OpenKCam v1.0

This document is an Editor’s draft. Please do not cite this document.

Editor: Alejandro Troccoli

Editor: Kari Pulli

Table of Contents

1. Introduction . . . 2
  1.1. Goals . . . 2
  1.2. Terminology . . . 2
2. Object model . . . 3
3. Extensions to the API . . . 3
  3.1. Permitted modifications to the code . . . 4
  3.2. Extending supported interfaces types . . . 4
  3.3. Extending Supported Object Types . . . 4
4. Image capture and processing . . . 4
  4.1. Requests and state management . . . 5
  4.2. Events . . . 5
  4.3. The control and capture pipeline . . . 5
  4.4. Statistics . . . 10
5. Events architecture . . . 10
  5.1. CaptureSession methods . . . 11
6. Structures . . . 12
  6.1. SensorMode . . . 12
7. Standard objects . . . 13
  7.1. CameraProvider . . . 13
  7.2. CameraDevice . . . 14
  7.3. CaptureSession . . . 14
  7.4. Request . . . 14
  7.5. Frame . . . 15
  7.6. Event . . . 15
8. Standard interfaces . . . 15
  8.1. ICameraDevice . . . 15
  8.2. ICameraProvider . . . 15
  8.3. ICameraProperties . . . 16
  8.4. IControlSettings . . . 18
  8.5. ICaptureSession . . . 24
  8.6. IFrame . . . 28

8.7. IRequest 28
8.8. IStreamProcessingSettings 29
8.9. IStream 29
8.10. List of items for future revision 30

1. Introduction

Cameras have become ubiquitous: they can be found in mobile phones, tablets, and automobiles. OpenKCam is an API for developing both simple and advanced applications using cameras, providing per-frame control of the capture and processing pipelines of the camera system.

1.1. Goals

The OpenKCam API should provide means to:

- Create a simple camera application with still capture and video recording capabilities.
- Provide per-frame control over the sensor, flash, and lens.
- Provide system-wide timestamping of the captured images to synchronize with other sensors (such as an Inertial Measurement Unit, an IMU).
- Enable multiple outputs with different formats and resolutions.
- Provide access to the data computed by the Image Signal Processor (ISP), such as histograms and sharpness maps.
- Extend the API to support new sensor technologies and image processing algorithms.

1.2. Terminology

The following terms are commonly used in this document:

Camera: A camera consists of an imaging sensor, a lens, and optionally a flash.

Imaging Sensor: A sensor refers to the imaging device consisting of a two-dimensional array of wells that captures photons. The output of the sensor is a two-dimensional array of integer values within a given range. These values are related to the number of photons captured in each well. In addition, a sensor can have a color-filter array that filters the wavelengths of visible light captured by each well.

RAW Image: A RAW image is the unprocessed output of the imaging sensor. The format, size, and bit-depth of the values in the RAW image depend on the sensor.

Image processing pipeline: A set of stages that transform the image data. The first stage of the image processing pipeline takes a RAW image as input.

Rendered Image: A rendered image is the output of the last stage of image processing pipeline.

Auto control loops: The auto-control loops are algorithms that provide the values to be used for exposure, focus, and white-balance. If the values are updated from the latest image, the updated values affect some future frame.

2. Object model

OpenKCam is a C++ API that provides two types of abstractions: **objects** and **interfaces**. An **object** represents both a set of resources that are assigned for a well-defined set of tasks, and the state of these resources. An object's type is defined when it is created. In addition, the object type determines the set of tasks it can perform. An **interface** is an abstraction of a set of related features that a certain object provides. An interface has a type which determines the exact set of methods of the interface. We can define the interface itself as a combination of its type and the object to which it is related. An **interface ID** identifies an interface type and is used within the source code to identify it.

Objects and **interfaces** are tightly related, an object exposes one or more interfaces, all of which have different interface types—that is, an object may contain at most one interface of each interface type. A given interface instance is exposed by exactly one object. The application controls the object's state and performs the object operations exclusively through the interfaces it exposes. Thus, the object itself is a completely abstract notion, (**TODO** verify this is the case) and has no actual representation in code, yet it is very important to understand that behind every interface stands an object. The relationship between object types and interface types is that the object type determines the types of interfaces that may be exposed by objects of this type. Each object type definition in this document includes a detailed list of the interfaces for that object.

```
class Object
{
public:
    virtual Interface * GetInterface( int32_t interfaceId ) = 0;
    virtual void Destroy() = 0;

protected:
    Object();
    virtual ~Object() {};

    Object( const Object & other );
    Object& operator=( const Object & other );
};
```

An object's lifetime is the time between the object's creation and its destruction. The application explicitly performs both object creation and destruction, as will be explained later in this document.

3. Extensions to the API

The OpenKCam API has a flexible architecture to enable extensions to the API. An extended API is defined as one that provides functionality additional to that defined by the specification, yet still conforms to the specification.

The main principles of the extensibility mechanism are:

- Any application written to work with the standard API will still work, unchanged, on the extended API.
- For an application that makes use of extensions, it will be possible and simple to identify cases where these extensions are not supported, and thus to degrade its functionality gracefully.

Possible extensions may include vendor-specific extensions, as well as future versions of OpenKCam.

3.1. Permitted modifications to the code

The OpenKCam header files shall not be edited. Any vendor-specific extensions to the API shall reside in header files other than the OpenKCam header files.

3.2. Extending supported interfaces types

An extended API may introduce new interface types and expose these interfaces on either existing object types or on extended object types. An extended API may also expose standard interfaces on standard / extended object types that do not normally require exposing these interfaces.

The extended interfaces will be defined in a manner similar to standard interfaces. The extended interface types will have unique IDs, generated by the extension provider. Note that the extended API may not alter standard interfaces or apply different semantics on standard interfaces, even if the syntax is preserved. Functions may not be added to any of the interfaces defined in the specification. To do that, a new interface which includes the desired standard interface must be defined, along with a new interface ID which must be generated. It is also highly recommended that whenever an interface’s signature changes (even slightly), a new interface ID will be generated, and the modified interface will be considered a new one. This allows applications already written to still work with the original interface.

