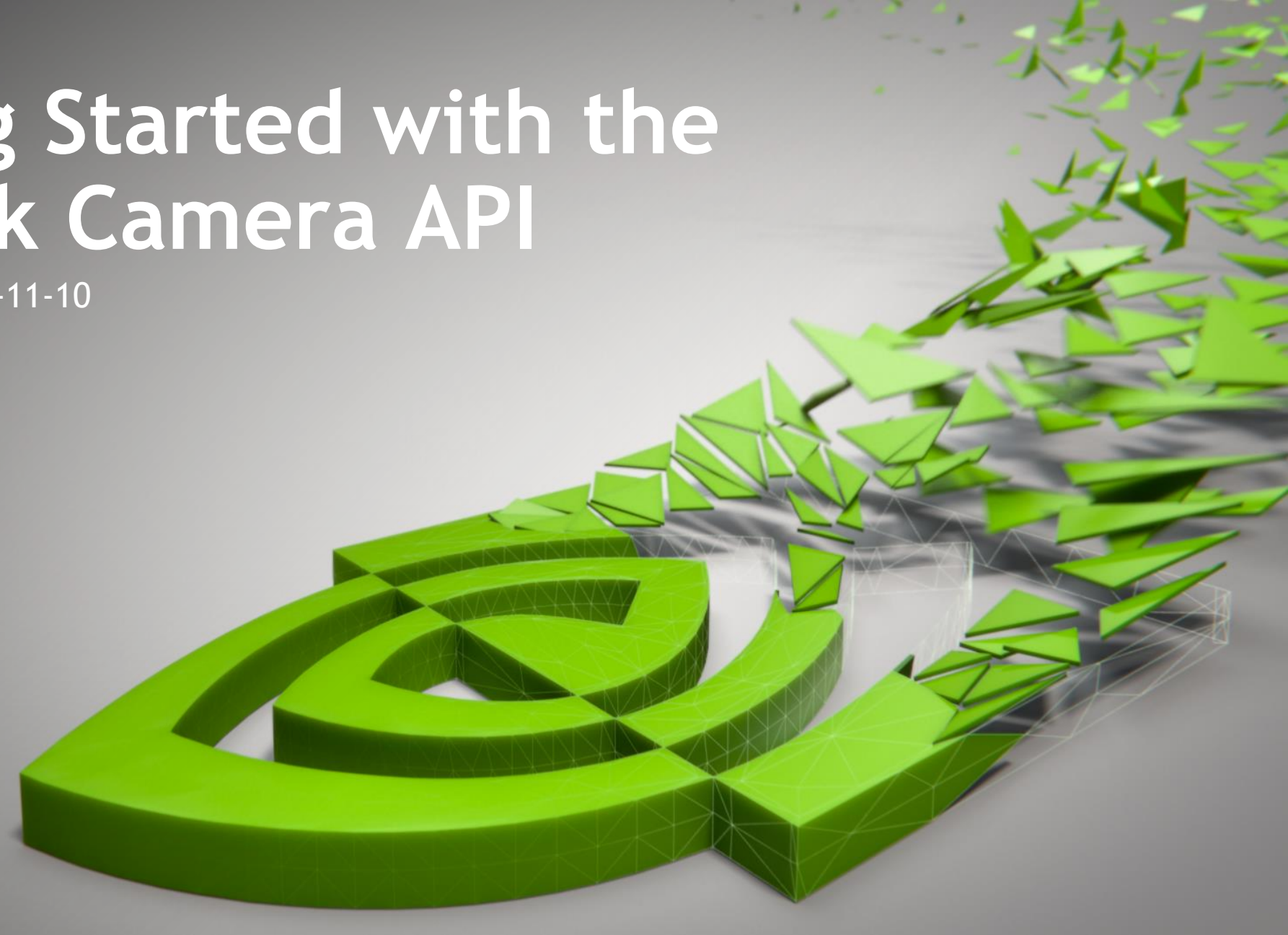
Getting Started with the JetPack Camera API

Ian Stewart, 2016-11-10

NVIDIA



AGENDA

- Jetson TX1 Camera Subsystem
- *libargus*: The JetPack C++ Camera API
  - Core API Elements and Design
  - Simple Camera Application Walkthrough
  - Events, Metadata, and Extensions
  - Writing Efficient Camera Applications
- Consuming and Processing *libargus* Images

Q/A SUPPORT:
BRAD CAIN
PETER MIKOLAJCZYK

JETSON TX1 CAMERA SUBSYSTEM

USB Cameras

USB cameras do not utilize TX1 camera subsystem

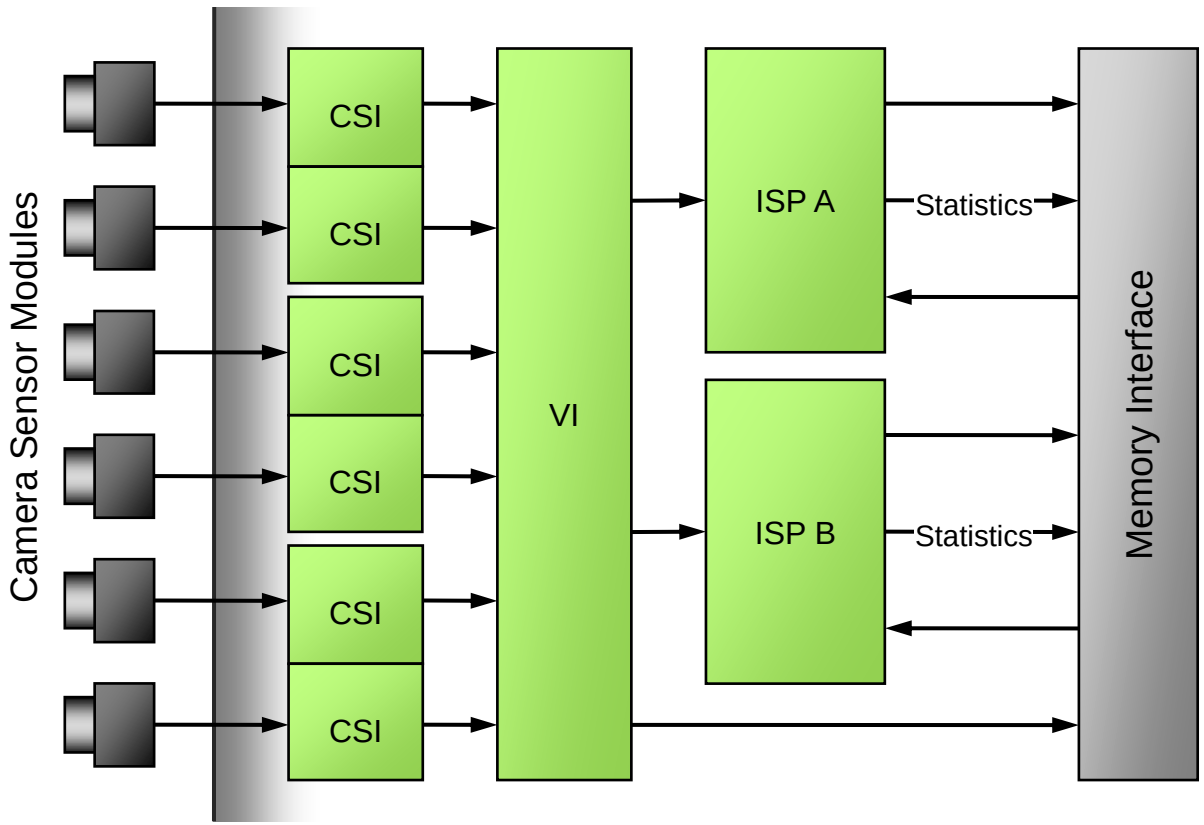
V4L2 provides USB camera support

Documentation: https://linuxtv.org/downloads/v4l-dvb-apis/uapi/v4l/v4l2.html

JETSON TX1 CAMERA SUBSYSTEM

JETSON TX1 CAMERA SUBSYSTEM

Overview



JETSON TX1 CAMERA SUBSYSTEM

Camera Serial Interface (CSI)

MIPI CSI 2.0 standard specification (http://www.mipi.org)

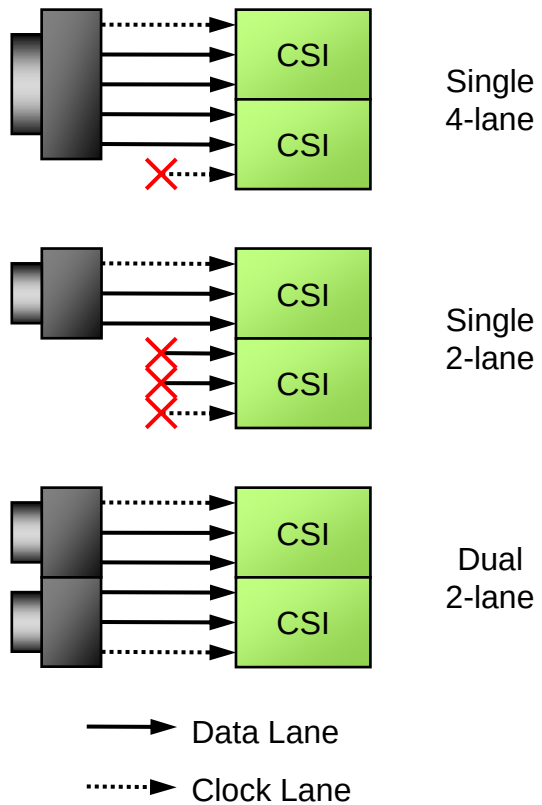
Three CSI x4 blocks, 12 total data lanes

Up to 1.5Gbps per lane

One 4-lane or two 2-lane cameras per block

600MP/s, or 20MP @ 30FPS (4-lane)

Up to six simultaneous camera streams

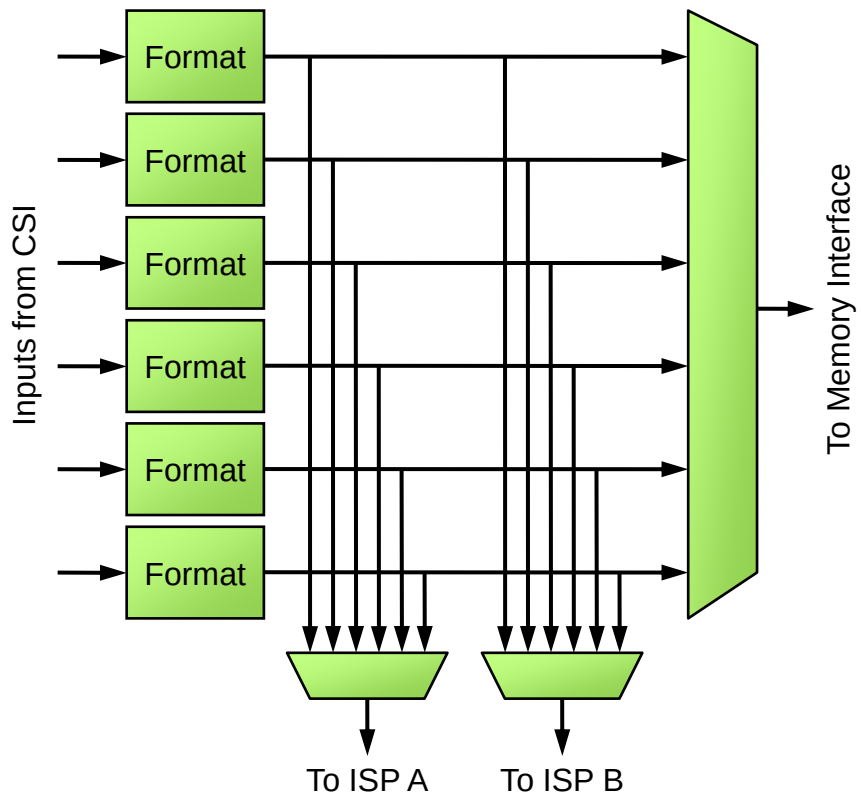


JETSON TX1 CAMERA SUBSYSTEM

Video Input (VI)

