Tools for Mobile Computational Photography

Alejandro Troccoli
Senior Research Scientist
NVIDIA
Principles of photography

18th century camera obscura

Susse Frére Daguerreotype camera 1839

Digital SLR

Mostly unchanged!
Capturing a moment
Capturing a moment
Computational photography

- Extend photography with computing

- Capture one or more photographs
- Process
- Render
Panoramic stitching
High dynamic range imaging
Lucky imaging
Lucky imaging
All-in-Focus Imaging
Low-light imaging
Mobile Computational Photography

- Computational photography for everyone
- Take, process and share instantly!
Trends in camera phone sales

- Sales keep growing
  - 2003 85 million 16% of the phones
  - 2010 805 million 65% of the phones
  - 2014 1.3 billion 85% of the phones

- Average resolution grows too
  - 2008 1 MP
  - 2009 2 MP
    - 2010 5+MP: ~ 13%, >100 million
  - 2014 5 MP
    - 2014 5+MP: ~ 42%, >550 million
Mobile photography challenges

Small optics
Handshake
Image noise
Per frame camera control
Post-processing
Per frame camera control

- Traditional camera APIs
  - autofocus()
  - takePicture()

- High latency on settings changes
What is the problem?

- The user has to wait another couple of seconds
- How bad is that?
Wrong model

Real imaging systems are pipelined
The FCam Architecture

• A software architecture for programmable cameras
The Sensor

- A pipeline that converts requests into images
- No visible state
- All parameters packed into the requests
Image Signal Processor (ISP)

- Receives sensor data, and optionally transforms it
- 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
- No hidden daemon running autofocus/metering
Implementations
FCam is an open-source C++ API for easy and precise control of digital cameras. It allows full low-level control of all camera parameters on a per-frame basis, making it easy to rewrite the camera's autofocus routine, to capture a burst of images all with different parameters, and to synchronize the operation of the camera lens and flash with all of the above.

FCam is the result of the Camera 2.0 joint research project on programmable cameras and computational photography between Marc Levoy's group in the Stanford Computer Graphics Laboratory and Kari Pulli's team at Nokia Research Center Palo Alto. A paper describing the FCam architecture, the motivation behind it, and some applications, was presented at SIGGRAPH 2010.
Shot - parameters for capture

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

FCam::Tegra::Shot shot;

// Configure sensor capture properties
shot.exposure = 20000;  // Exposure in microseconds
shot.frameTime = 33000; // Frame rate 30fps
shot.gain = 1.0f;       // Sensor gain
shot.image = FCam::Image(2592, 1944, FCam::YUV420p);

// Post-processing parameters
shot.whiteBalance = 3200;  // Color temperature
shot.histogram.enabled = true;
shot.sharpness.enabled = true;
```
Shot is passed to a Sensor

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

// Wait for the frame
frame = sensor.getFrame();
Bracketed capture example

```cpp
std::vector<FCam::Tegra::Shot> shot(3);

// Configure the burst
shot[0].frameTime = 33000;  // Frame rate 30fps
shot[0].gain = 1.0f;  // Sensor gain


shot[0].exposure = 10000;  // Medium exposure
shot[1].exposure = 20000;  // Long exposure
shot[2].exposure = 5000;  // Short exposure

shot[0].image = FCam::Image(1920, 1080, FCam::YUV420p);
shot[1].image = FCam::Image(1920, 1080, FCam::YUV420p);
shot[2].image = FCam::Image(1920, 1080, FCam::YUV420p);
```
Bracketed capture example

// Send the vector for capture
sensor.capture(shot);

FCam::Tegra::Frame frame[3];

frame[0] = sensor.getFrame();
frame[1] = sensor.getFrame();
frame[2] = sensor.getFrame();
Flash device and actions

```cpp
FCam::Tegra::Flash flash;

sensor.attach(&flash);