3.3. Extending Supported Object Types

An extended API may introduce new object types to those specified in the standard API. The extended objects may expose either standard or extended interface types. When standard interfaces are exposed, they must still behave as specified. However, the extended API may provide extended interface types with different semantics.

4. Image capture and processing

OpenKCam divides the image capture and processing pipeline into two:

- **The control and capture pipeline** that sets up the image sensor and other camera devices such us lens and flash. It also includes the **auto control** loops that compute suitable exposure, gain, focus and white-balance values. The output of the control and capture pipeline is a RAW image that is then delivered to one or more image processing pipelines. There is a single control and capture pipeline per camera.

- **The image processing pipeline** which produces a rendered image from a RAW image. OpenKCam allows for multiple image processing pipelines working on the same RAW image, each producing a different rendered image. The rendered images may differ in terms of size, format, and pipeline configuration.

4.1. Requests and state management

The configuration of the above pipelines is done through capture requests. A capture request completely defines the settings to be used at every stage, from capture to image processing. As a result, there is no global state. Instead, state moves through the pipeline with a request. The result of processing a request will be one or more frames. A frame is an object that encapsulates the image data plus the request settings and the actual result settings.

OpenKCam will keep a first-in first-out (FIFO) queue of requests. A request submitted by the client goes to the end of the queue. To enable video streaming without having to submit a request per frame, OpenKCam has a repeating request slot. The client can request to repeat a request. In this case, the request is placed in the repeat slot instead of the FIFO. Whenever the FIFO is empty, the OpenKCam implementation will check if there is a request in the repeating slot, and if that is the case, submits a copy of the request for processing. As a result, request in the FIFO have higher priority than requests in the repeating slot.

In addition, OpenKCam also defines the concept of grouped requests in a burst. A burst is simply an ordered set of requests that will be executed as an atomic unit. This is in particular meaningful when the client submits a repeating burst: if the client submits a request to the FIFO queue while a repeating burst is being processed, the request in the FIFO will only begin execution once all the requests in the bursts have been completed.

4.2. Events

Events provide an asynchronous mechanism for notifications to the OpenKCam client about the progress of a request. Any stage of the image processing pipeline can produce output data that is delivered with an event. For example, the Histogram stage can trigger an event once the histogram has been computed. This event could happen far in advance of the final image output being generated.

OpenKCam defines a set of event points that are available to the client. By default, all event points are disabled, meaning they will not generate events. A client that wants to receive an event notification needs to create an *EventQueue* bound to a particular event point.

4.3. The control and capture pipeline

The control and capture pipeline is responsible for the configuration of the image sensor, lens and flash. For every request that is submitted for capture, the control and capture pipeline solves for the configuration of the camera based on the control settings and the state of the auto control loops. There are three auto control loops available: auto exposure, auto focus and auto white-balance, each with its own state machine. Auto control state transition notification are provided by events on the **EVENT_AUTO_CONTROL_STATE_CHANGED** binding point.

4.3.1. Auto exposure

The auto exposure control loop provides values to set the frame duration, exposure time and gain. If a variable aperture lens is also available, the aperture value is also set by the auto exposure control loop. Furthermore, auto exposure can select the use of flash and do metering for flash captures. In order to achieve optimal settings for a high-quailty still image, in particular when flash is requested, a precapture sequence is needed.

Auto exposure modes

The following modes of operation are available for auto exposure:

- **AE_MODE_OFF**: The auto exposure control loop is disabled. Exposure, gain, frame duration and aperture values are set by the client through IControlSettings::SetExposureTimeRange, IControlSettings::SetGainRange, IControlSettings::SetFrameDurationRange, IControlSettings::SetApertureRange.
- **AE_MODE_ON**: The auto exposure control loop is enabled. Any request that sets the flash mode to **FLASH_MODE_SINGLE** will fire the flash.
- **AE_MODE_ON_AUTO_FLASH**: The auto exposure control loop is enabled. A request that sets the flash mode to **FLASH_MODE_SINGLE** will only cause the flash to fire under low light conditions.

When flash is enabled or requested, the client should trigger a precapture sequence to meter the scene for the proper flash intensity.

Auto exposure states

The auto exposure control loop can be in one of these states:

- **AE_STATE_INACTIVE**: This is the initial state of the auto exposure control loop. This state is set when the camera device is opened, or when auto exposure has been just enabled. It is also the state that is to be reported when auto exposure is off. This is a transient state when auto exposure is enabled and might not be reported to the client.
- **AE_STATE_CONVERGED**: auto exposure has converged to an optimal set of values.
- **AE_STATE_FLASH_REQUIRED**: auto exposure has converged to a good set of values, but the scene brightness is not optimal for a high-quality capture and flash is required.
- **AE_STATE_SEARCHING**: The auto exposure control loop is seeking for a good set of values. When the search is complete the new state will be one of **AE_STATE_CONVERGED** or **AE_STATE_FLASH_REQUIRED**. This is a transient state and might not be reported to the client.
- **AE_STATE_LOCKED**: auto exposure is in locked state. Values will remain fixed until the lock is released. See auto exposure locking

Precapture

To achieve the optimal exposure settings for a high-quality still capture, the client should submit a repeating request with the **PRECAPTURE_TRIGGER_ON** value (see IControlSettings::SetPrecaptureTrigger) While requests having the trigger asserted are processed, the auto exposure control loop will search for an optimal set of values. In addition, if flash has been requested, the precapture sequence might pulse the flash to compute a suitable flash intensity for the current scene. Precapture has its own state machine with the following states:

- **PRECAPTURE_INACTIVE**: Precapture is not running.
- **PRECAPTURE_RUNNING**: Precapture is running but has not yet finished.
- **PRECAPTURE_COMPLETE**: Precapture is complete.