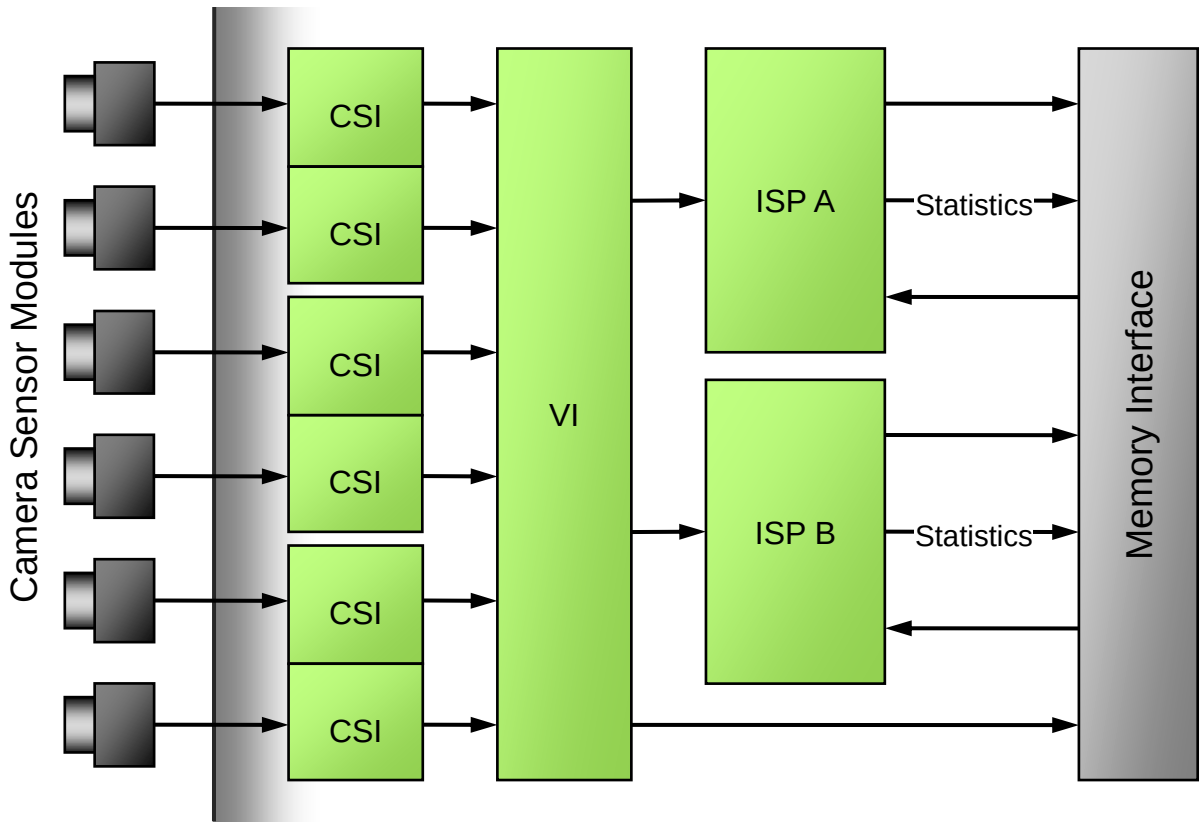
Formats CSI data into pixel streams suitable for memory storage or ISP processing

Routes pixels to memory and/or one or both ISP units



JETSON TX1 CAMERA SUBSYSTEM

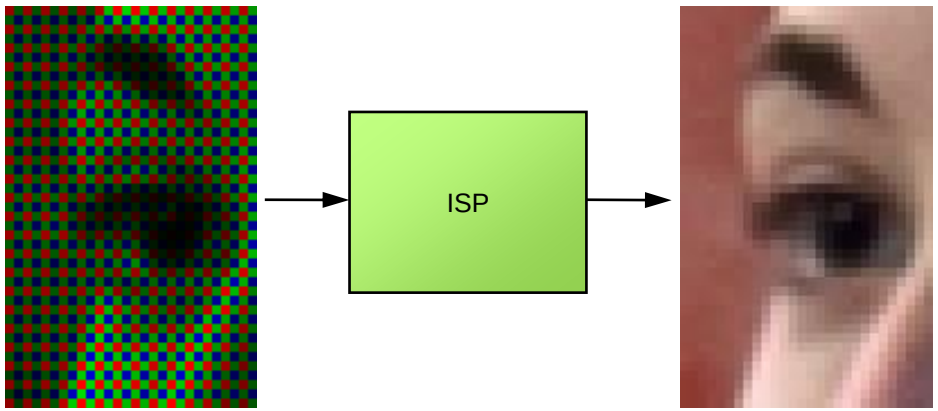
Overview



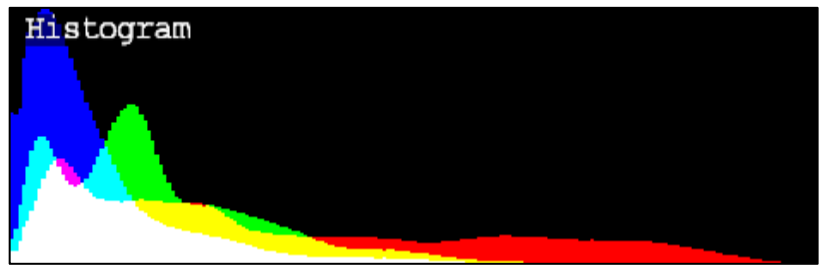
JETSON TX1 CAMERA SUBSYSTEM

Image Signal Processors (ISP)

1) Image Processing

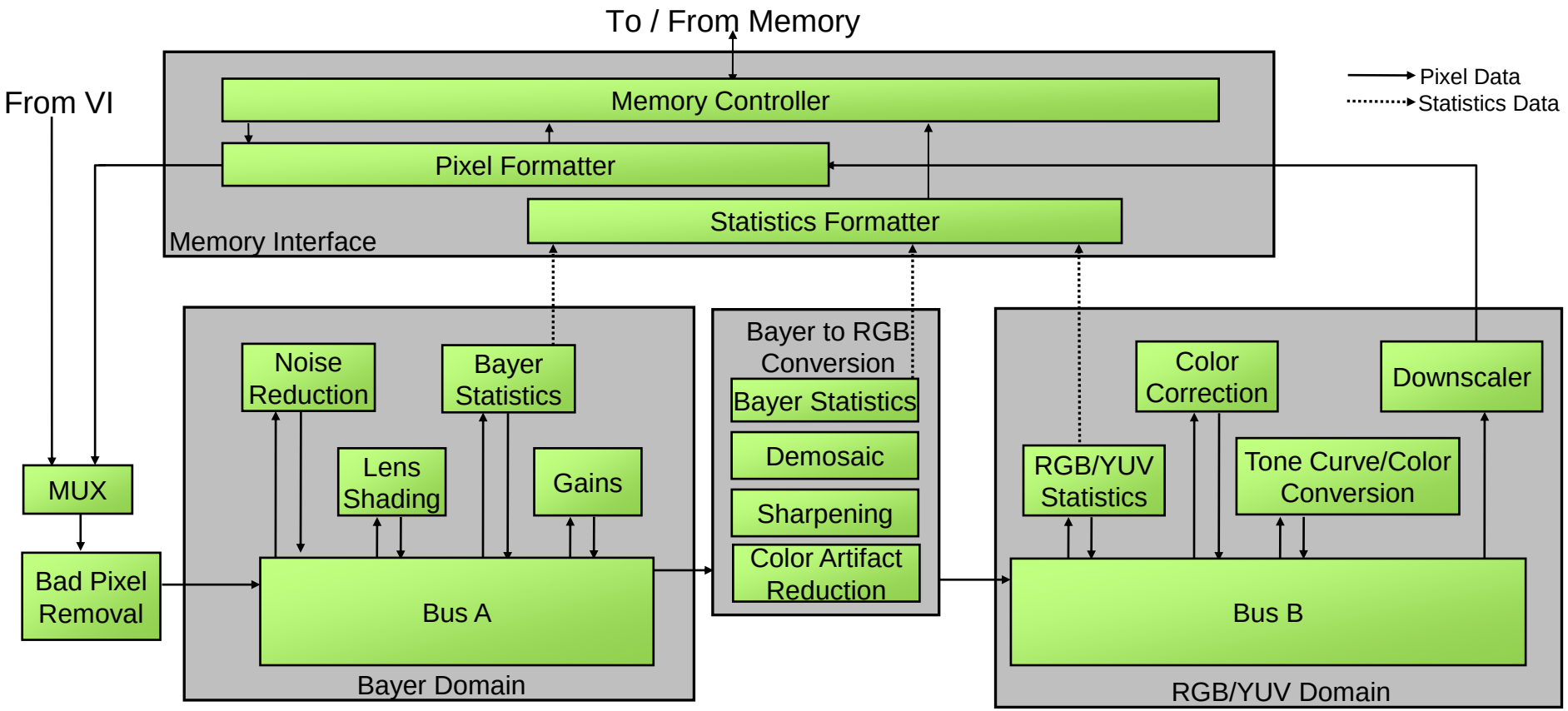


2) Statistics Generation



JETSON TX1 CAMERA SUBSYSTEM

2x Image Signal Processors (ISP)



JETSON TX1 CAMERA SUBSYSTEM

2x Image Signal Processors (ISP)

Sensors up to 6000 pixels wide (27MP)

1200MPix/s combined throughput

Equivalent to 100s of GOPS of CPU/GPU operations

JETSON TX1 CAMERA SUBSYSTEM

Scaling Partner: Leopard Imaging Inc.

Leopard Imaging Inc. specializes in the creation of camera solutions for the TX1 and the Jetson Embedded Program.

