



VisionLabs
MACHINES CAN SEE

VisionLabs FaceEngine Handbook

written for LUNA SDK Mobile Aurora version 5.17.0

Contents

Introduction	5
1 Core Concepts	6
1.1 Common Interfaces and Types	6
1.1.1 Reference Counted Interface	6
1.1.2 Automatic reference counting	6
1.1.2.1 Referencing - without acquiring ownership of object lifetime	6
1.1.2.2 Acquiring - own object lifetime	7
1.1.3 Serializable object interface	8
1.1.4 Auxiliary types	8
1.1.4.1 Image type	8
1.2 Beta Mode	9
2 FaceEngine Structure Overview	10
3 Core Facility	11
3.1 Common Interfaces	11
3.1.1 Face Engine Object	11
3.1.2 Settings Provider	11
3.2 Helper interfaces	11
3.2.1 Archive interface	11
3.3 Data Paths	12
3.3.1 Model Data	12
3.3.2 Configuration Data	12
4 Detection facility	13
4.1 Overview	13
4.2 Detection structure	13
4.3 Face Detection	13
4.3.1 Image coordinate system	13
4.3.2 Face detection	14
4.3.3 Redetect method	14
4.3.4 Face Alignment	14
4.3.4.1 Five landmarks	14
5 Image Warping	15
6 Parameter Estimation Facility	17
6.1 Overview	17

6.2	Best shot selection functionality	17
6.2.1	Eyes Estimation	17
6.2.2	BestShotQuality Estimation	19
6.2.3	LivenessOneShotRGB Estimation	23
7	Descriptor processing facility	26
7.1	Overview	26
7.1.1	Person Identification Task	26
7.2	Descriptor	26
7.2.1	Descriptor Versions	27
7.2.2	Descriptor Batch	27
7.2.3	Descriptor Extraction	28
7.2.4	Descriptor Matching	29
8	System Requirements	31
8.1	Aurora installations	31
9	Hardware requirements	32
9.1	Mobile installations	32
9.1.1	CPU requirements	32
9.1.2	Memory requirements	32
9.1.3	Number of threads on mobile devices	33
10	Best practices	34
10.1	Overview	34
10.1.1	Creation and deletion order	34
10.1.2	Multithread scenario	34
10.1.3	Thread pools	35
10.1.4	Estimators. Creation and Inference	35
10.1.5	Forking process	35
11	Device-specific constraints	36
11.1	Image constraints	36
12	Appendix A. Specifications	37
12.1	Runtime performance for mobile environment	37
12.1.1	Aurora	37
12.1.1.1	Aurora environment. Matcher performance	37
12.1.1.2	Aurora environment. Extractor performance	37
12.1.1.3	Aurora environment. Detector performance	38
12.1.1.4	Aurora environment. Estimations performance with batch interface	38

12.1.1.5	Aurora environment. Estimations performance without batch interface .	39
12.2	Descriptor size	39
12.3	Feature matrix	40
13	Appendix B. Glossary	42
13.1	Descriptor	42
13.2	Cooperative Photoshooting and Recognition	42
13.3	Matching	42

Introduction

This short guide describes core concepts of the product, shows main FaceEngine features and suggests usage scenarios.

This document is not a full-featured API reference manual nor a step by step tutorial. For reference pages, please see Doxygen API documentation that is shipped with FaceEngine. For complete examples, please head to our developer portal.

What this book does, however, is this:

- It describes ideas behind resource management and gives a clue why one or another decision was made. With this in mind, you are ready to write efficient code with FaceEngine;
- It breaks down full face analysis and recognition pipeline in parts and shows how one part affects all the others. This information will help you to adapt FaceEngine to your needs, which is somewhat more productive than blindly following tutorials;
- It details things that are important and omits things that are obvious, so you get information that matters most.

This book is split up into several chapters. There are chapters dedicated to each FaceEngine facility; there are chapters with conceptual overviews; there are chapters with generic information. We tried to write the book starting from low-level concepts and moving on to face detection, description and recognition tasks solving one problem at a time. Although sometimes we just had to give references to chapters ahead, we tried to minimize such jumps.

The opening chapter of this book is called “Core concepts”. It will tell you about memory management techniques, object creation and destruction strategies that are widely used across the entire FaceEngine. The following chapters catch up telling how higher level FaceEngine components are created from those building blocks.