The first request with the precapture trigger will set the precapture sequence state to **PRECAPTRE_RUNNING** and put auto exposure in the **AE_STATE_SEARCHING** state. To meter for flash during the precapture sequence, a request should set the flash mode to **FLASH_MODE_PRECAPTURE** or **FLASH_MODE_PRECAPTURE_REDEYE**.

The behavior of auto exposure during precapture depends on the auto exposure mode and flash mode:

Table 1. Precapture behavior according to AE_MODE and FLASH_MODE

AE_MODE	FLASH_MODE	Behavior
ON	OFF	Auto exposure will meter for a capture without flash.
ON	PRECAPTURE	Auto exposure will meter for a capture with flash.
ON	PRECAPTURE_REDEYE	Auto exposure will meter for a capture with flash and pulse additional flashes to achieve red eye reduction.
ON_AUTO_FLASH	OFF	Auto exposure will meter for a capture without flash.
ON_AUTO_FLASH	PRECAPTURE	Auto exposure will do metering and meter for flash only if the scene brightness is low.
ON_AUTO_FLASH	PRECAPTURE_REDEYE	Auto exposure will do metering and meter for flash with red eye reduction only if the scene brightness is low.

Once the precapture sequence is completed, the precapture state will be changed to **PRECAPTURE_COMPLETE**. Precapture will remain in this state as long as the precapture trigger is present in the request. In addition, while precapture is in **PRECAPTURE_COMPLETE** auto exposure will be in **AE_STATE_LOCKED**. This means that after precapture is completed, and while the precapture trigger is still present, auto exposure will remain locked. This is particularly useful for burst imaging or bracketed captures.

To reduce capture latency, a request in the FIFO can be blocked on the precapture state until precapture has finished. If a blocking request is submitted and precapture state is inactive, then the request is not blocked and goes through.

The state of the precapture sequence is available in the frame metadata (and through events?).

Auto exposure locking

In addition to the automatic auto exposure locking that occurs after precapture, the client can lock the auto exposure control loop to keep the current exposure, gain, frame duration and aperture values fixed. Locking is achieved by setting IControlSettings::SetAELock to **true**. When a request that enables locking is processed, auto exposure will transition to the **AE_STATE_LOCKED** state. When locking is disabled, auto exposure will transition to **AE_STATE_SEARCHING**. The value of AELock has no effect if the auto exposure mode is **AE_MODE_OFF** or when the precapture state is not **PRECAPTURE_INACTIVE**.

Auto exposure regions

Settings priorities.

4.3.2. Auto focus

The auto focus control loop moves the lens to achieve a sharp in-focus image.

Auto focus modes

The following modes of operation are available for the auto focus control loop:

- **AF_MODE_OFF**: The auto focus control loop is disabled. The position of the lens is set by the client through IControlSettings::SetFocusDistanceRange
- **AF_MODE_AUTO**: The auto focus control loop is enabled. In this mode the lens doesn’t move until the client triggers an auto focus scan by setting an auto focus precatpure trigger.
- **AF_MODE_CONTINUOUS_SLOW**: The auto focus control loop is enabled. In this mode the auto focus algorithm tries to keep a focused image stream by passively moving the lens. The lens moves in a manner that is suitable for video recording, that is, with slow motions and without overshooting.
- **AF_MODE_CONTINUOUS_FAST**: The auto focus control loop is enabled. When an af active scen is triggered, the lens will move fast to converge to a focused state.

Focuser state

- **AF_FOCUSER_MOVING** : The focuser is moving during frame acquisition.
- **AF_FOCUSER_NOT_MOVING**: The focuser is not moving.

Auto focus states

- **AF_STATE_FOCUSED** : The auto focus control loop has determined the ROI is sharp and in-focus.

- **AF_STATE_NOT_FOCUSED** : The ROI is not focused.

Auto focus triggers

To actively trigger an auto focus sweep on a request, the client should set IControlSettings::SetAFTrigger to **AF_TRIGGER_SCAN**. The implementation is guaranteed to report AF_FOCUSER_MOVING until the focus sweep is complete (AF_FOCUSER_NOT_MOVING). When the sweep is complete, the focuser will stop moving until another scan is triggered.

The application should keep the **AF_TRIGGER_SCAN** during the entire focus sweep. If **AF_TRIGGER_SCAN** is not asserted during the focus sweep, the focuser will stay fixed in its last position. When the focuser has completed the sweep and it is in **AF_FOCUSER_NOT_MOVING** state the focuser stays in that state. To restart a new focus sweep, the client has to send at least one frame with **AF_TRIGGER_SCAN** not asserted.

Auto focus regions

4.3.3. Auto white-balance

The auto white-balance control loop computes a color transform that removes the color cast of the scene illuminant to render white objects in the scene as white. If the client wants to set white-balance manually, it can do so in one of two ways: it can set the color temperature of the illuminant using IControlSettings::SetAWBMode, or it can set the color correction matrix using IControlSettings::SetColorCorrectionMatrix.

Auto white-balance modes

The auto white-balance control loop mode of operation is set by IControlSettings::SetAWBMode. The following modes are available:

- **AWB_MODE_AUTO** : The auto white-balance control loop is active, any values set by IControlSettings::SetColorCorrectionMatrix are ignored.
- **AWB_MODE_OFF** : The auto white-balance control loop is disabled. The white-balance color transform is defined by IControlSettings::SetColorCorrectionMatrix.

The following modes set white-balance to the specified illuminant. These modes disable the auto-whitebalance control loop and ignore the value in IControlSettings::SetColorCorrectionMatrix.

- **AWB_MODE_INCANDESCENT**
- **AWB_MODE_FLUORESCENT**
- **AWB_MODE_WARM_FLUORESCENT**
- **AWB_MODE_DAYLIGHT**
- **AWB_MODE_CLOUDY_DAYLIGHT**
- **AWB_MODE_TWILIGHT**
- **AWB_MODE_SHADE**

Auto white-balance states

- **AWB_STATE_INACTIVE** : This is the state the auto white-balance control loop takes when first enabled, or when the auto white-balance mode is set to **AWB_MODE_OFF**.
- **AWB_STATE_SEARCHING** : The auto white-balance control loop doesn't have a good set of values for the current conditions.
- **AWB_STATE_CONVERGED** : The auto white-balance control loop has a good set of values for the current conditions.
- **AWB_STATE_LOCKED** : The auto-white balance control loop is locked. The values will remain fixed until it is unlocked or disabled.