Two sensors available today for direct purchase:

IMX185 (1080p) [Purchase] [Data Sheet]

IMX274 (4k): [Purchase] [Data Sheet]

Included adapter board supports up to 3 sensors

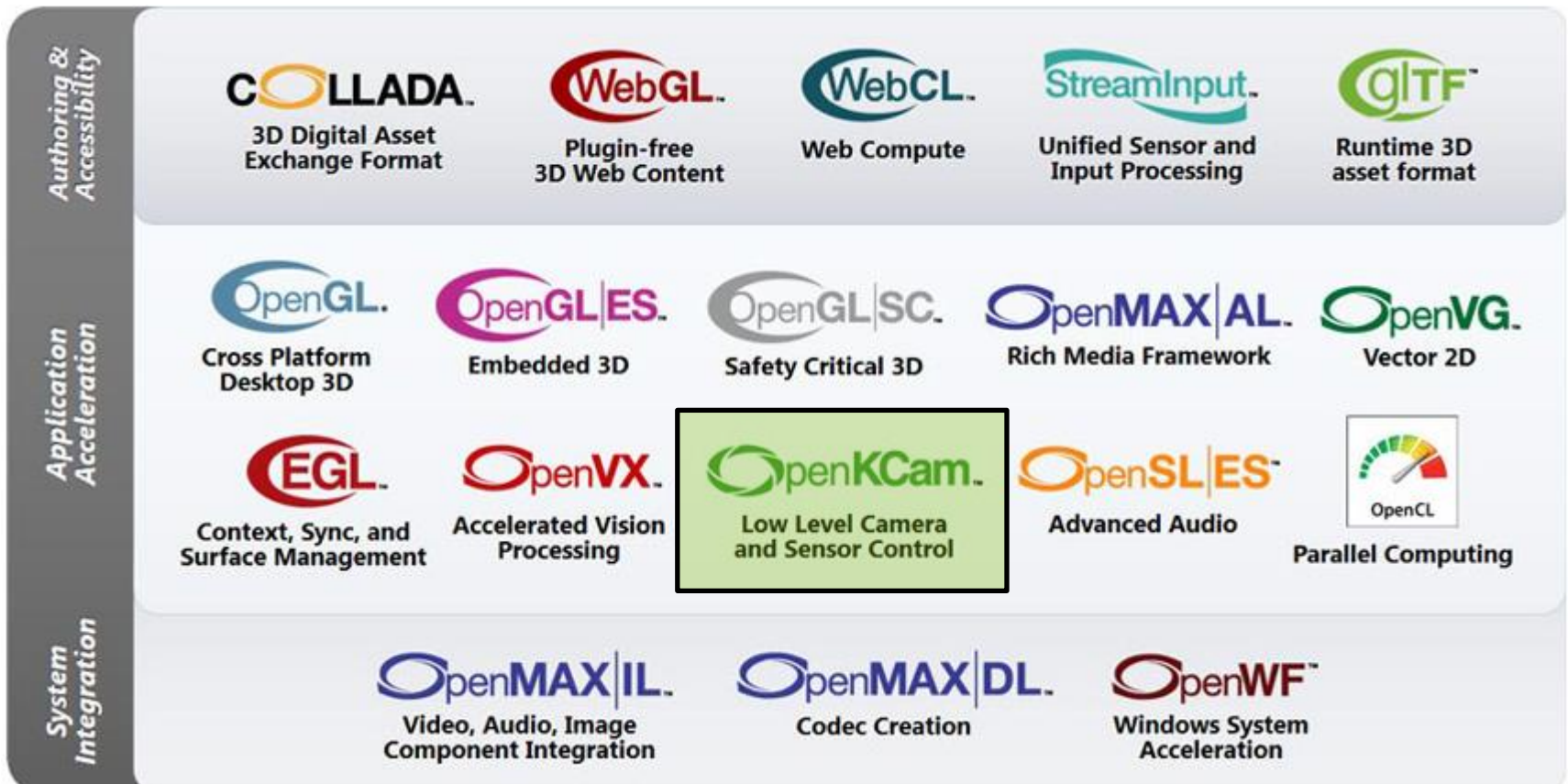
Skillset to provide custom solutions to cover the entire range of TX1-based visual computing products

JETPACK CAMERA API: LIBARGUS

”the image processing industry still lacks a camera API with low-level control of the camera sensor to generate the input image stream needed by cutting-edge computational photography and computer vision.”

Khronos OpenKCam Working Group, May 2013
(https://www.khronos.org/openkcam)

KHRONOS GROUP



JETPACK CAMERA API: LIBARGUS

Design Goals

Open Standard

Cross Platform

Low-level control of camera subsystem

Frame-based capture control

Metadata output for frame statistics

Multi-stream, multi-camera, multi-process support

Efficient GPU processing and API interop via EGLStreams

Extendable and backwards compatible

JETPACK CAMERA API: LIBARGUS

Coding Standards

Argus:: namespace

C++03

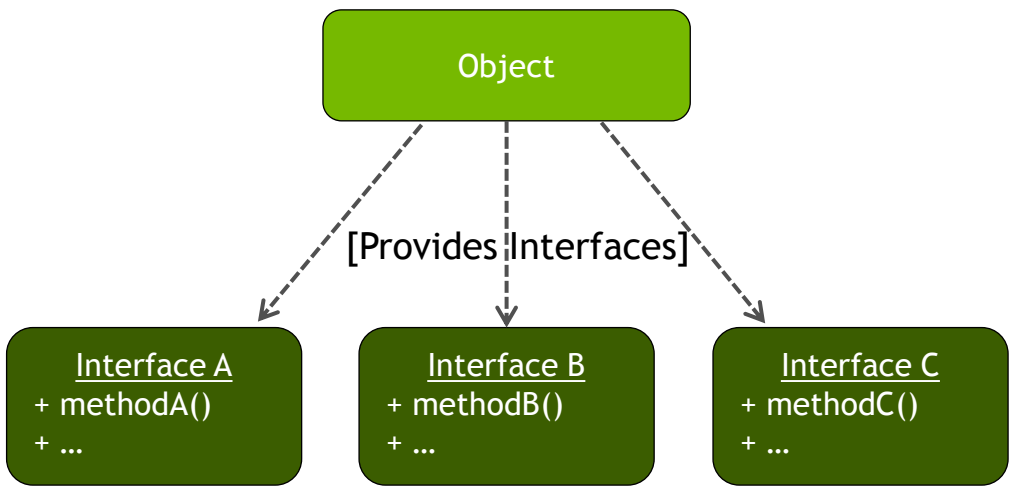
No exceptions

No RTTI

JETPACK CAMERA API: LIBARGUS

Objects and Interfaces

Objects do not have methods. All methods are provided by **Interfaces**.



JETPACK CAMERA API: LIBARGUS

Objects

Unique handle to API entity

All objects are **InterfaceProviders**

Two types of objects:

1. **Destructable**: created, owned, and destroyed by the client
2. Non-**Destructable**: children of other libargus objects; owned and destroyed by parent object.

JETPACK CAMERA API: LIBARGUS

Interfaces

Pure virtual class

Name prefixed with ‘I’ (ie. **IEvent**)

Identified by 128 bit UUID, **InterfaceID**

Acquired at runtime from an **InterfaceProvider** (ie. Object)

Valid throughout lifetime of object

Interfaces do not change once published*

New functionality added using new interfaces*

*Interfaces still subject to change before libargus 1.0 release

JETPACK CAMERA API: LIBARGUS

InterfaceProvider

```
class InterfaceProvider {
public :

    // Acquire an interface specified by 'interfaceId'.
    // Returns an instance of the requested interface, or NULL if the
    // interface is not supported by the object.
    virtual Interface* getInterface (const InterfaceID& interfaceId) = 0;

};
```

JETPACK CAMERA API: LIBARGUS

IRequest Interface Example

IRequest Interface:

```
// Request objects support the IRequest Interface, identified by IID_REQUEST.
DEFINE_UUID(InterfaceID, IID_REQUEST, eb9b3750,fc8d,455f,8e0f,91,b3,3b,d9,4e,c5);
class IRequest : public Interface
{
public:

     // Begin IRequest methods (truncated for example)
     virtual Status enableOutputStream(OutputStream* stream) = 0;
     virtual Status disableOutputStream(OutputStream* stream) = 0;
     …
};
```

Example Usage:

```
// Get IRequest interface from a Request object
IRequest* iRequest = static_cast<IRequest*>(request->getInterface(IID_REQUEST));

// Call a method using the IRequest interface
iRequest->enableOutputStream(stream);
```

JETPACK CAMERA API: LIBARGUS

Destructable Objects

```
// Destructable objects must be explicitly destroyed.
class Destructable {
public:
    virtual void destroy() = 0;
};
```

Requests are Destructable:

```
class Request : public InterfaceProvider, public Destructable
{
public:
    virtual Interface* getInterface (const InterfaceID& interfaceId) = 0;
    virtual void destroy() = 0;
};
```

JETPACK CAMERA API: LIBARGUS

Template Utilities

interface_cast<typename InterfaceT>(InterfaceProvider* obj)

Wraps **InterfaceProvider::getInterface()** with C++ casting semantics

Safe to call with NULL **InterfaceProvider**

UniqueObj<typename DestructableT>

Movable smart pointer used with **Destructable** objects

Calls **destroy()** method of wrapped object during destruction

JETPACK CAMERA API: LIBARGUS

Template Utilities

Before:

```
{
  Request* request =
    iCaptureSession->createRequest();
  if (!request)
    goto cleanup;

  IRequest* iRequest = static_cast<IRequest*>
     (request->getInterface(IID_REQUEST));
  if (!iRequest)
    goto cleanup;

  if (!iRequest->enableOutputStream(stream))
    goto cleanup;

  iCaptureSession->capture(request);

cleanup:
  if (request)
    request->destroy();
}
```

After:

```
{
  UniqueObj<Request> request
    (iCaptureSession->createRequest());

  IRequest* iRequest =
    interface_cast<IRequest>(request);
  if (!iRequest)
    RETURN_ERROR("Failed to create Request");

  if (!iRequest->enableOutputStream(stream))
    RETURN_ERROR("Failed to enable stream");

  iCaptureSession->capture(request.get());
}
```

JETPACK CAMERA API: LIBARGUS

Contents of libargus Release

libargus provided within JetPack Multimeda API package:

argus/include - API Headers

argus/docs - Documentation

argus/samples – Samples **(**including ‘*oneShot*’, source of following walkthrough)

argus/apps – Reference Camera Application

LIBARGUS API AND SAMPLE WALKTHROUGH

LIBARGUS API AND SAMPLE WALKTHROUGH

Outline

1) Establish connection to libargus driver
2) Select a camera device
3) Create a capture session to use the device
4) Create an output stream for image output
5) Create and configure a capture request
6) Submit the capture request

Sample source: **argus/samples/oneShot**

LIBARGUS API AND SAMPLE WALKTHROUGH

CameraProvider

Singleton instance establishes connection to libargus driver

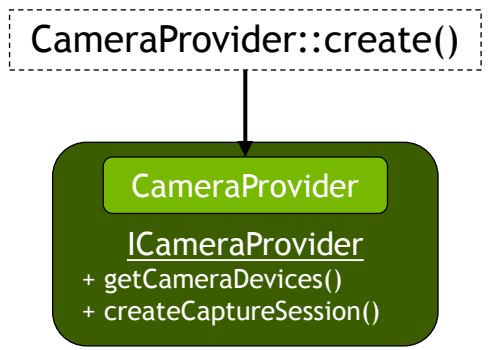
First object that any libargus application must create

```
static CameraProvider* CameraProvider::create(Status* status);
```