1 Core Concepts

1.1 Common Interfaces and Types

1.1.1 Reference Counted Interface

Everything in FaceEngine object system starts from here. The *IRefCounted* interface provides methods for reference counter access, increment, and decrement. All reference counted objects imply a custom memory management model. This way they support automated destruction when reference count drops to zero as well as more sophisticated strategies of partial destruction and weak referencing required for FaceEngine internal needs. The bare minimum of such functions is exposed to a user allowing:

- to notify the object that it is required by a client via *retaining* a reference to it;
- to notify the object that it is no longer required by *releasing* a reference to it;
- to get actual reference counter value.

Reference counted objects expect some special treatment as well. **Be sure never to call *delete* on any pointer to object derived from IRefCounted! Doing so leads to heap corruption.** Simply calling *release* notifies the system when the object should be destroyed and it does this properly for you.

However, it is not recommended to interact with the reference counting mechanism manually as doing so may be error-prone. Instead, you are strongly advised to use smart pointers that are specially designed to handle such objects and provided by FaceEngine. See section “[Automatic reference counting](#)” for details.

1.1.2 Automatic reference counting

For your convenience, a special smart pointer class *Ref* is provided. It is capable of automatic reference counter incrementing upon its creation and automatic decrementing upon its destruction. It also does an assertion of the inner raw pointer being non-null, thus preventing errors.

Two ways of working with *Ref* are possible:

1.1.2.1 Referencing - without acquiring ownership of object lifetime

```
ISomeObject* createSomeObject();
{
/* Here createSomeObject returns an object with initial reference count of 1
   (otherwise, it would be dead). Then Ref adds another one for itself
   making a total reference count of 2!
*/
Ref<ISomeObject> objref = make_ref(createSomeObject());
/* Here we use the object in any way we want expecting it to be properly
   destroyed when control will leave this scope.
```

```

*/

}
/* Here we have left the scope and Ref was automatically destroyed like any
   other object created on the stack. At the same time, it decreased
   reference count of its internal object by 1 making it 1 again.
*/

```

However, the object is not destroyed automatically! For this to happen, it should have precisely 0 references. Moreover, in this example, the raw pointer to the object is lost, so it is impossible to fix it in any way; thus a memory leak is introduced.

1.1.2.2 Acquiring - own object lifetime

So keeping that in mind we introduce a concept of ownership acquiring. By acquiring an object, you mean that its raw pointer is not going to be used and only a valid Ref to it is required. To acquire ownership, use a special `::acquire()` function. The fixed version of the above example would look like this:

```

ISomeObject* createSomeObject();
{
/* Here createSomeObject returns an object with initial reference count of 1
   (otherwise, it would be dead). Then we acquire it leaving a total
   reference count of 1.
*/
Ref<ISomeObject> objref = acquire(createSomeObject());
/* Here we use the object in any way we want.
*/
}

/* Here we have left the scope and Ref was automatically destroyed like any
   other object created on the stack. At the same time, it decreased
   reference count of its internal object by 1 making it 0. The object is
   destroyed properly by the object system.
*/

```

Do not store or use raw pointers to the object when using the `::acquire()` function, as ownership acquiring invalidates them.

Acquiring way of working with Ref is pretty like standard library `shared_ptr` own lifetime of the object after it returned by `std::make_shared()`.

You can statically cast object type during acquiring or referencing. To achieve this, use special versions of the `::make_ref_as()` and `::acquire_as()` functions. It is your responsibility to ensure that such a cast is possible.

Please refer to FaceEngine Reference Manual for more details on available convenience methods and functions.

As a side note, be informed that *typedefs* for Ref's to all reference counted types are declared. All of them match the following naming convention: *InterfaceNamePtr*. So, for example, *Ref<IDetector>* is equivalent to *IDetectorPtr*.

1.1.3 Serializable object interface

This interface represents an object. Object's contents may be serialized to some data stream and then read back. Think of this as loading and saving.

To interact with the aforementioned data stream, the serializable object needs a user-provided adapter. Such adapter is called the *archive*. See a detailed explanation of it in section “[Archive interface](#)” in chapter “Core facility”.

Serializable interfaces: *IDescriptor*, *IDescriptorBatch*.