Auto white-balance regions

4.3.4. Image processing pipeline

TODO: Move all IStreamSettings to IControlSettings

4.4. Statistics

Discussion in Vancouver: 1. Stats (histograms, face detection, sharpness map) will be computed over entire crop rectangle.

All statistics will be computed after the last stage of the image processing pipeline, before scaling/post-processing.

Statistics can be available early through events, plus they will be available with the Frame. The application will have to explicitly enable the statistics using IControlSettings::SetHistogramMode, SetSharpnessMapMode, SetFaceDetectionMode. TODO: Need to also add static properties for these.

TODO: Explore if we need this - define when we come to RAW.

5. Events architecture

The configuration stage of a `CaptureSession` allows for the creation of streams and event queues. All streams and event queues must be created before the `CaptureSession` is committed. An `EventQueue` is a small first-in first-out queue of fixed size that is associated to a given event point within the capture session or stream. When the client binds an event point there are two queues created. One of these queues is managed by the OpenKCam implementation. Events that are generated during the processing of a request are placed in this queue. The second queue is created empty and returned to the client. Whenever the client needs to query for events it will pass this queue in the corresponding `WaitForEvents` call. During the call to `WaitForEvents` the implementation will copy any events in its internal queues to the corresponding client event queues passed in the call. If all of the implementation-side queues are empty, the call will block, until one or more events become available, at which point they will be copied to the client queue. This separation between client and implementation queues enables the creation of deterministic event loops because once the `WaitForEvent` call returns, the client side queues have a fixed number of events for the client to process. If any new event occurs between `WaitForEvent` calls, these events are queued in the implementation-side queue.

5.1. CaptureSession methods

5.1.1. CreateSessionEventQueue

```
IEventQueue *CreateSessionEventQueue( int32_t sessionEventType,
Status *status = 0 );
```

Creates an event queue for the given session event types. The following event types are available:

- **EVENT_CAPTURE_STARTED** : This event is triggered when the sensor has started exposing a new frame.
- **EVENT_AE_STATE_CHANGED** : This event is triggered when the auto exposure control loop has transitioned state.
- **EVENT_AF_STATE_CHANGED** : This event is triggered when the auto focus control loop has transitioned state.
- **EVENT_AWB_STATE_CHANGED** : This event is triggered when the auto-whitebalance control loop has transitioned state.
- **EVENT_ERROR** : An error has occurred during the capture.

5.1.2. CreateStreamEventQueue

```
IEventQueue *CreateStreamEventQueue(
    IStream *stream,
    int32_t streamEventType,
    Status *status = 0);
```

Creates an event queue for the given stream type event. The following stream event types are available:

- **EVENT_FRAME_AVAILABLE** : A new frame is available on this stream.
- **EVENT_STATISTICS_AVAILABLE** : A new set of statistics are available for this stream.

5.1.3. WaitForEvents

```
// TODO: What do we do - do we return a vector.
// TODO: Replace std::vector with something else.
// Specify up to queue size events will be copied. If the internal
// queue size is larger, the application will have to call WaitForEvents
// again.
void WaitForEvents(
    std::vector<IEventQueue*> &queues, // in-out,
    int32_t timeout,
    Status *status = 0);
```

Waits for the next available event on any of the queues, if multiple events happen simultaneously they will all be returned; if multiple events have happened since the last call, they will all be returned immediately.

When an event occurs, the events are moved from the internal queue and moved to the external one.

Returns the first queue with an available event. If one of the queues is not empty this functions returns the `IEventQueue` interface for this non-empty queue. If all queues are empty, the function blocks for a maximum time as set in the timeout value. On blocking, when the timeout expires the function returns NULL and sets status to indicate the timeout.

5.1.4. IEventQueue

This interface provides a mechanism for accessing events in the event queue.

GetNextEvent

```
Event *GetNextEvent();
```

Returns a pointer to the next event in the queue, or NULL if empty.

6. Structures

6.1. SensorMode

```
struct SensorMode
{
    int32_t  aspectRatio;
    uint32_t pixelArrayWidth;
    uint32_t pixelArrayHeight;
    uint32_t minExposureTime;
    uint32_t maxExposureTime;
    uint32_t minFrameDuration;
    uint32_t maxFrameDuration;
};
```

The `SensorMode` structure describes the properties of an available sensor configuration. The pixel array size represents the maximum width and height in pixels that the sensor will readout in this mode. The pixel array size also determines the coordinate system for various regions that can be specified. The origin (0,0) of this coordinate system is the top left pixel of the pixel array, with (pixelArrayWidth-1, pixelArrayHeight-1) being the bottom right pixel.

Members

- **aspectRatio**: A constant describing the aspect ratio of the sensor under this configuration. The standard defines **SENSOR_ASPECT_RATIO_4_3** and **SENSOR_ASPECT_RATIO_16_9**.
- **pixelArrayWidth**: The width of the active pixel array.
- **pixelArrayHeight**: The height of the active pixel array.
- **minExposureTime**: The shortest exposure time that this sensor mode can accept.
- **maxExposureTime**: The longest exposure time that this sensor mode can accept.
- **minFrameDuration**: The shortest frame duration that this sensor mode can accept.
- **maxFrameDuration**: The longest frame duration that this sensor mode can accept.

7. Standard objects

7.1. CameraProvider

The `CameraProvider` object provides access to the cameras in the system. It is the first object every application running OpenKCam needs to create. As such, it has the special method `CameraProvider::Create()` that is OpenKCam's entry point.

All processes can call CameraProvider::Create once, and get a separate instance of the object. However, a second call from the same process to CameraProvider::Create will fail.

Supported interfaces:

- `ICameraProvider`: provides functions to query the capabilities of the cameras that are present, and open one or more of them.