ICameraProvider provides access to **CameraDevices** and **CaptureSession** creation

```
class ICameraProvider : public Interface {
public:
    virtual Status getCameraDevices(std::vector<CameraDevice*>* devices) const = 0;
    virtual CaptureSession* createCaptureSession(CameraDevice* device, Status* status) = 0;
    …
};
```

```
// 1) Create CameraProvider to establish libargus driver connection.
UniqueObj<CameraProvider> cameraProvider(CameraProvider::create());
ICameraProvider* iCameraProvider = interface_cast<ICameraProvider>(cameraProvider);
if (!iCameraProvider)
    EXIT("Failed to establish libargus connection");
```



LIBARGUS API AND SAMPLE WALKTHROUGH

CameraDevice

Object representing a single camera device

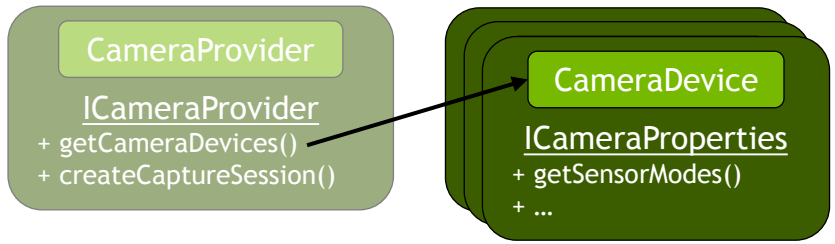
Child object owned by **CameraProvider**

ICameraProperties interface exposes device properties and capabilities

```
class ICameraProperties : public Interface {
public:
    virtual UUID getUUID() const = 0;
    virtual Status getSensorModes(std::vector<SensorMode*>* modes) const = 0;
    virtual uint32_t getMaxAeRegions() const = 0;
    virtual uint32_t getMaxAwbRegions() const = 0;
    virtual Range<int32_t> getFocusPositionRange() const = 0;
    virtual Range<float> getLensApertureRange() const = 0;
};
```

```
// 2) Query available CameraDevices from CameraProvider
std::vector<CameraDevice*> cameraDevices;
iCameraProvider->getCameraDevices(&cameraDevices);
if (cameraDevices.size() == 0)
    EXIT("No camera devices available");

// Use first available device
CameraDevice *selectedDevice = cameraDevices[0];
```



LIBARGUS API AND SAMPLE WALKTHROUGH

CaptureSession

Maintains exclusive connection to one or more **CameraDevices**

Creates **OutputStreams** and **Requests**

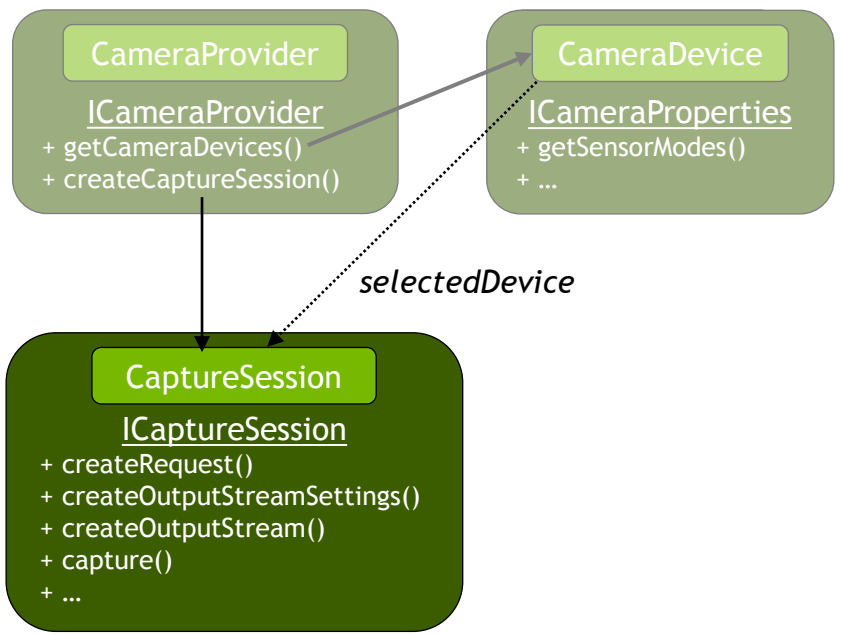
Submit capture requests and manage capture queue

Generates capture-related **Events**

One time or repeat capture methods

```
class ICaptureSession: public Interface {
public:
    virtual Request* createRequest(const CaptureIntent& intent = CAPTURE_INTENT_PREVIEW) = 0;
    virtual OutputStream* createOutputStream(const OutputStreamSettings* settings) = 0;
    virtual uint32_t capture(const Request* request, uint64_t timeout = TIMEOUT_INFINITE) = 0;
    virtual Status repeat(const Request* request) = 0;
    …
};
```

```
// 3) Create CaptureSession for selected CameraDevice
UniqueObj<CaptureSession> captureSession(iCameraProvider->createCaptureSession(selectedDevice));
ICaptureSession *iCaptureSession = interface_cast<ICaptureSession>(captureSession);
if (!iCaptureSession)
    EXIT("Failed to create CaptureSession");
```



LIBARGUS API AND SAMPLE WALKTHROUGH

OutputStream

Destination streams for capture request outputs

Creates EGLStream and connects as Producer endpoint

EGLStream consumer must connect to stream to consume frames

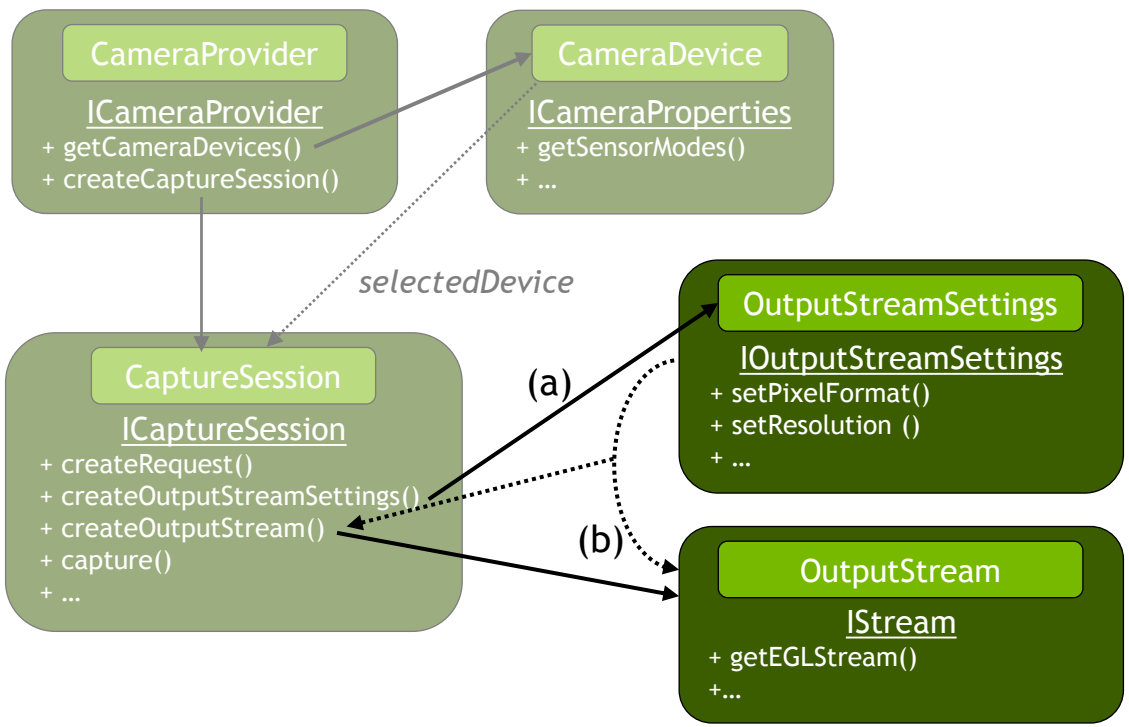
→ Omitted from this example

Creation parameters provided by transient **OutputStreamSettings** object

→ Stream attributes immutable once created

```
// 4a) Create and configure OutputStreamSettings
UniqueObj<OutputStreamSettings> streamSettings(iCaptureSession->createOutputStreamSettings());
IOutputStreamSettings *iStreamSettings = interface_cast<IOuputStreamSettings>(streamSettings);
iStreamSettings->setPixelFormat(PIXEL_FMT_YCbCr_420_888);
iStreamSettings->setResolution(Size(640, 480));

// 4b) Create OutputStream
UniqueObj<OutputStream> outputStream(iCaptureSession->createOutputStream(streamSettings.get()));
IStream* iStream = interface_cast<IStream>(outputStream);
if (!iStream)
    EXIT("Failed to create OutputStream");
```



LIBARGUS API AND SAMPLE WALKTHROUGH

Request

Contains all settings for a single capture

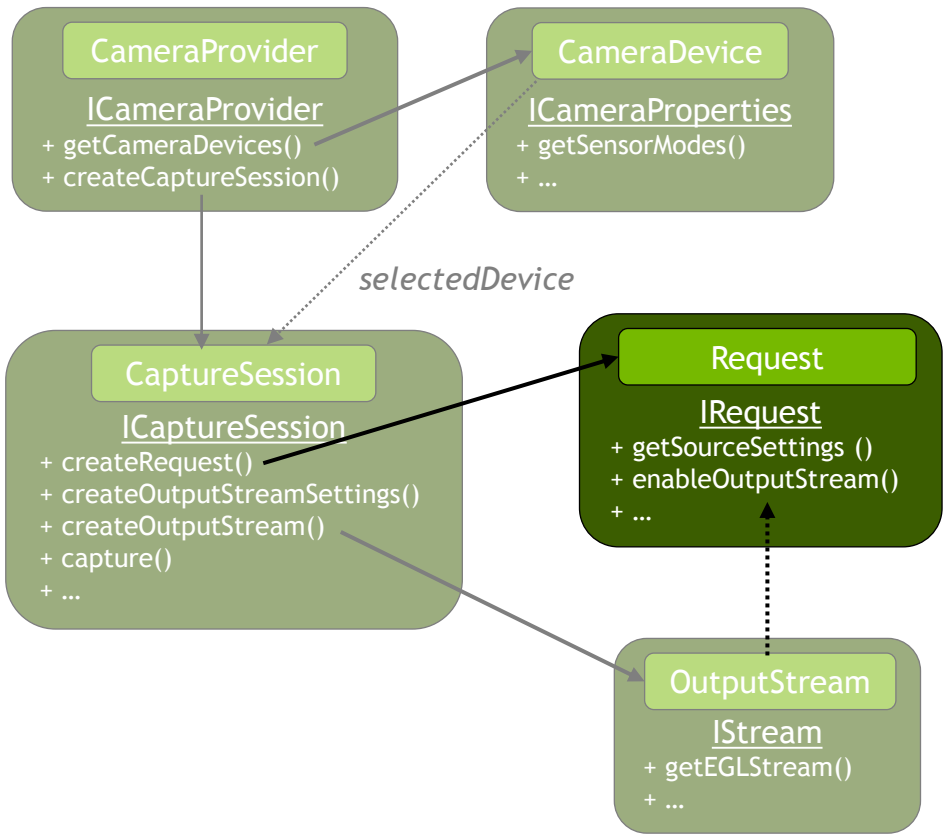
Child **InterfaceProviders** provide specialized configuration:

ISourceSettings – **CameraDevice** settings (eg. sensor mode)
IAutoControlSettings – ISP-based Autocontrol and processing settings
IStreamSettings – Per-**OutputStream** settings (eg. clip rect)

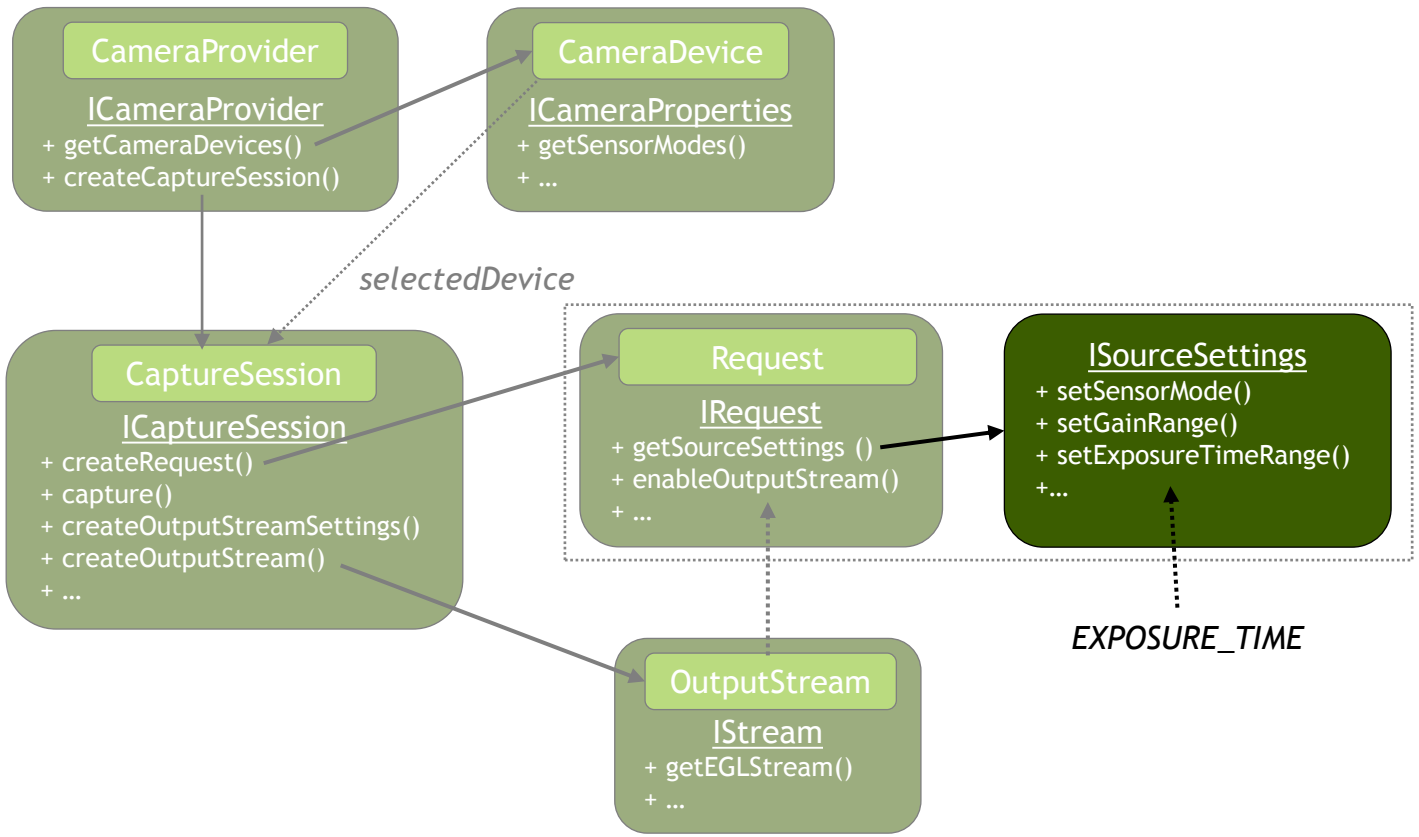
```
class IRequest : public Interface {
public:
     // Core IRequest settings
     virtual Status enableOutputStream(OutputStream* stream) = 0;
     virtual Status disableOutputStream(OutputStream* stream) = 0;
     …

     // Specialized configuration objects
     virtual InterfaceProvider* getSourceSettings() = 0;
     virtual InterfaceProvider* getAutoControlSettings() = 0;
     virtual InterfaceProvider* getStreamSettings(const OutputStream* stream) = 0;
     …
};
```

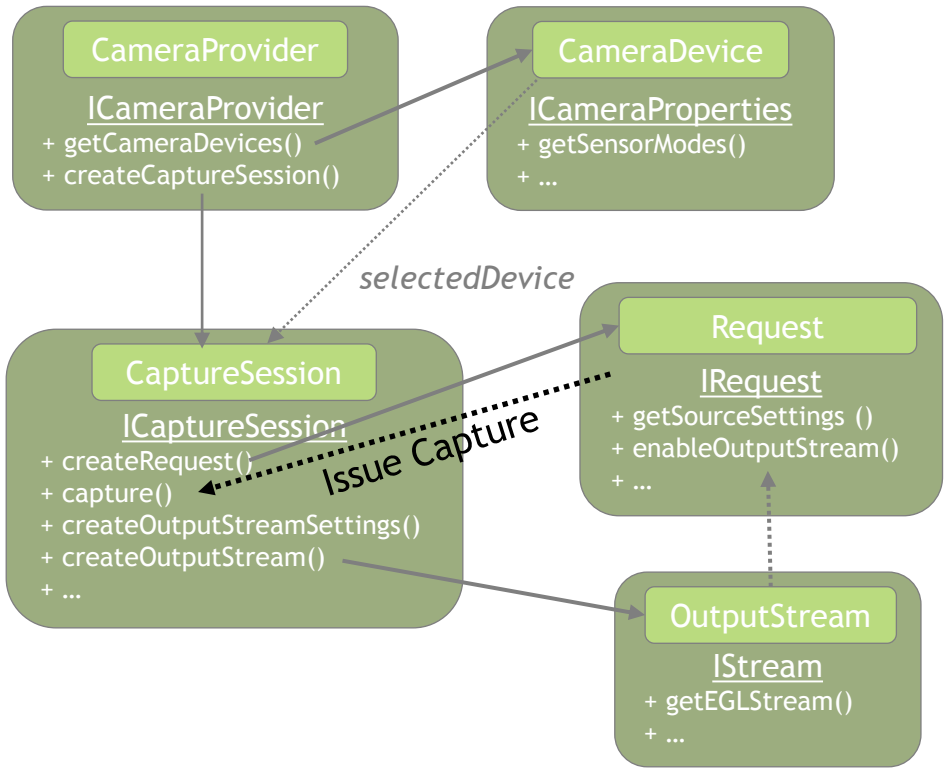
```
// 5) Create Request and enable the output stream
UniqueObj<Request> request(iCaptureSession->createRequest());
IRequest *iRequest = interface_cast<IRequest>(request);
iRequest->enableOutputStream(outputStream.get());
```



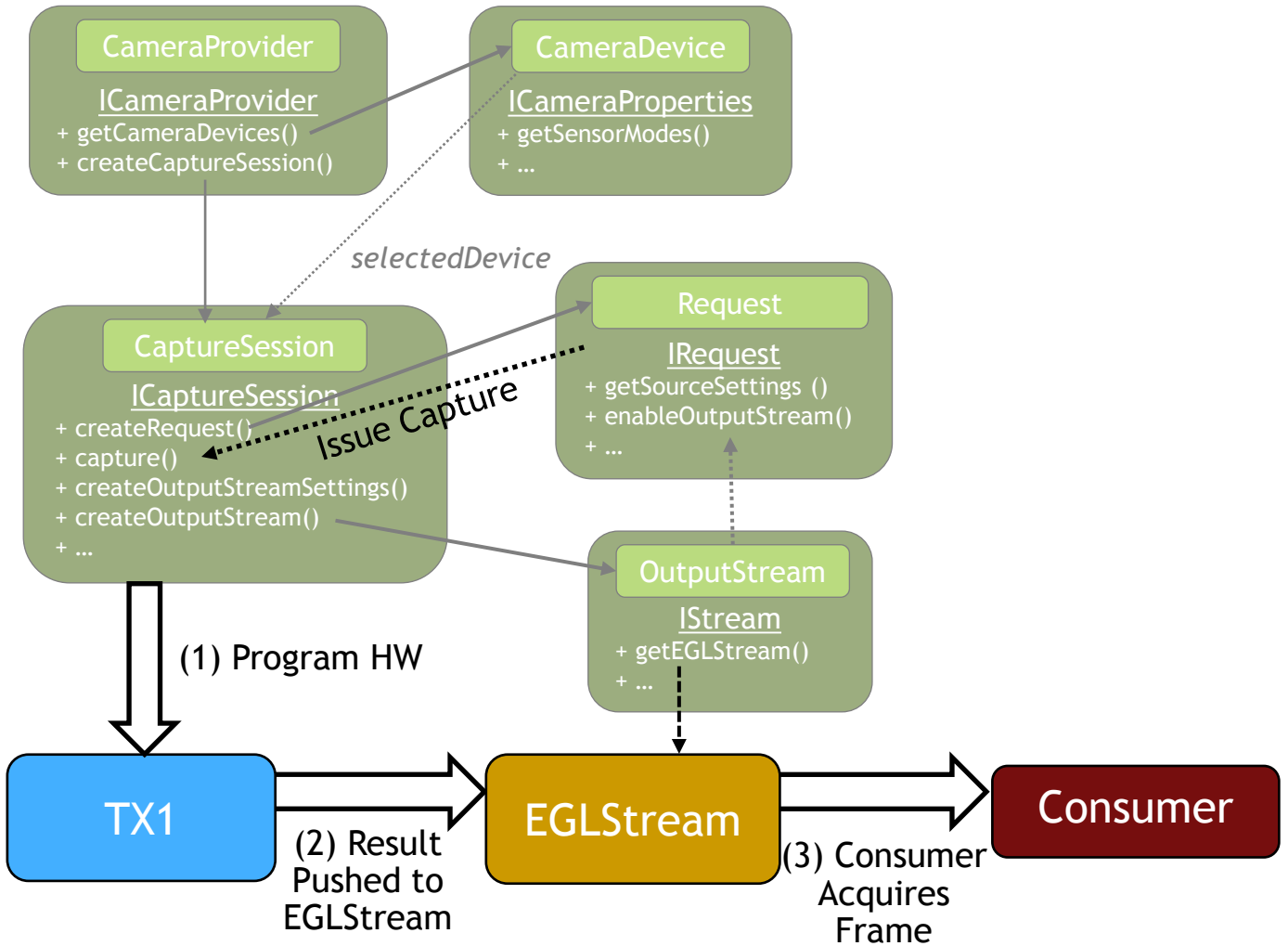
```
// 5b) Set the exposure time for the request.
ISourceSettings *iSourceSettings = interface_cast<ISourceSettings>(iRequest->getSourceSettings());
iSourceSettings->setExposureTimeRange(EXPOSURE_TIME);
```



```
// 6) Issue capture request
iCaptureSession->capture(request.get());
```



```
// 6) Issue capture request
iCaptureSession->capture(request.get());
```



EVENTS, METADATA, AND EXTENSIONS

EVENTS, METADATA, AND EXTENSIONS

Events

Event-generating objects expose the **IEventProvider** interface

Available event types depend on object providing the events

Events are pulled from an **IEventProvider** into **EventQueues**

Client controls which events are enabled for each queue

```
class IEventProvider : public Interface {
public:
     virtual Status getAvailableEventTypes(std::vector<EventType>* types);
     virtual EventQueue* createEventQueue(const std::vector<EventType>& eventTypes);
     virtual Status waitForEvents(EventQueue* queue, uint64_t timeout = INFINITE);
     …
};
```

EVENTS, METADATA, AND EXTENSIONS

Capture Events

CaptureSession the only **EventProvider**, generates capture-related events:

ERROR – Error occurred during capture

CAPTURE_STARTED – Signals start of exposure (ie. shutter open)

CAPTURE_COMPLETE – Capture has completed.

