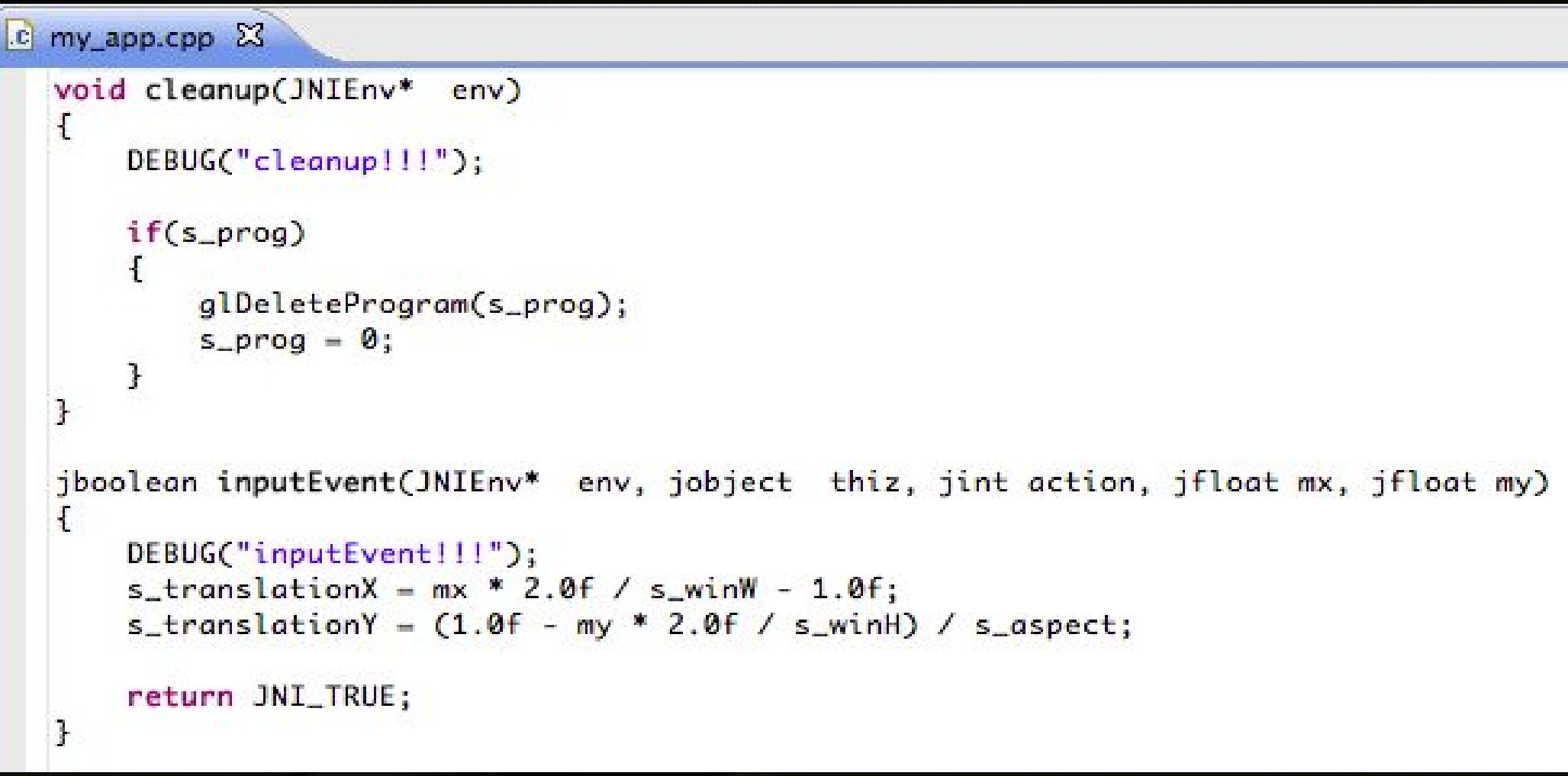
```
precision mediump float;

varying vec4 col_var;
void main()
{
    gl_FragColor = col_var;
    //gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Rendering callback function

my_app.cpp

```
jint render(JNIEnv*  env, jobject  thiz, jfloat javatime,
            jint drawWidth, jint drawHeight, jboolean forceRedraw)
{
    s_winW = (float)drawWidth;
    s_winH = (float)drawHeight;
    s_aspect = s_winW / s_winH;

    if (s_paused)
        return 0;

    s_angle += s_delta;

    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(s_prog);

    nv_set_attrib_by_name(s_prog, "pos_attr", 2, GL_FLOAT, 0, 0, s_vert);
    nv_set_attrib_by_name(s_prog, "col_attr", 4, GL_FLOAT, 0, 0, s_col);