7.1.1. CameraProvider::Create

```
CameraProvider * Create();
```

This is the entry point to the OpenKCam API runtime.

Returns:
A pointer to a new instance of the CameraProvider.

TODO: Do we need to provide a version number to create?

7.2. CameraDevice

The `CameraDevice` represents a camera. A `CameraDevice` instance can map one-to-one to a camera in the system. **TODO**: can or must? In addition, a `CameraDevice` object can be a **virtual** camera that maps to multiple cameras that produce a single image.

A CameraDevice object takes exclusive access to the underlying hardware resources, therefore if a CameraDevice is already opened by some other client, the call to open the CameraDevice will fail.

Supported interfaces:

- `ICameraDevice`: provides functions to create capture sessions and handle events.
- `ICameraProperties`: provides functions to query the capabilities of the `CameraDevice` object.

7.3. CaptureSession

A `CaptureSession` defines a set of streams that process the output of the imaging sensor. A stream is a pipeline of processing stages that produce an image output. Once all streams are setup, the `CaptureSession` can be configured and begin accepting requests.

Supported interfaces:

- `ICaptureSession`: provides functions to create streams, configure the capture session and submit requests.

7.4. Request

A `Request` object defines the camera control parameters plus the set of streams and its associated settings to be used on a capture to be submitted to the API runtime. Changes to the state request do not affect any previously submitted requests. Every time a request is submitted to capture, the implementation copies the request settings and the client is free to make subsequent changes to the `Request` object to configure a new capture.

Supported interfaces:

- `IRequest`: provides functions to configure the request control parameters and per-stream processing parameters.

7.5. Frame

A `Frame` object is the resulting output of a stream. It embeds an image, the settings that were used for capture and processing, and statistics that were computed by the ISP.

Supported interfaces:

- `IFrame:` provides access to frame data such as capture time, frame sequence number, statistics, and the image object.
- `IControlSettings:` to query the camera control settings that were in effect when the capture was taken.
- `IStreamProcessingSettings:` to query the stream settings that were used in the processing of the image.

7.6. Event

An **Event** represent data that is produced asynchronously by the KCam implementation. **TODO**: What kinds of events are there?

8. Standard interfaces

8.1. ICameraDevice

8.1.1. CreateCaptureSession

```
CaptureSession * CreateCaptureSession( int32_t sensorModeIdx ) = 0;
```

Parameters:

- **sensorModeIdx**: An index to the sensor mode for this capture session. The list of sensor modes is given by ICameraProperties::GetSensorModes. The sensor mode will define the maximum image size for the streams that the `CaptureSession` will be able to create.

Returns:
A pointer to the newly created `CaptureSession` object, or `NULL` in case of a failure.

8.2. ICameraProvider

8.2.1. GetAvailableCameras

```
const std::vector<ICameraProperties *> & GetAvailableCameras() const;
```

Queries the cameras that are available in the system. Each camera in the system has a unique identifier given by a `UUID`. This `UUID` is persistent, that is, the same camera will always be identified by the same `UUID`.

Returns:
A vector of pointers to `ICameraProperties` interfaces.

8.2.2. OpenDevice

```
CameraDevice * OpenDevice( const UUID & uuid,
                           Status *     status = 0 );
```

Opens the camera matching the UUID and returns a pointer to a `CameraDevice` object.

Parameters:

- **uuid:** [input] The unique identifier of the camera to be opened.
- **status:** [output] Optional parameter provided by the caller to return an error code for this call.

Returns:
On success, returns a pointer to a newly created `CameraDevice` object. If the operation fails, returns `NULL`. In case of failure, the `status` parameter can be set to: **TODO**: What?

NOTE Multiple camera devices corresponding to different physical cameras can be active simultaneously.

8.3. ICameraProperties

The `ICameraProperties` interface describes properties of a camera in the system.

8.3.1. GetUUID

```
const & UUID GetUUID() const;
```

Returns:
A unique-identifier (UUID) for the camera. This UUID is to be used when opening a CameraDevice object via CameraProvider::open().

8.3.2. GetSupportedPixelFormats

```
const std::vector<int32_t> & GetSupportedPixelFormats() const;
```

Returns:
A list of supported pixel formats. **TODO**: where are the formats defined?

8.3.3. GetPreferredImageSizes

```
const std::vector<Size> & GetPreferredImageSizes(
                              uint32_t sensorModeIdx,
                              int32_t format ) const;
```

Parameters:

- **format:** [input] One of the supported pixel formats.

Returns:
A list of the preferred image sizes for the given pixel format. All sizes returned are supported by the runtime. If the format is not one of the supportedPixelFormats, returns an empty set of sizes.

Notes: While the stream can support any image size, some image sizes can be supported more efficiently than others. The preferred list of image sizes is one that can be efficiently supported. Streams sized at other resolutions might be inefficient.

8.3.4. GetGainRange

```
Range<float> GetGainRange() const;
```

Returns:
The range that gain can take.

8.3.5. GetSensorModes

```
const std::vector<SensorMode> & GetSensorModes() const;
```

Returns:
A list of available sensor modes with their timing constraints.

8.3.6. GetLensFocusDistanceRange

```
Range<float> GetLensFocusDistanceRange() const;
```

Returns:
The focus distance range of the camera module in diopters (1/m) Infinity is 0. Diopters translate linearly to the distance the lens moves.

8.3.7. GetLensApertureRange

```
Range<float> GetLensApertureRange() const;
```

Returns:
The lens aperture range. For a fixed aperture lens, min and max take the

Units: f-number.

8.3.8. GetLensOpticalStabilizationModes

```
std::vector<int32_t> GetLensOpticalStabilizationModes();
```

Returns the list of available optical stabilization modes. Possible modes are: - **LENS_OPTICAL_STABILIZATION_ON** - **LENS_OPTICAL_STABILIZATION_OFF**

If the lens doesn't support optical stabilization, the list will contain a single entry with **LENS_OPTICAL_STABILIZATION_OFF**. A lens with OIS that is always on will return a list with a single entry **LENS_OPTICAL_STABILIZATION_ON**. If the lens supports OIS and the user client can control its behavior, then the supported modes should include both values.