Capture requests identified by increasing ID

```
class ICaptureSession : public Interface {
    // Capture ID assigned at time of request.
    virtual uint32_t capture(const Request* request, uint64_t timeout = TIMEOUT_INFINITE) = 0;
};
```

```
class IEvent : public Interface {
public:
    virtual uint32_t getCaptureId() const = 0;
};
```

EVENTS, METADATA, AND EXTENSIONS

Metadata

Completed captures accompanied by **CaptureMetadata**

Provides report of settings used for the capture

Settings provided by a **Request** are not guaranteed to be met

Statistics from ISP and/or other sources may be included with metadata

Two ways to read metadata:

1. Events
2. Embedded EGLStream data

EVENTS, METADATA, AND EXTENSIONS

Metadata via Events

CAPTURE_COMPLETE events expose the **IEventCaptureComplete** interface:

```
class IEventCaptureComplete : public Interface {
public:
    virtual const CaptureMetadata* getMetadata() const = 0;
};
```

Reading events and metadata:

```
while (running) {
    // Wait for a new capture complete event and extract metadata.
    iEventProvider->waitForEvents(queue);
    const IEventCaptureComplete* iEvent =
        interface_cast<const IEventCaptureComplete>(iQueue->getNextEvent(););
    const CaptureMetadata* metadata = iEvent->getMetadata();

    // Print out various metadata details.
    ICaptureMetadata *iMetadata = interface_cast<ICaptureMetadata>(metadata);
    cout << "Capture ID: " << iMetadata->getCaptureId() << endl;
    cout << "Color Saturation: " << iMetadata->getColorSaturation() << endl;
    cout << "Exposure Time: " << iMetadata->getSensorExposureTime() << endl;
};
```

EVENTS, METADATA, AND EXTENSIONS

Statistics-Driven Capture Control

Application-layer ISP and sensor control capture thread using events:

```
while (running) {
    // Wait for a new capture complete event and extract metadata.
    iEventProvider->waitForEvents(queue);
    const IEventCaptureComplete* iEvent =
        interface_cast<const IEventCaptureComplete>(iQueue->getNextEvent());
    const CaptureMetadata* metadata = iEvent->getMetadata();

    // Modifies the request settings based on metadata results of the previous frame.
    modifyRequestUsingPreviousMetadata(metadata, request);

    // Submit the next capture request.
    iCaptureSession->capture(request);
};
```

Statistics-driven capture control samples:

userAutoExposure and **userAutoWhiteBalance**

EVENTS, METADATA, AND EXTENSIONS

Extensions

Add functionality to core libargus API

Non-standard or hardware/platform-dependent features (ie. BayerSharpnessMap)

Luxuries/conveniences (ie. FaceDetect)

Extension definitions in “**include/Argus/Ext**”, **Ext::** namespace, identified by UUID

```
class ICameraProvider : public Interface {
public:
    virtual bool supportsExtension(const ExtensionName& extension) const = 0;
};
```

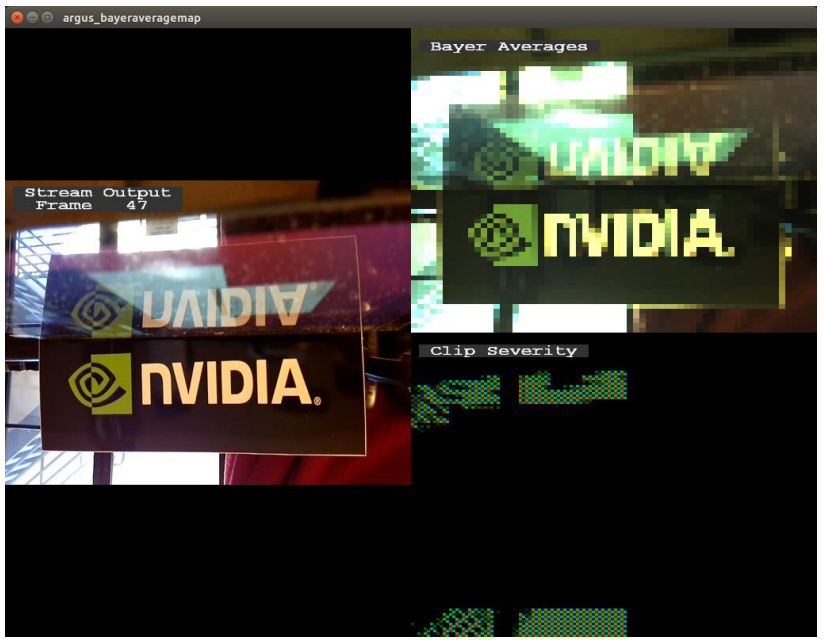
```
DEFINE_UUID(ExtensionName, EXT_FACE_DETECT, eb9b3750,fc8d,455f,8e0f,91,b3,3b,d9,4e,c5);
namespace Ext {
  class IFaceDetectCaps : public Interface {…}
  class IFaceDetectSettings : public Interface {…}
  class IFaceDetectMetadata : public Interface {…}
}
```

EVENTS, METADATA, AND EXTENSIONS

Extensions (ISP Statistics)