1.1.4 Auxiliary types

1.1.4.1 Image type

Since FaceEngine is a computer vision library, it is natural for it to implement some image concept. Therefore, an *Image* class exists. It is designed as a reference counted container for raw pixel color data. Reference counting allows a single image to be shared by several objects. However, one should understand, that each *Image* object is holding a reference to some data, so if the data is modified in any way, this affects all other objects holding the same reference. To make a deep copy of an *Image*, one should use the *clone()* method, since assignment operators just make a reference. It is also possible to clip a part of an image into a new image by means of *extract()* method.

Pixel data may be characterized by color channel layout, i.e., a number of color channels and their order. The engine defines a *Format* structure for that. The *Format* determines:

- Number of color channels (e.g., RGB or grayscale);
- Order of color channel (e.g., RGB vs. BGR).

FaceEngine assumes 8 bits (i.e., 1 byte) per color channel and implements 8 BPP grayscale, 24 BPP RGB/BGR and padded 32 BPP formats. Format conversion functions are also provided for convenience; see the *convert()* function family.

The *Image* class supports data range mapping. It is possible to map a subset of bytes in a rectangular area for reading or writing. The mapped pixels are represented by the *SubImage* structure. In contrast to *Image*, *SubImage* is just a data view and is *not* reference counted. You are not supposed to store *SubImages* longer than it is necessary to complete data modification. See the documentation of the *map()* function family for details.

The supports IO routines to read/write OOM, JPEG, PNG and TIFF formats via FreeImage library.

The absence of image IO is dictated by the fact that FaceEngine focuses on being lightweight and with the minimum possible number of external dependencies. It is not designed solely with image processing purpose in mind. I.e., one may treat video frames as *Images* and process them one by one. In this case, an external (possibly proprietary) video codec is required.

1.2 Beta Mode

Some features in LUNA SDK are available just in Beta mode. This is experimental features which may be unstable. If you want use them, you have to activate betaMode param in config (faceengine.conf).

2 FaceEngine Structure Overview

FaceEngine is subdivided into several facilities. Each facility is dedicated to a single function. Below there is a list of all facilities with short descriptions of functionality they provide. Detailed information may be found in corresponding chapters of this handbook.

FaceEngine facility list:

- Core facility. This facility stores shared low-level FaceEngine types and factories. This facility is responsible for normal functioning of all other facilities by providing settings accessors and common interfaces. The core facility also contains the main FaceEngine root object that is used to create instances of all higher level objects;
- Face detection facility. This facility is dedicated to object detection. It contains various object detector implementations and factories;
- Parameter estimation facility. This facility is dedicated to various image parameter estimation, such as blurriness, transformation and so forth. It contains various estimator implementations and factories;
- Descriptor processing facility. This facility is dedicated to descriptor extraction and matching. The descriptor is a set of features, describing an object, invariant to object transformation, size or other parameters. Descriptor matching allows judging with certain probability whether two objects are the same. This facility contains various descriptor extractors and containers as well as factories, required to produce them.

So, each facility is a set of classes dedicated to some common for them problem domain. Facilities are independent of each other, with several exceptions, like that all higher level facilities depend on the core facility. Interfacility dependencies are thoroughly described in corresponding chapters of this handbook. The actual set of facilities may vary depending on particular FaceEngine distributions as facilities may be licensed and shipped separately.

This handbook describes the very complete FaceEngine distribution, assuming all facilities are available. The facilities are listed in order of increasing complexity. Applying functions from these facilities in this order allows creating a complete face detection, analysis, recognition and matching pipeline with a significant degree of flexibility. The following chapters break down such pipeline in details.

3 Core Facility

3.1 Common Interfaces

3.1.1 Face Engine Object

The Face Engine object is a root object of the entire FaceEngine. Everything begins with it, so it is essential to create at least one instance of it. Although it is possible to have multiple instances of the Face Engine, it is impractical to do so (as explained in section “[Automatic reference counting](#)” in chapter “Core concepts”). To create a Face Engine instance call *createFaceEngine* function. Also, you may specify default *dataPath* and *configPath* in *createFaceEngine* parameters.

3.1.2 Settings Provider

Settings provider is a special entity that loads settings from various locations. Since settings might be shared among several objects, it is useful to cache them to minimize disk reads and provide a dictionary-like interface for named value lookup.