8.3.9. GetFlashAvailable

```
bool GetFlashAvailable() const;
```

Returns:
True if a flash is available for this camera, False otherwise.

8.4. IControlSettings

The control settings define how the capture parameters for the sensor, lens, and flash are to be determined.

8.4.1. AEAntibandingMode

```
void SetAEAntiBandingMode( int32_t mode );
int32_t GetAEAntiBandingMode() const;
```

Modes: OFF, AUTO, 50HZ, 60HZ.

8.4.2. AELock

```
void SetAELock( bool lock );
bool GetAELock() const;
```

The auto exposure control loop locks the current values for exposure time, gain, and frame duration. This setting is ignored if AEMode is set to **AE_MODE_OFF**.

8.4.3. AEMode

```
void SetAEMode( int32_t mode );
int32_t GetAEMode() const;
```

Setter and getter that defines the mode of operation for the auto exposure control loop. The following modes are available: **AE_MODE_OFF**, **AE_MODE_ON**, **AE_MODE_ON_AUTO_FLASH**, as described in detail in the auto exposure section.

8.4.4. AEPriorityMode

```
void SetAEPriorityMode( int32_t mode );
int32_t GetAEPriorityMode() const;
```

The auto exposure priority mode defines behavior which of shutter or aperture (TODO: what about framerate?) should the auto exposure control loop prioritize. The following values are possible:

- **AE_PRIORITY_MODE_AUTO**: the implementation decides which to give preference to.
- **AE_PRIORITY_MODE_EXPOSURE_TIME**: AE will attempt to fix the exposure time at the target level specified by IControlSettings::SetExposureTimeRange
- **AE_PRIORITY_MODE_APERTURE**: AE will attempt to fix the aperture, when available, at the target level set by IControlSettings::SetApertureRange
- **AE_PRIORITY_MODE_GAIN**: AE will try to fix the gain at the target level set by IControlSettings::SetGainRange.

TODO: Need to explain this one in terms of exposure time, frame duration and aperture.

8.4.5. AERegions

```
void SetAERegions( const std::vector<Rectangle> & regions );
const std::vector<Rectangle> & GetAERegions();
```

Sets a list of areas to use for the auto exposure metering. The maximum number of regions available by the camera device is given by ICameraProperties::GetMaxAERegions. If the maximum reported is 0, the entire pixel array area is used for metering. The coordinates of the rectangles are in pixels and are relative to the current sensor mode pixel array size, with (0,0) being the top-left, and (*SensorMode.pixelArrayWidth* - 1, *SensorMode.pixelArray* -1) the bottom right.

Units: Pixel coordinates.

8.4.6. AETrigger

```
void SetAETrigger( int32_t trigger );
int32_t GetAETrigger() const;
```

Indicates whether the auto exposure algorithm should start a precapture sequence when this request is processed.

Parameters:

- *trigger*: One of **AE_TRIGGER_PRECAPTURE** or **AE_TRIGGER_NONE**.

By default, this property is set to **AE_TRIGGER_NONE**. To trigger the precapture sequence, this property should be set to **AE_TRIGGER_PRECAPTURE**, and keep asserted until the precapture sequence is completed. TODO: See ae precapture. Add event.

8.4.7. AFMode

```
void SetAFMode( int32_t mode );
int32_t GetAFMode() const;
```

One of the following values **AF_MODE_OFF**, **AF_MODE_AUTO**, **AF_MODE_CONTINUOUS_SLOW**, **AF_MODE_CONTINUOUS_FAST**.

See autofocus details.

8.4.8. ApertureRange

```
void SetApertureRange( RangeWithTarget<float> range );
RangeWithTarget<float> GetApertureRange() const;
```

8.4.9. AFRegions

```
void SetAFRegions( const std::vector<Rectangle> & regions );
const std::vector<Rectangle> & GetAFRegions();
```

Similar to AE regions.

8.4.10. AFTrigger

```
void SetAFTrigger( int32_t trigger ) const;
int32_t GetAFTrigger() const;
```

Need to keep asserted during a scan sweep. See autofocus.

8.4.11. AWBLock

```
void SetAWBLock( bool lock );
bool GetAWBLock();
```

8.4.12. AWBMode

```
void SetAWBMode( int32_t mode );
int32_t GetAWBMode() const;
```

One of the following values:

- **AWB_MODE_OFF**:
- **AWB_MODE_AUTO**:
- **AWB_MODE_INCANDESCENT**:
- **AWB_MODE_FLUORESCENT**:
- **AWB_MODE_WARM_FLUORESCENT**:
- **AWB_MODE_DAYLIGHT**:
- **AWB_MODE_CLOUDY_DAYLIGHT**:
- **AWB_MODE_TWILIGHT**:
- **AWB_MODE_SHADE**:

8.4.13. AWBRegions

```
void SetAWBRegions( const std::vector<Rectangle> & regions ) = 0;
const std::vector<Rectangle> & GetAWBRegions() = 0;
```

Same as AERegions.

8.4.14. ColorTemperatureRange

```
void SetColorTemperatureRange( Range<int32_t> range );
Range<int32_t> GetColorTemperatureRange() const;
```

When auto-whitebalance is enabled, the range provides a lower and upper bound limit on the color temperature that the auto-whitebalance control loop can take. When auto-whitebalance is disabled, the application should set the lower bound to the desired color temperature.

Units: Kelvin.

8.4.15. ExposureCompensation

```
void SetExposureCompensation( float ev );
float GetExposureCompensation() const;
```

Sets the exposure adjustment step in stops.

When the auto exposure control loop is enabled, the exposure compensation is applied by auto exposure when computing the new exposure time, gain and frame duration for the current request.

What happens is exposure compensation computes a value outside the min/max ranges? Clamp at the min/max ranges for exposure time/gain.