    float rad = (float)(s_angle * 0.0174532924F);
    glUniform2f(glGetUniformLocation(s_prog, "rot"), (float)cos(rad), (float)sin(rad));
    glUniform2f(glGetUniformLocation(s_prog, "translation"), s_translationX, s_translationY);
    glUniform1f(glGetUniformLocation(s_prog, "aspect"), s_aspect);

    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_BYTE, s_ind);

    return 1; // return true to update screen
}
```

Cleanup, and touch input processing



GPGPU Image Processing

- How-To:
  - Set Viewport to output image size
  - Render Full-Screen Quad
  - Output pixels are processed in parallel as fragment shader invocations
- Notes:
  - Limited set of data types (no floats)
  - Supports only data gathering, not scattering
  - Only fragment processors are used
    - VPs, raster, etc., are pretty much idle
  - Higher memory BW than CPU

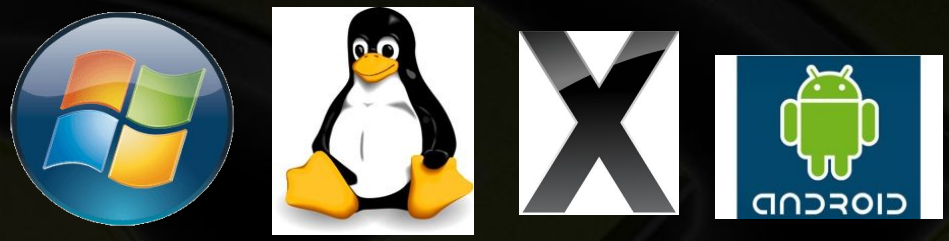
Measuring compute

- Measure 5x5 convolution
  - CPU: 2200
  - CPU + MT: 560
  - Neon: 380
  - Neon + MT: 100
  - GPU: 30

OpenCV

Thousands of Developers, Cross Platform API

- Open standard for Computer Vision



- 12 years old, professionally developed
  - Over 3 Million Downloads!
- > 500 Algorithms

Common API for Server, Workstation, Desktop and now Mobile Platforms!

OpenCV functionality overview

General Image Processing

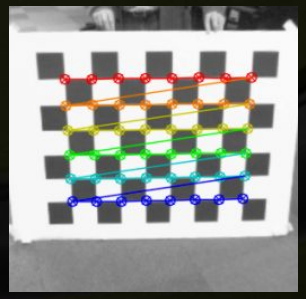
Segmentation

Machine Learning, Detection

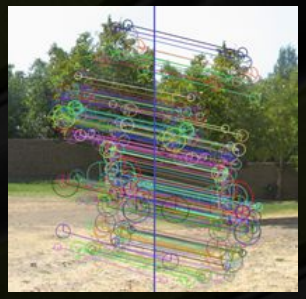
Image Pyramids

Transforms