This is what the provider does. The provider object stands somewhat aside FaceEngine facility structure and is created by a separate factory function *createSettingsProvider*. This function accepts configuration file path as a parameter (see section “[Configuration data](#)” for details). By default, the engine holds a single provider instance for all facilities. Think of it as a reference counted config file. This provider is passed by the Face Engine object to each factory it creates. The factory, in turn, can read its configuration data from the object and pass it further to its child objects. In typical scenarios, you should not bother with providers as the engine does everything for you. However, when relying on custom factory creation parameters (see the description in section “[Face engine object](#)”), you have to create and supply a provider wherever it is required manually.

3.2 Helper interfaces

3.2.1 Archive interface

Archive interface is used to provide serialization functions with a data source. It contains methods primarily for data reading and writing. Note, that *IArchive* is not derived from *IRefCounted*, thus does not imply any special memory management strategies.

A few points to keep in mind when implementing your archive:

- FaceEngine objects that use *IArchive* for serialization purposes do call only *write()* (during saving) or only *read()* (during loading) but never both during the same process unless otherwise is explicitly stated;
- During saving or loading FaceEngine objects are free to write or read their data in chunks; e.g., there may be several sequential calls to *write()* in the scope of a single serialization request. The same

is true for *read()*. Basically, *read()* and *write()* should behave pretty much like C *fread()* and *fwrite()* standard library functions.

Any *IArchive* implementation should be aware of these notes.

Since these interface methods are pretty obvious and mostly self-explanatory, we advise you to check out FaceEngine Reference Manual for the details.

3.3 Data Paths

3.3.1 Model Data

Various FaceEngine modules may require data files to operate. The files contain various algorithm models and constants used at runtime. All the files are gathered together into a single *data* directory.

One may override the data directory location by means of *setDataDirectory()* method which is available in *IFaceEngine*. Current data location may be retrieved via *getDataDirectory()* method.

3.3.2 Configuration Data

The configuration file is called *faceengine.conf* and stored in */data* directory by default. ConfigurationGuide.pdf with parameter description and default values is located at */doc* package folder.

At runtime, the configuration file data is managed by a special object that implements *ISettingsProvider* interface (see section “[Settings provider](#)”). The provider is instantiated by means of *createSettingsProvider()* function that accepts configuration file location as a parameter or uses aforementioned defaults if not specified.

One may supply a different configuration to any factory object by means of *setSettingsProvider()* method, which is available in each factory object interface, including *IFaceEngine*. Currently, bound settings provider may be retrieved via *getSettingsProvider()* method.

4 Detection facility

4.1 Overview

Object detection facility is responsible for quick and coarse detection tasks, like finding a face in an image.

4.2 Detection structure

The detection structure represents an images-space bounding rectangle of the detected object as well as the detection score.

Detection score is a measure of confidence in the particular object classification result and may be used to pick the most “confident” face of many.

Detection score is the measure of classification confidence and not the source image quality. While the score is related to quality (low-quality data generally results in a lower score), it is not a valid metric to estimate the visual quality of an image.

4.3 Face Detection

Object detection is performed by the *IDetector* object. The function of interest is *detect()*. It requires an image to detect on and an area of interest (to virtually crop the image and look for faces only in the given location).

Also, face detector implements *detectAsync()* which allows you to asynchronously detect faces and their parameters on multiple images.

Note: Method *detectAsync()* is experimental, and it’s interface may be changed in the future.

Note: Method *detectAsync()* is not marked as *noexcept* and may throw an exception.

4.3.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.

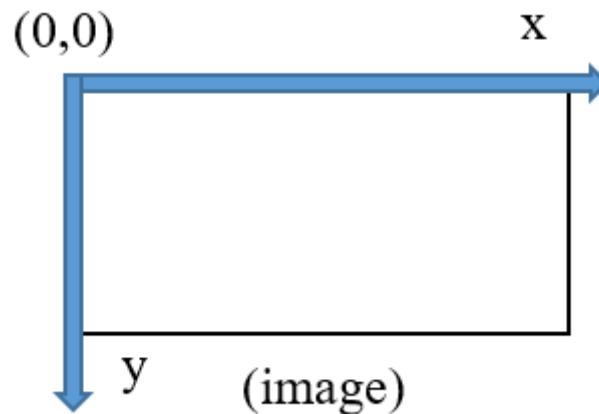


Figure 1: Source image coordinate system

4.3.2 Face detection

When a face is detected, a rectangular area with the face is defined. The area is represented using coordinates in the image coordinate system.