8.4.16. ExposureTimeRange

```
virtual void SetExposureTimeRange( RangeWithTarget<uint32_t> exposureTimeRange );
virtual RangeWithTarget<uint32_t> GetExposureTimeRange() const;
```

When auto exposure is enabled, this range sets a lower and upper bound limit for the values the exposure timecan take. When auto exposure is disabled, the application should set the lower bound to the desired exposure time.

Units: microseconds

8.4.17. FlashMode

```
void SetFlashMode( int32_t mode );
int32_t GetFlashMode() const;
```

When the auto exposure control loop is disabled, the application can control the flash behavior by setting one of the following values:

- **FLASH_MODE_OFF**: The flash will not fire.
- **FLASH_MODE_SINGLE**: The flash will fire once for this request.
- **FLASH_MODE_PRECAPTURE**: Set this mode for a precapture with flash
- **FLASH_MODE_PRECAPTURE_REDEYE**: Set this mode for a precapture sequence with flash + red eye reduction.
- **FLASH_MODE_TORCH**: The torch will be set to on. **We probably need torch on torch off**.

8.4.18. FocusDistanceRange

```
void SetFocusDistanceRange( RangeWithTarget<float> range );
RangeWithTarget<float> GetFocusDistanceRange() const;
```

When auto focus is enabled, this value is a lower bound to the focus distance. The auto focus control loop shall not set a focus distance shorter than this value. When auto focus is disabled, the focuser will be fixed at the target.

Units: Diopters

8.4.19. FrameDurationRange

```
void SetFrameDurationRange( RangeWithTarget<uint32_t> range );
RangeWithTarget<uint32_t> GetFrameDurationRange() const;
```

When auto exposure is enabled, this range sets a lower and upper bound limit for the values the frame duration can take. When auto exposure is disabled the frame rate is set to the target. Frame duration will cap exposure time.

Units: microseconds

8.4.20. GainRange

```
void SetGainRange( RangeWithTarget<float> gainRange );
RangeWithTarget<float> GetGainRange() const;
```

When auto exposure is enabled, this range sets a lower and upper bound limit for the values the gain duration can take. When auto exposure is disabled, the application should set target to the desired gain.

8.4.21. LensOpticalStabilizationMode

```
void SetLensOpticalStabilizationMode( int32_t );
int32_t GetLensOpticalStabilizationMode();
```

8.4.22. ColorCorrectionMatrix

```
void SetColorCorrectionMatrix( float *matrix );
float *GetColorCorrectionMatrix() const;
```

The color correction matrix is a 3x3 matrix that maps sensor RGB to linear sRGB.

8.4.23. DeadPixelMode

```
void SetDeadPixelMode( int32_t mode );
int32_t GetDeadPixelMode();
```

8.4.24. DenoiseMode

```cpp
void SetDenoiseMode( int32_t mode );
int32_t GetDenoiseMode() const;
```

8.4.25. EdgeSharpeningMode

```cpp
void SetEdgeSharpeningMode( int32_t mode ) = 0;
int32_t GetEdgeSharpeningMode() const = 0;
```

8.4.26. ToneMapCurve

```cpp
void SetToneMapCurve( const std::vector< std::pair<float, float> > & curve );
```

Sets the tone map curve for the luminance channel. The curve is given as a series of points that map the range [0,1] → [0,1].

TODO: Come back to it. Luminance only? Last thing that is done?

8.4.27. ToneMapMode

```cpp
void SetToneMapMode( int32_t mode );
int32_t GetToneMapMode() const;
```

8.4.28. LensShadingMode

```
[source,cpp]
----
void SetLensShadingMode( int32_t mode );
int32_t GetLensShadingMode() const;
----
```

8.5. ICaptureSession

TODO: Current discussion on buffer management. Buffers are pre-allocated and passed into the stream. Ownership of the buffers moves between the API and the app. When the API returns the buffer through an Image, the app is responsible of destroying the image object and return the buffer to the API. If the app takes ownership of the buffer for a long period of time, this could cause the API to starve for buffers. In this case requests in the FIFO will be stalled and not issued until the next frame interval and a buffer is available. Similarly for repeating requests. When a request is starved, an event **EVENT_OUT_OF_BUFFERS** will be triggered to notify the client of such condition.

8.5.1. CreateStream

TODO: Check uint32_t vs int32_t! TODO: status = NULL TODO: Look at error handling in GL spec.

```
IStream * CreateStream( uint32_t  pixelFormat,
                        uint32_t  width,
                        uint32_t  height,
                        uint32_t  numImageBuffers,
                        Image ** imageBuffers ,
                        Status * status = 0 );
```

Creates a new stream to output images of a given format and size.

Parameters:

- **pixelFormat:** [input] The output format for this stream.
- **width:** [input] The width of the output image size.
- **height:** [input] the height of the output image size.
- **numImageBuffers:** [input] The number of image buffers to use with this stream.
- **imageBuffers:** [input] A pointer to an array of image buffers. If `NULL`, the stream will allocate `numImageBuffers`.
- **status:** [output] An optional pointer to return the status of this call.

Returns:
A pointer to an `IStream` interface, or `NULL` on failure. Possible status codes are: INVALID_PIXEL_FORMAT, INVALID_WIDTH_OR_HEIGHT, OUT_OF_MEMORY. OUT_OF_RESOURCES.

Notes: To know how many streams you can create, keep creating until it fails with OUT_OF_RESOURCES, or OUT_OF_MEMORY.

It is not ok to destroy a buffer that you have passed in before CaptureSession::Destroy returns.

Buffers passed into the stream during the create call are owned by the API

8.5.2. CreateRequest

```
Request * CreateRequest( Status status = 0 ) = 0;
```

Returns:
A pointer to the newly created `Request` object, or `NULL` in case of failure. The `Request` object created is only valid for the `CaptureSession` it was created on.

On failure, returns NULL and a status code. status could be OUT_OF_MEMORY.