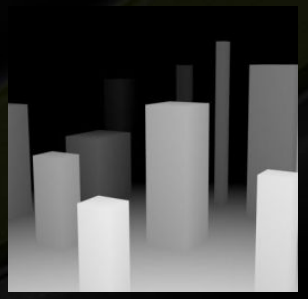
Fitting



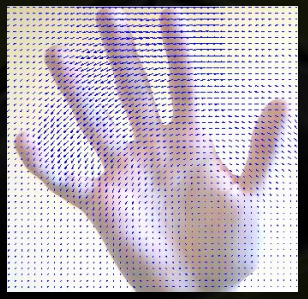
Camera Calibration



Features



Depth Maps



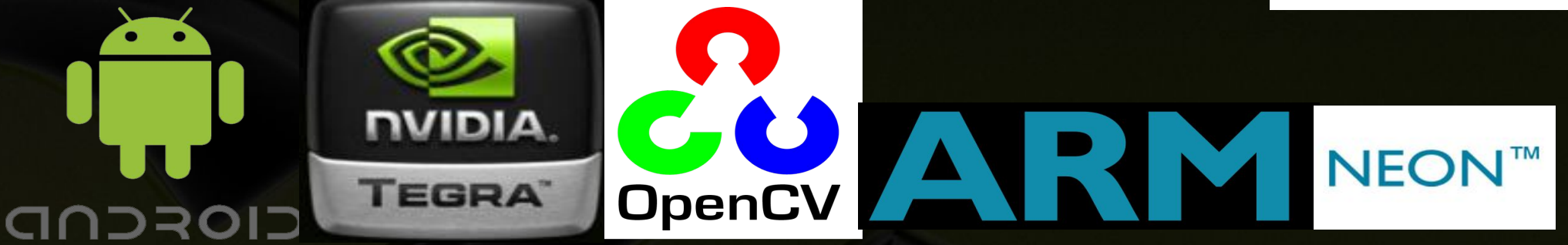
Optical Flow

Inpainting

Tracking

OpenCV for Android

Optimized for ARM, Tegra, & Android



- **Install from**
  - **http://opencv.itseez.com/doc/tutorials/introduction/android_binary_package/android_binary_package.html**

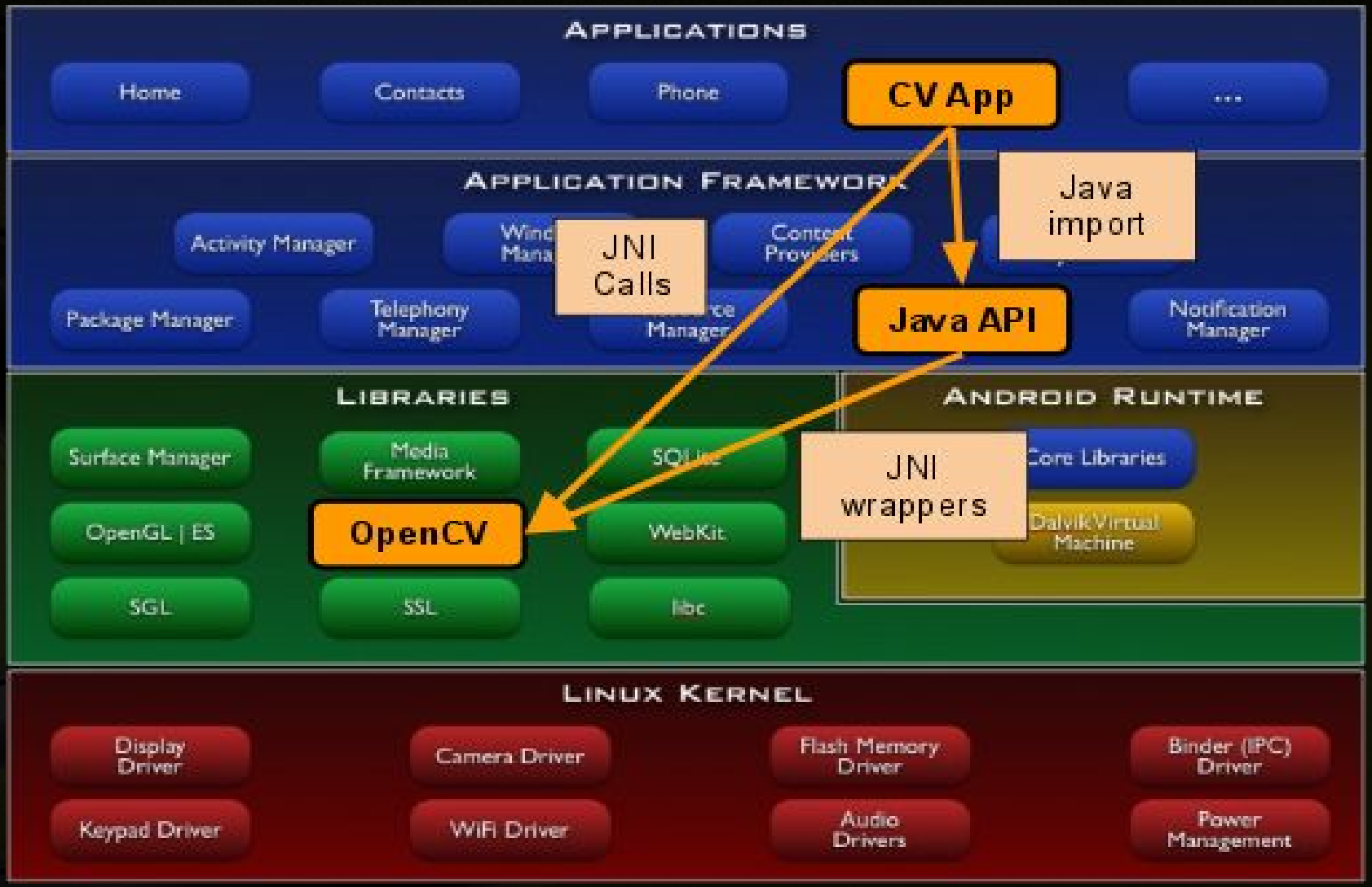
Tutorial: Android Camera

- Part of the Android OpenCV distribution
- Get camera image
- Display it

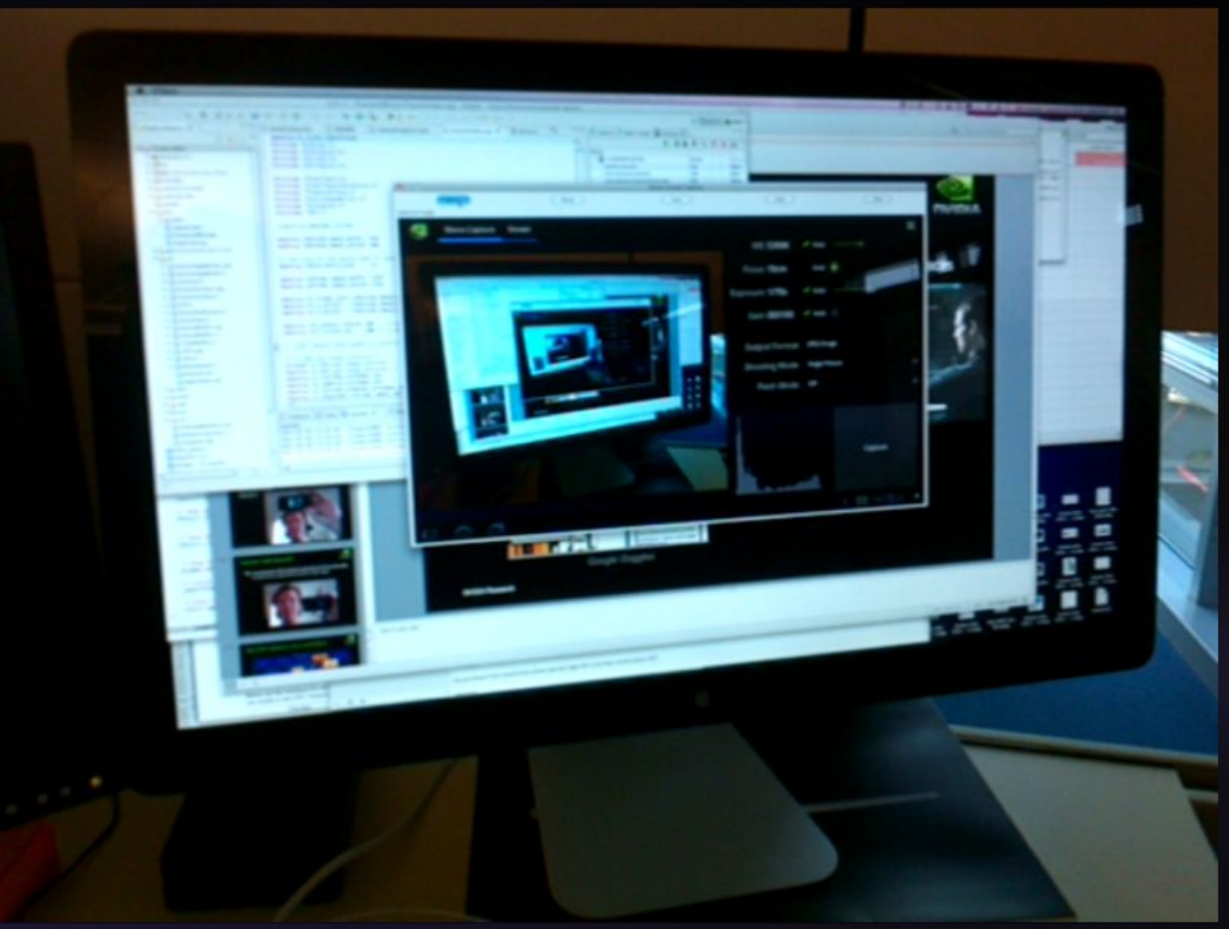
Tutorial: Add OpenCV

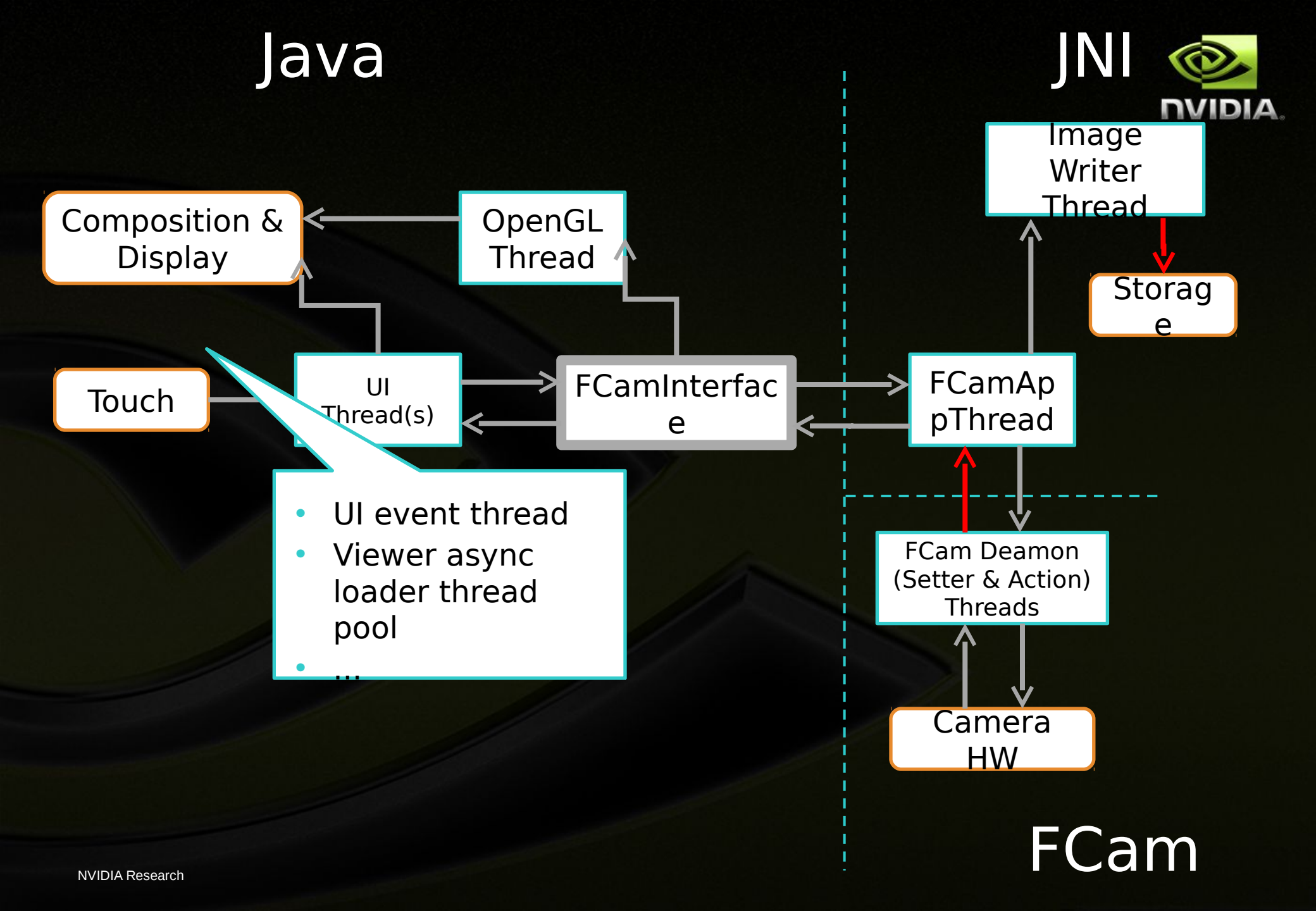
- The second part of the tutorial adds OpenCV functionality
  - real-time Canny edge detector from the input image

OpenCV supports Java and Native



FCamera – sample camera app




Java
JNI
NVIDIA
Image Writer Thread
Composition & Display
OpenGL Thread
Storag e
Touch
UI Thread(s)
FCamInterfac e
FCamAp pThread
UI event thread
Viewer async loader thread pool
...
FCam Deamon (Setter & Action) Threads
Camera HW
FCam

Additional material

- http://developer.nvidia.com/developer-webinars
  - Optimizing NVIDIA Tegra Android With Oprofile And Perf
  - High-Performance Graphics With NVIDIA Tegra Using PerfHUD ES
  - Tegra Debugging With NVIDIA Debug Manager For Android
- http://developer.nvidia.com/tegra-resources-archive
  - Android Application Lifecycle in Practice: A Developer's Guide
  - Android Accelerometer Whitepaper
  - …