BayerSharpnessMap – Image sharpness metrics

BayerAverageMap – Bayer averages and clipping statistics



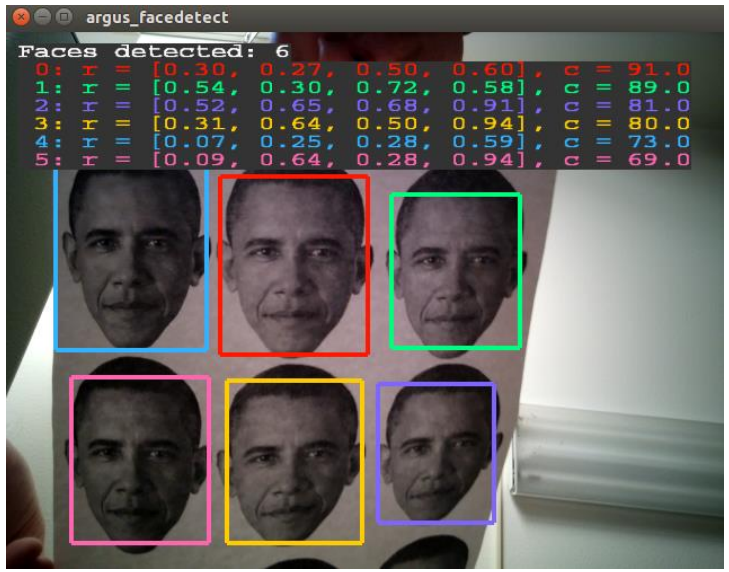
EVENTS, METADATA, AND EXTENSIONS

Extensions

DeFog – Minimize fog effects

SensorPrivateMetadata – Generic access to sensor-embedded metadata

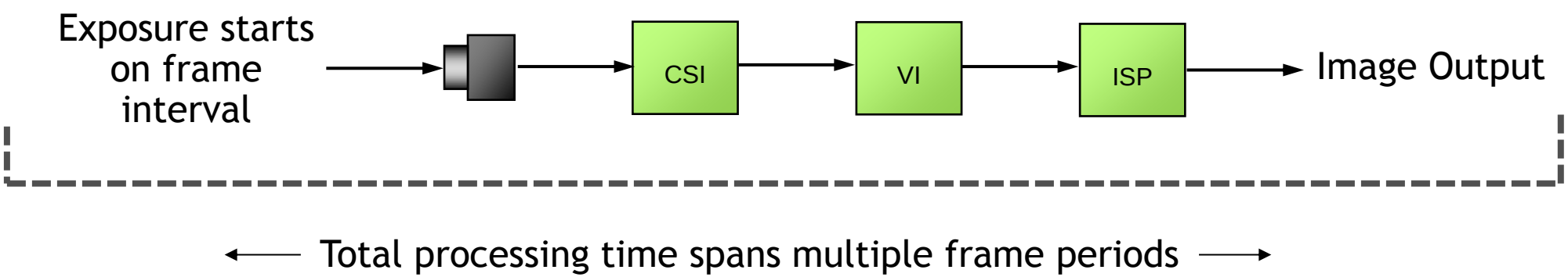
FaceDetect – Face detection



EFFICIENT CAMERA APPLICATIONS

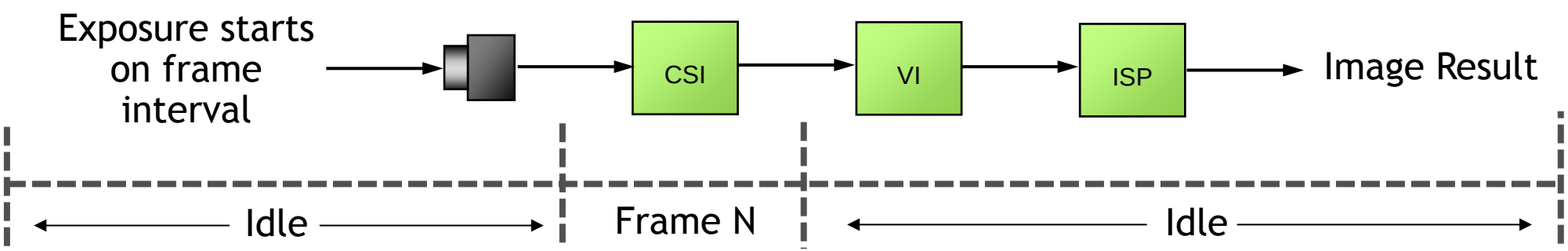
EFFICIENT CAMERA APPLICATIONS

Capture Pipeline



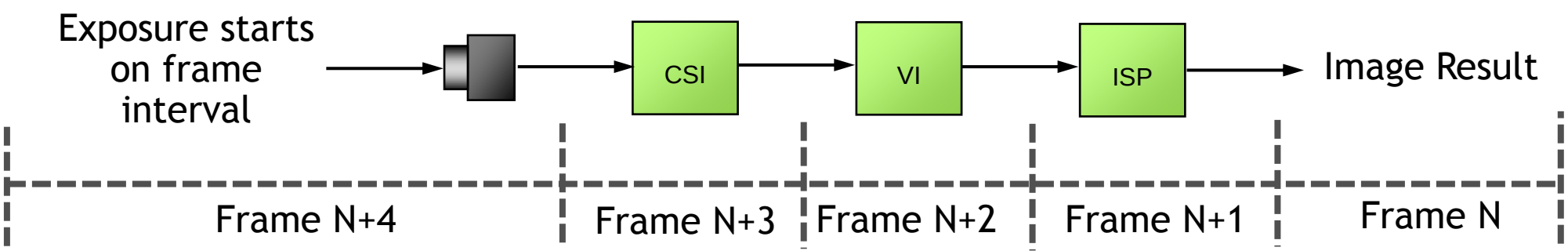
EFFICIENT CAMERA APPLICATIONS

Capture Pipeline - Single Capture



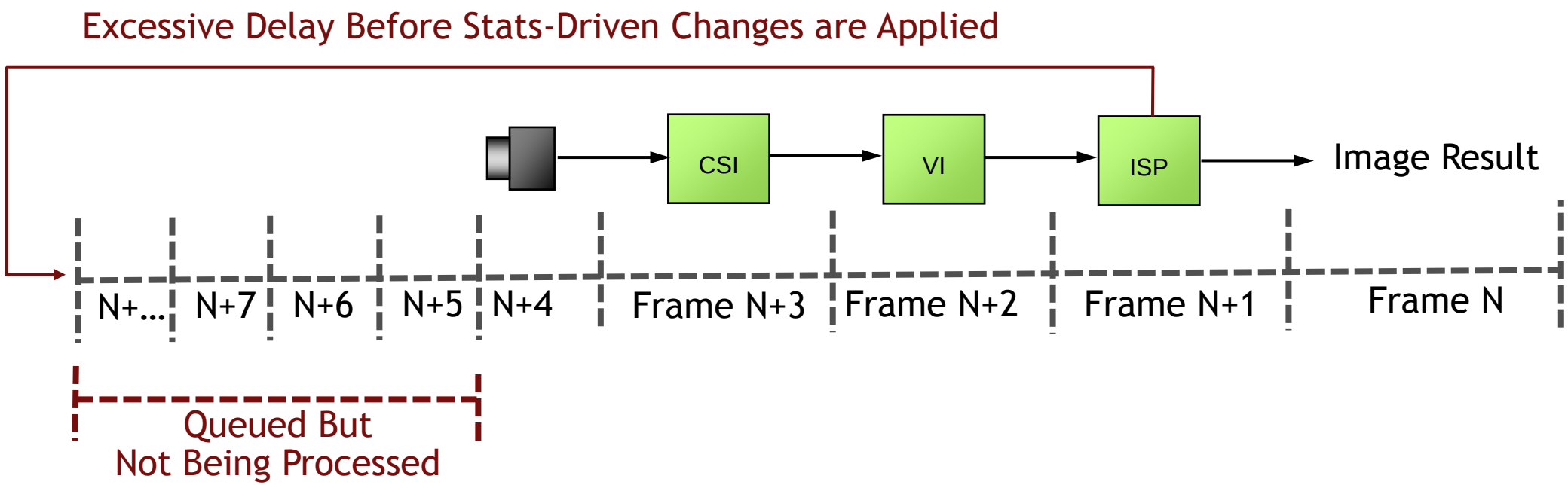
EFFICIENT CAMERA APPLICATIONS

Capture Pipeline – Concurrent Processing



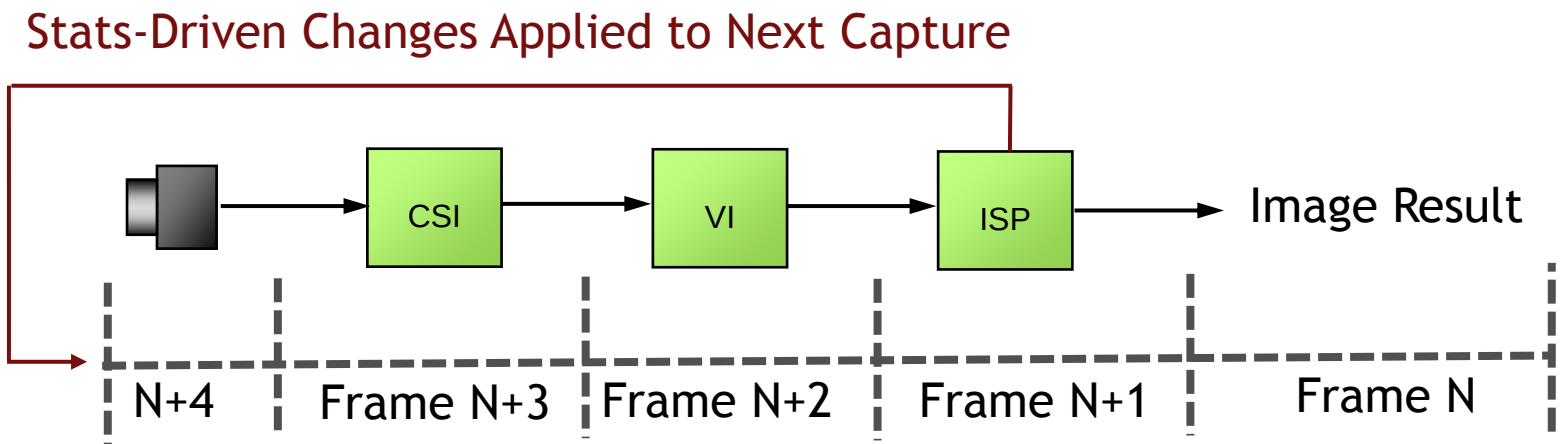
EFFICIENT CAMERA APPLICATIONS

Capture Pipeline – Excessive Requests



EFFICIENT CAMERA APPLICATIONS

Capture Pipeline - Optimal Requests



EFFICIENT CAMERA APPLICATIONS

Repeat Capture Methods

Solution: use **ICaptureSession::repeat()** capture methods

→ Puts libargus driver in control of maintaining the optimal queue depth

Replace current repeat request with another call to **repeat()**

Repeat captures stopped by calling **stopRepeat()**

→ Returns range of capture IDs generated by last repeated request

```
class ICaptureSession: public Interface {
public:
    virtual Status repeat(const Request* request) = 0;
    virtual Status repeatBurst(const std::vector<const Request*>& requestList) = 0;
    virtual bool isRepeating() const = 0;
    virtual Range<uint32_t> stopRepeat() = 0;
};
```

EFFICIENT CAMERA APPLICATIONS

Statistics-Driven Capture Control, Updated

```
// Start the initial repeat capture requests.
iCaptureSession->repeat(request);

while (running) {
    // Wait for new capture complete events, extract metadata from last completed capture.
    iEventProvider->waitForEvents(queue);
    uint32_t numEvents = iQueue->getSize();
    const IEventCaptureComplete* iEvent =
        interface_cast<const IEventCaptureComplete>(iQueue->getEvent(numEvents - 1));
    const CaptureMetadata* metadata = iEvent->getMetadata();

    // Modifies the request settings based on metadata results of the previous frame.
    modifyRequestUsingPreviousMetadata(metadata, request);

    // Replace the repeat capture request with the updated settings.
    iCaptureSession->repeat(request);
};
```

IMAGE CONSUMPTION AND EGLSTREAMS

EGLSTREAMS

Overview

Stream of images between two APIs: producer and consumer

Provides:

- Buffer allocation
- Synchronization
- State Management
- Embedded Metadata

Specifications: https://www.khronos.org/registry/egl/

EGLSTREAMS

Existing EGLStream Consumers

OpenGL / OpenGL ES - GPU Rendering

→ https://www.khronos.org/opengl/

GStreamer – Video Encoding

→ https://gstreamer.freedesktop.org/

CUDA – GPU Compute

→ https://developer.nvidia.com/cuda-zone

IMAGE CONSUMPTION

EGLStream::FrameConsumer

Written specifically for and included with libargus

Headers: **argus/includes/EGLStream**

Offers JPEG encoding and native buffer compatibility

Within **EGLStream::** namespace

Uses libargus types and object/interface model

Highly integrated with Argus

→ no knowledge of EGLStreams required

IMAGE CONSUMPTION

FrameConsumer

Static creation/connection to **Argus::OutputStream** or EGLStream handle

```
class FrameConsumer : public InterfaceProvider, public Destructable
{
    static FrameConsumer* create(Argus::OutputStream* stream);
    static FrameConsumer* create(EGLDisplay display, EGLStream stream);
};
```

Acquires **Frames** from the stream

```
class IFrameConsumer : public Interface
{
     virtual Frame* acquireFrame(uint64_t timeout = TIMEOUT_INFINITE) = 0;
};
```

IMAGE CONSUMPTION

FrameConsumer: Frame

Provides **Image** data and EGLStream frame details

```
class IFrame : public Interface
{
    virtual Image* getImage() const = 0;
    virtual uint64_t getNumber() const = 0;
    virtual uint64_t getTime() const = 0;
};
```

And embedded **Argus::CaptureMetadata**

```
class IArgusCaptureMetadta : public Interface
{
    virtual Argus::CaptureMetadata* getMetadata() const = 0;
};
```

IMAGE CONSUMPTION

FrameConsumer: IImageJPEG

IImageJPEG provides JPEG encoding:

```
class IImageJPEG : public Interface
{
     virtual Status writeJPEG(const char* filename) = 0;
};
```

IMAGE CONSUMPTION

FrameConsumer Example (oneShot JPEG Encoding)

```
// Create output stream.
UniqueObj<OutputStream> stream(iSession->createOutputStream(streamSettings));

// Create and connect FrameConsumer to output stream.
UniqueObj<EGLStream::FrameConsumer> consumer(EGLStream::FrameConsumer::create(stream));
EGLStream::IFrameConsumer *iConsumer =  interface_cast<EGLStream::IFrameConsumer>(consumer.get());

// Submit capture request outputting to stream
iSession->capture(request);

// Acquire a Frame from the consumer.
UniqueObj<EGLStream::Frame> frame(iConsumer->acquireFrame());
EGLStream::IFrame *iFrame = interface_cast<EGLStream::IFrame>(frame);

// Get the Argus::CaptureMetadata embedded in the frame.
EGLStream::IArgusCaptureMetadata *iArgusMetadata = interface_cast<IArgusCaptureMetadata>(frame);
Argus::CaptureMetadata* metadata = iArgusMetadata->getMetadata();

// Get the Image from the Frame.
EGLStream::Image *image = iFrame->getImage();

// Use the JPEG interface to encode and write the JPEG file.
EGLStream::IImageJPEG *iImageJPEG = interface_cast<EGLStream::IImageJPEG>(image);
iImageJPEG->writeJPEG("filename.jpg");
```

Q: ”Are EGLStreams really the *only* way to consume outputs from libargus?”