8.5.3. Capture

```
uint32_t Capture( const Request * request,
                  Status *        status = 0 )
```

Submits a single capture request to the request queue. The runtime will queue a copy of the request. The client can submit the same request instance in a future call. This is a blocking call, the function will block until it has succesfully inserted the request in the request queue.

Parameters:

- **request:** [input] The request object with the capture configuration.
- **status:** [output] An optional pointer to return the status of this call.

Returns:
On success, a positive integer that identifies the capture; a negative number on failure.

8.5.4. CaptureBurst

```
uint32_t CaptureBurst( const std::vector<Request*> & requestList,
                       Status *                      status = 0 );
```

Submits a burst to the request queue. The implementation will queue a copy of the burst. In addition, the implementation should either accept the entire burst or refuse it completly (that is, no partial bursts will be accepted). Blocking call.

Parameters:

- **requestList:** [input] a list of requests that make up the burst.
- **status:** [output] An optional pointer to return the status of this call.

Returns:
On success, a positive integer that identifies the burst; a negative number on failure.

8.5.5. Repeat

```
int Repeat( const Request * request,
            Status *        status = 0 ) = 0;
```

Sets up a repeating request. This is a convenience method that will queue a request whenever the request queue is empty and the camera is ready to accept new requests.

Parameters:

- **request:** [input] The request object with the capture configuration.
- **status:** [output] An optional pointer to return the status of this call.

Returns:
On success, an a positive integer that identifies the capture; a negative number on failure.

8.5.6. RepeatBurst

```
int RepeatBurst( const std::vector<Request*> & requestList,
                 Status *                      status = 0 );
```

Setup a repeating burst.

Parameters:

- **requestList:** [input] a list of requests that make up the burst.
- **status:** [output] An optional pointer to return the status of this call.

Returns:
On success, a positive integer that identifies the burst; a negative number on failure.

8.5.7. CancelRequests

```
void CancelRequests( Status * status = 0 );
```

Removes one or all previously submitted requests from the queue. When all requests are cancelled, both the FIFO and the streaming requests will be removed. **TODO**: how to separate between one vs. all?

Parameters:

- **status:** [output] An optional pointer to return the status of this call.

8.5.8. IsRepeating

```
bool IsRepeating() const;
```

Returns true if there is a streaming request or burst in place.

8.5.9. StopRepeating

```
void StopRepeating();
```

Clears the repeating request slot.

8.5.10. Destroy

Tears down the capture session. This method is called when the camera needs to be teared down.

All requests in the pipeline will be aborted, any WaitForEvents calls will return with CAPTURE_SESSION_DESTROYED, streams are destroyed, any IStream* previously returned is invalidated.

8.6. IFrame

The `IFrame` interface provides access to frame data such as capture time, frame sequence number, statistics, and the image object.

8.6.1. GetCaptureTime

```
uint64_t GetCaptureTime() const;
```

Returns:
The time the first row of the image started to expose.

8.6.2. GetStatistics

NOTE TODO: Should statistics be an interface?

```
const Statistics & GetStatistics() const = 0;
```

8.6.3. GetImage

```
Image * GetImage() const = 0;
```

NOTE TODO: Needs to be revisited - is the Image an object? Then, what is its lifetime, is it decoupled from that of a frame?

Returns:
A pointer to the `Image` object associated with the frame. The Image is owned by the frame and is valid as long as the frame is valid.

8.7. IRequest

8.7.1. EnableOutputStream

```
void EnableOutputStream( IStream * stream )
```

Enables a stream to be executed during the request. By default, on an empty request, all streams are disabled.

8.7.2. DisableOutputStream

Disables a previously enabled stream.

```
void DisableOutputStream( IStream * stream)
```

8.7.3. GetOutputStreams

```
const std::vector<IStream *> & GetOutputStreams() const;
```

Returns:
A list of all the streams that have been enabled for this request.

8.7.4. GetStreamSettingsInterface

```
Interface * GetStreamSettingsInterface( UUID     interfaceId,
                                        Stream * stream );
```

The post-processing settings are configurable per-stream. This method returns an interface to access the post-processing settings of a previously enabled output stream.

8.8. IStreamProcessingSettings

8.8.1. VideoStabilizationMode

```
void SetVideoStabilizationMode( int32_t mode );
int32_t GetVideoStabilizationMode() const;
```

8.9. IStream

8.9.1. GetFrame

```
Frame * GetFrame( int      timeout,
                  Status * status = 0 );
```

Returns the next available frame in the stream. This is a blocking call. The timeout provides a maximum time to wait for.

Parameters:

- **timeout:** [input] Time in microsecondse to block for. **TODO**: should the type be unsigned int?
- **status:** [output] An optional pointer to return the status of this call.

Returns:
If a frame is available before the timeout, a pointer to the newly created Frame object will be returned. Otherwise, if the timeout period lapses, a NULL pointer is returned and status is set to `UNAVAILABLE`.

8.9.2. GetAvailableHistograms

```
const std::vector<int32_t> & GetAvailableHistograms() const;
```

Returns a list of histogram types that are available for this stream.

8.9.3. GetNumFreeImages

```
uint32_t GetNumFreeImages() const;
```

Returns:
The number of available of `Image` buffers. Each stream is given on creation a number of `Image` buffers to work with. Ownership of these buffers changes from the API to the client and back. When a `Frame` object is returned, the `Image` packed in the frame is owned by the client. When the `Frame` is destroyed, the `Image` is returned to the API. This call returns the number of images that are currently available to the API to store new incoming data.

8.9.4. GetNumAvailableFrames

```
uint32_t GetNumAvailableFrames() const;
```

Returns:
The number of `Frame` objects that are currently available in `Stream` output queue for the client to retrieve.

8.9.5. GetOutputType

```
int32_t GetOutputType() const;
```

Returns:
The format description for the `Image` objects created by this stream.

8.10. List of items for future revision

1. Crop rectangle support (and how it affect the ROIs)
2. Multi-state streams (streams with different processing parameters).
3. Zoom lens control. Challenges: how do we move a zoom with a per-frame API.

4. Flash intensity
5. Weights for AE/AWB/AF regions.