4.3.3 Redetect method

Face detector implements *redetect()* method which is intended for face detection optimization on video frame sequences. Instead of doing full-blown detection on each frame, one may *detect()* new faces at a lower frequency (say, each 5th frame) and just confirm them in between with *redetect()*. This dramatically improves performance at the cost of detection recall. Note that *redetect()* updates face landmarks as well.

Also, face detector implements *redetectAsync()* which allows you to asynchronously redetect faces on multiple images based on the detection results for the previous frames.

Note: Method *redetectAsync()* is experimental, and its interface may be changed in the future.

Note: Method *redetectAsync()* is not marked as *noexcept* and may throw an exception.

Detector works faster with larger value of *minFaceSize*.

4.3.4 Face Alignment

4.3.4.1 Five landmarks

Face alignment is the process of special key points (called “landmarks”) detection on a face. FaceEngine does landmark detection at the same time as the face detection since some of the landmarks are by-products of that detection.

At the very minimum, just **5** landmarks are required: two for eyes, one for a nose tip and two for mouth corners. Using these coordinates, one may warp the source photo image (see Chapter “[Image warping](#)”) for use with all other FaceEngine algorithms.

All detector may provide *5 landmarks* for each detection without additional computations.

Typical use cases for 5 landmarks:

- Image warping for use with other algorithms:
 - Quality and attribute estimators;
 - Descriptor extraction.

5 Image Warping

Warping is the process of face image normalization. It requires landmarks and face detection (see chapter “[Detection facility](#)”) to operate. The purpose of the process is to:

- compensate image plane rotation (roll angle);
- center the image using eye positions;
- properly crop the image.

This way all warped images look the same and one can tell that, e.g., left eye is always in a box, defined by the certain coordinates. This way certain transform invariance is achieved for input data so various algorithms can perform better.

The warper (see `IWarper` in `IWarper.h`):

- Implements the `warp()` function that accepts span of source `fsdk : : Image` in R8B8G8 format, span of `fsdk : : Transformation` and span of output `fsdk : : Image` structures;
- Implements the `warpAsync()` function that accepts span of source `fsdk : : Image` in R8B8G8 format and span of `fsdk : : Transformation`.

Note: Method `warpAsync()` is experimental, and it’s interface may be changed in the future. **Note:** Method `warpAsync()` is not marked as `noexcept` and may throw an exception.

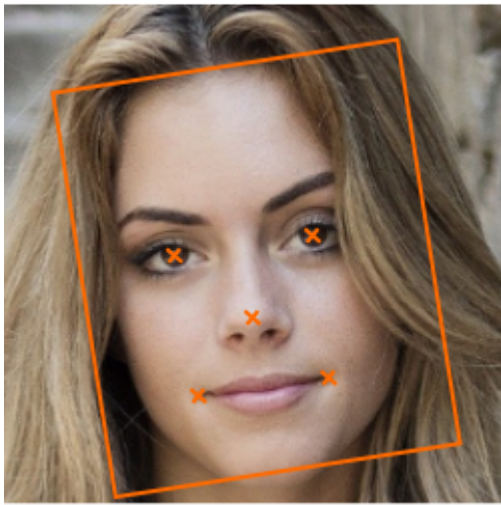


Figure 2: Face warping

Be aware that image warping is not thread-safe, so you have to create a *warper* object per worker thread.

6 Parameter Estimation Facility

6.1 Overview

The estimation facility is the only multi-purpose facility in FaceEngine. It is designed as a collection of tools that help to estimate various images or depicted object properties. These properties may be used to increase the precision of algorithms implemented by other FaceEngine facilities or to accomplish custom user tasks.

6.2 Best shot selection functionality

6.2.1 Eyes Estimation

Name: EyeEstimator

Algorithm description:

The estimator is trained to work with warped images (see chapter “[Image warping](#)” for details).

This estimator aims to determine:

- Eye state: Open, Closed, Occluded;
- Precise eye iris location as an array of landmarks;
- Precise eyelid location as an array of landmarks.