- You (maybe)

Q: ”Are EGLStreams really the *only* way to consume outputs from libargus?”

- You (maybe)

A: No*

EGLSTREAMS

Limitations

Depends on EGL

Less Control

Does not support all use cases

Requires consumer API that supports EGLStreams

→ V4L2 does not support EGLStreams

LIBARGUS IMAGE CONSUMPTION

Without EGLStreams?

Currently working on native buffer support for libargus (2017)

Temporary solution: **IImageNativeBuffer** FrameConsumer interface

→ Copies EGLStream images to native NvBuffers

→ NvBuffer definitions and utilities: **include/nvbuf_utils.h**

```
class IImageNativeBuffer : public Interface
{
     virtual Status copyToNvBuffer(int bufferFd) = 0;
};
```

LIBARGUS SAMPLES

LIBARGUS SAMPLES

Basic Samples

oneShot:

Most basic Argus sample, used for walkthrough today.

Performs single capture and writes JPEG (FrameConsumer)

userAutoExposure and **userAutoWhiteBalance:**

Uses metadata, BayerHistogram and Ext::BayerAverageMap extension.

Demonstrates application-layer sensor and capture control using image metadata/statistics

LIBARGUS SAMPLES

EGLStream Consumer Samples

openglBox:

OpenGL consumer renders camera stream onto a 3D spinning cube.

gstVideoEncode:

GStreamer consumer pipeline encodes and outputs h264 video file from stream

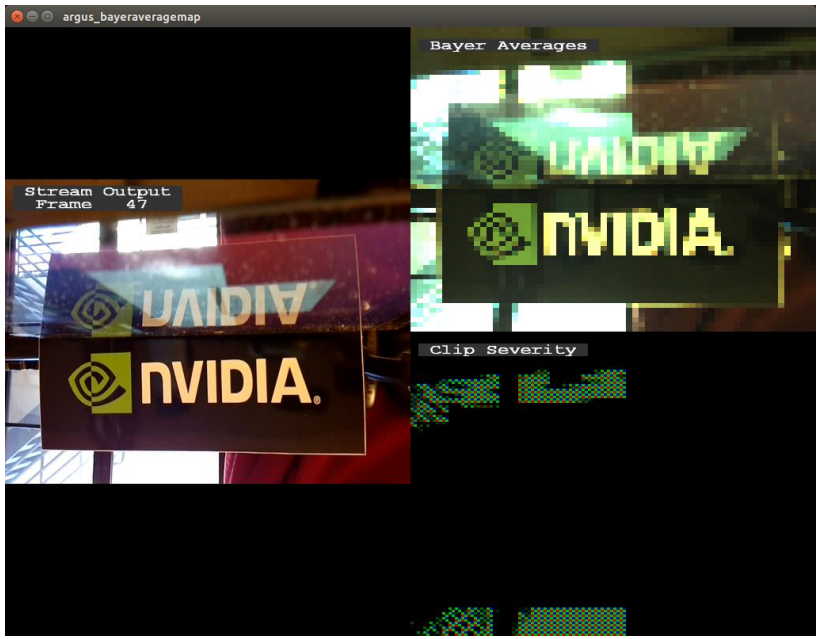
cudaHistogram:

Uses CUDA consumer to compute histogram stats for each stream frame

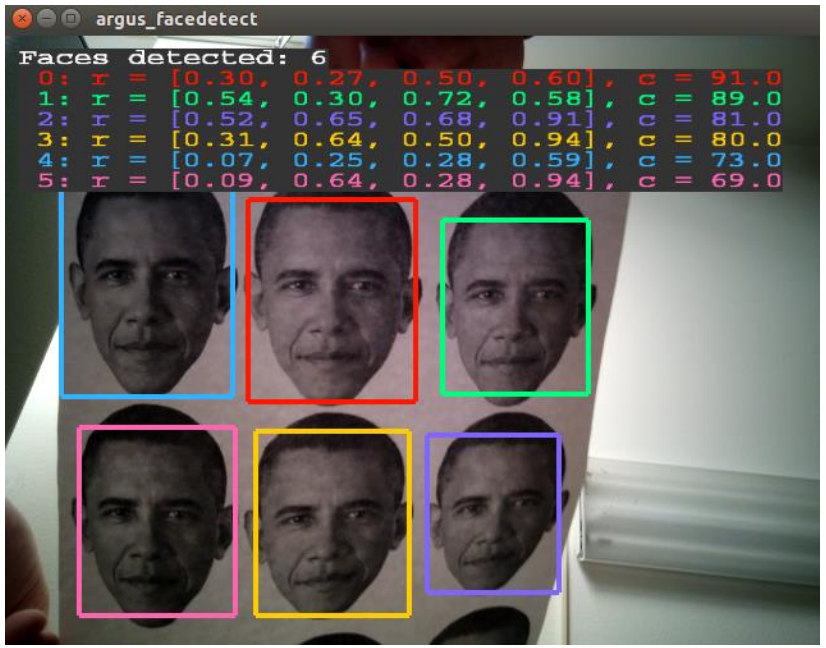
LIBARGUS SAMPLES

Metadata Visualizations (OpenGL)

bayerAverageMap
(Ext::BayerAverageMap)



faceDetect
(Ext::FaceDetect)



LIBARGUS SAMPLES

Multi-Stream Samples

multiStream:

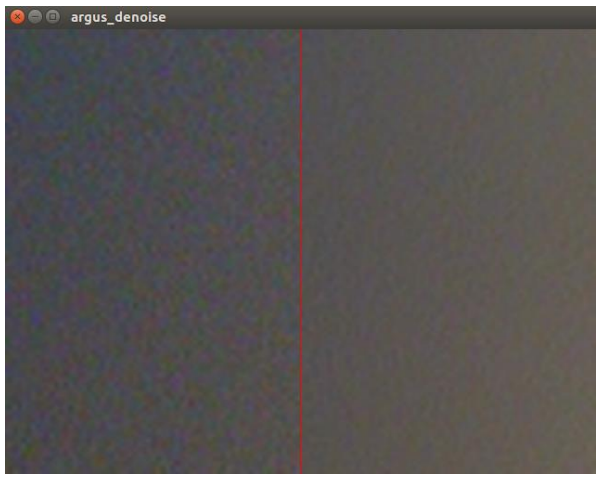
Simultaneous preview (OpenGL) and still capture (JPEG) streams

Uses burst captures for reduced still capture frequency

denoise:

Side-by-side comparison of denoise effects (OpenGL)

Uses per-stream Request settings (IStreamSettings) to disable denoise for one of the two streams.



LIBARGUS SAMPLES

Multi-Sensor Samples

multiSensor (requires 2 sensors)**:**

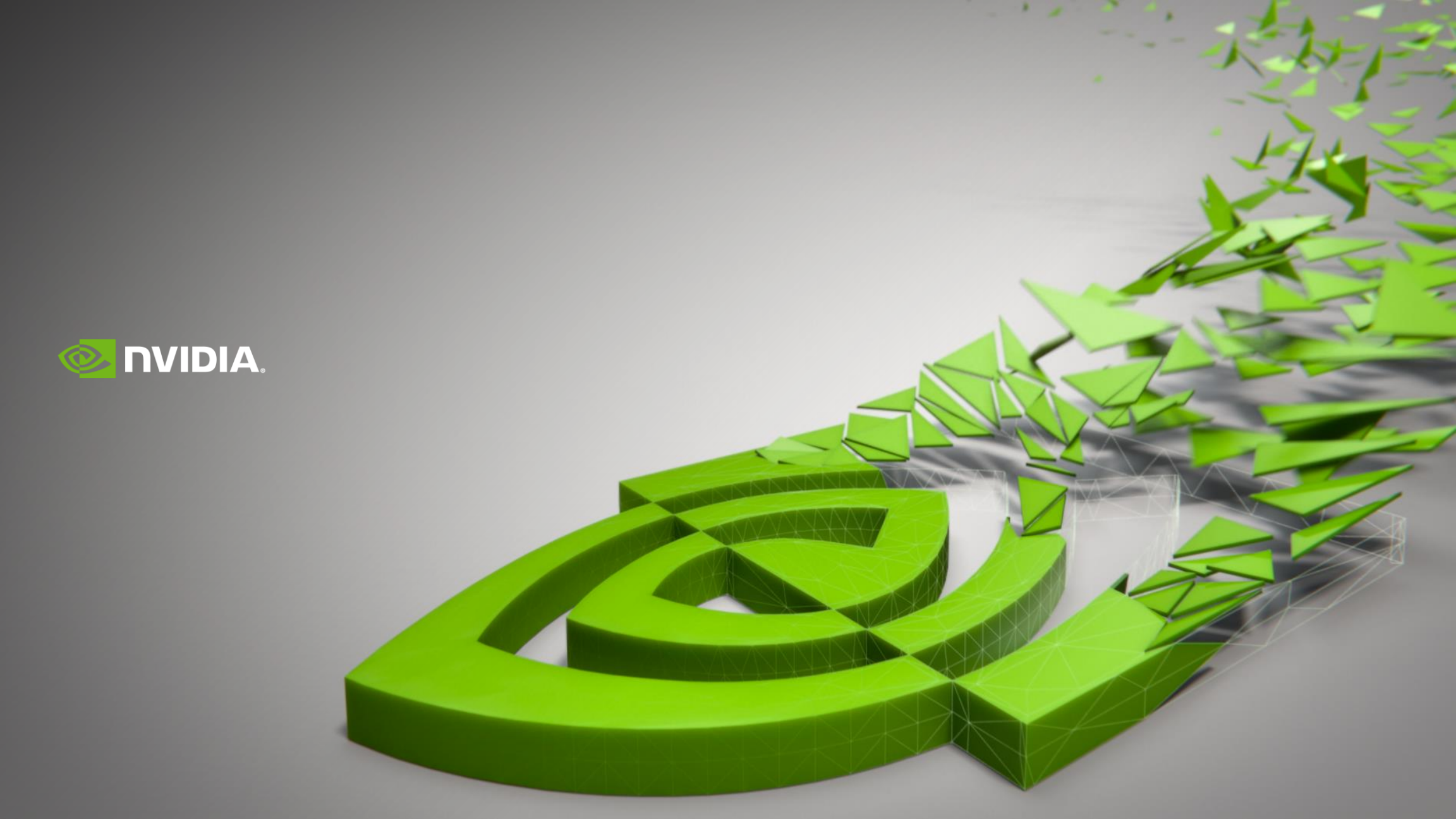
Uses two CaptureSessions, one per sensor.

One sensor/session is used for OpenGL preview, the other for JPEG captures.

syncSensor (requires stereo/synchronized sensors)

Single CaptureSession opened with stereo sensor pair

Uses CUDA to compute stereo disparity between the two streams.


NVIDIA