FCam::Tegra::Flash::FireAction fire(&flash);
fire.time = 0;  // fire at beginning of exposure
fire.duration = shot.frametime;
fire.brightness = flash.maxBrightness();

shot.addAction(fire);

sensor.capture(shot);
```
Lens device and actions

FCam::Tegra::Lens lens;

sensor.attach(&lens);

FCam::Tegra::Lens::FocusAction lensAction(&lens);
lensAction.time = 0;  // start focus with exposure
lensAction.focus = 0;  // in diopters, 0 == inf;
lensAction.speed  = lens.maxFocusSpeed();

shot.addAction(lensAction);

sensor.capture(shot);
Frame parameters and tags

```cpp
frame = sensor.getFrame();
frame.exposure()
frame.whiteBalance()
frame.exposureStartTime()
frame.exposureEndTime()

FCam::Lens::Tags lensTags(frame)
lensTags.initialFocus
lensTags.finalFocus

FCam::Flash::Tags flashTags(frame)
flashTags.brightness
```
Putting it all together

```cpp
shot.exposure = 20000;
shot.gain = 1.0f;
shot.histogram.enabled = true;

sensor.stream(shot);

while (1) {
    FCam::Tegra::Frame frame = sensor.getFrame();
    display(frame);
    autoExpose(&shot, frame);
    autoWhiteBalance(&shot, frame);

    sensor.stream(shot);
}
```
FCameraPro
Post-processing
Image processing using OpenGL ES 2.0

API

Vertex Uniforms & Attributes

Textures

Vertex Shader

Primitive Assembly & Rasterization

Varying(s)

Fragment Uniforms

Fragment Shader

Per-Fragment Operations

Framebuffer
Image processing using OpenGL ES 2.0

Texture

Fragment shader

Quad
Example OpenGL ES 2.0 thresholding

**Vertex shader**

```glsl
attribute vec4 a_position;
attribute vec2 a_texCoord;

varying vec2 v_texCoord;

void main() {
    gl_Position = a_position;
    v_texCoord = a_texCoord;
}
```

**Fragment shader**

```glsl
precision mediump float;

uniform float threshold;
uniform sampler2D s_texture;

varying vec2 v_texCoord;

void main() {
    vec4 color = texture2D(s_texture, v_texCoord);

    // step returns 0 if color < thres, 1 otherwise (for each component)
    gl_FragColor = step(threshold, color);
}
```
Non photorealistic viewfinder

1. Bilateral Filter
2. Edge detection

Fragment shaders

Courtesy of Tony Hyun Kim and Irving Lin
Image processing on CPU

- Quad core
  - Higher multi-threaded performance

- ARM NEON
NEON example: thresholding

C++ implementation

```cpp
for (int i = 0; i < roi; ++i) {
    out[i] = in[i] > thresh ? 255:0;
}
```

Perf for: 4096x4094x4

CPU with 1 threads - time 424ms
NEON with 1 threads - time 37ms

NEON Intrinsics

```cpp
//pack into 8x16 vectors for 16 thresholds at once!
uint8x16_t thresh_u = vdupq_n_u8(pData->thresh);

int j = 0;
for( ; j <= roi; j += CACHE_LINE_SIZE )
{
    uint8x16_t v0  = vld1q_u8((const uint8_t*)(src + j));
    uint8x16_t v1  = vld1q_u8((const uint8_t*)(src + j + 16));
    uint8x16_t r0 = vcgtq_u8(v0, thresh_u);
    uint8x16_t r1 = vcgtq_u8(v1, thresh_u);
    vst1q_u8((uint8_t*)(dst + j), r0);
    vst1q_u8((uint8_t*)(dst + j + 16), r1);
}
```
Image processing using OpenCV

- Open source API 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 stereo matching
OpenCV on Tegra

Being optimized for GPU and ARM NEON
Which one to pick?

Leverage existing code

- OpenCV

Customized

- OpenGL ES
- ARM NEON
Wrapping up

Mobile computational photography

FCam
Camera API

Frames

OpenCV
CPU + GPU

OpenGL ES
GPU

ARM NEON
CPU
A picture is worth a thousand words

Great picture!

Awesome!
Questions?