You can only pass warped image with detected face to the estimator interface. Better image quality leads to better results.

Eye state classifier supports three categories: “Open”, “Closed”, “Occluded”. Poor quality images or ones that depict obscured eyes (think eyewear, hair, gestures) fall into the “Occluded” category. It is always a good idea to check eye state before using the segmentation result.

The precise location allows iris and eyelid segmentation. The estimator is capable of outputting iris and eyelid shapes as an array of points together forming an ellipsis. You should only use segmentation results if the state of that eye is “Open”.

Implementation description:

The estimator:

- Implements the *estimate()* function that accepts **warped source image** and warped landmarks, either of type Landmarks5 or Landmarks68. The warped image and landmarks are received from the warper (see `IWarper::warp()`);
- Classifies eyes state and detects its iris and eyelid landmarks;
- Outputs EyesEstimation structures.

Orientation terms “left” and “right” refer to the way you see the *image* as it is shown on the screen. It means that left eye is not necessarily left from the person’s point of view, but is on the left side of the screen. Consequently, right eye is the one on the right side of the screen. More formally, the label “left” refers to subject’s left eye (and similarly for the right eye), such that $x_{right} < x_{left}$.

`EyesEstimation::EyeAttributes` presents eye state as enum `EyeState` with possible values: Open, Closed, Occluded.

Iris landmarks are presented with a template structure `Landmarks` that is specialized for 32 points.

Eyelid landmarks are presented with a template structure `Landmarks` that is specialized for 6 points.

The **EyesEstimation structure** contains results of the estimation:

```
struct EyesEstimation {
    /**
     * @brief Eyes attribute structure.
     * */
    struct EyeAttributes {
        /**
         * @brief Enumeration of possible eye states.
         * */
        enum class State : uint8_t {
            Closed,        //!< Eye is closed.
            Open,          //!< Eye is open.
            Occluded       //!< Eye is blocked by something not transparent
                           , or landmark passed to estimator doesn't point to an eye
                           .
        };

        static constexpr uint64_t irisLandmarksCount = 32; //!< Iris
            landmarks amount.
        static constexpr uint64_t eyelidLandmarksCount = 6; //!< Eyelid
            landmarks amount.

        /// @brief alias for @see Landmarks template structure with
            irisLandmarksCount as param.
        using IrisLandmarks = Landmarks<irisLandmarksCount>;

        /// @brief alias for @see Landmarks template structure with
            eyelidLandmarksCount as param
        using EyelidLandmarks = Landmarks<eyelidLandmarksCount>;

        State state; //!< State of an eye.
```

```

        IrisLandmarks iris; //!< Iris landmarks.
        EyelidLandmarks eyelid; //!< Eyelid landmarks
    };

    EyeAttributes leftEye; //!< Left eye attributes
    EyeAttributes rightEye; //!< Right eye attributes
};

```

API structure name:

IEyeEstimator

Plan files:

- eyes_estimation_flwr8_cpu.plan
- eyes_estimation_ir_cpu.plan
- eye_status_estimation_flwr_cpu.plan
- eyes_estimation_flwr8_cpu-avx2.plan
- eyes_estimation_ir_cpu-avx2.plan
- eyes_estimation_ir_gpu.plan
- eyes_estimation_flwr8_gpu.plan
- eye_status_estimation_flwr_cpu.plan
- eye_status_estimation_flwr_cpu-avx2.plan
- eye_status_estimation_flwr_gpu.plan

6.2.2 BestShotQuality Estimation

Name: BestShotQualityEstimator

Algorithm description:

The BestShotQuality estimator is designed to evaluate image quality to choose the best image before descriptor extraction. The BestShotQuality estimator consists of two components - AGS (garbage score) and Head Pose.

AGS aims to determine the source image score for further descriptor extraction and matching.

Estimation output is a float score which is normalized in range [0..1]. The closer score to 1, the better matching result is received for the image.

When you have several images of a person, it is better to save the image with the highest AGS score.

Recommended threshold for AGS score is equal to **0.2**. But it can be changed depending on the purpose of use. Consult VisionLabs about the recommended threshold value for this parameter.

Head Pose determines person head rotation angles in 3D space, namely pitch, yaw and roll.

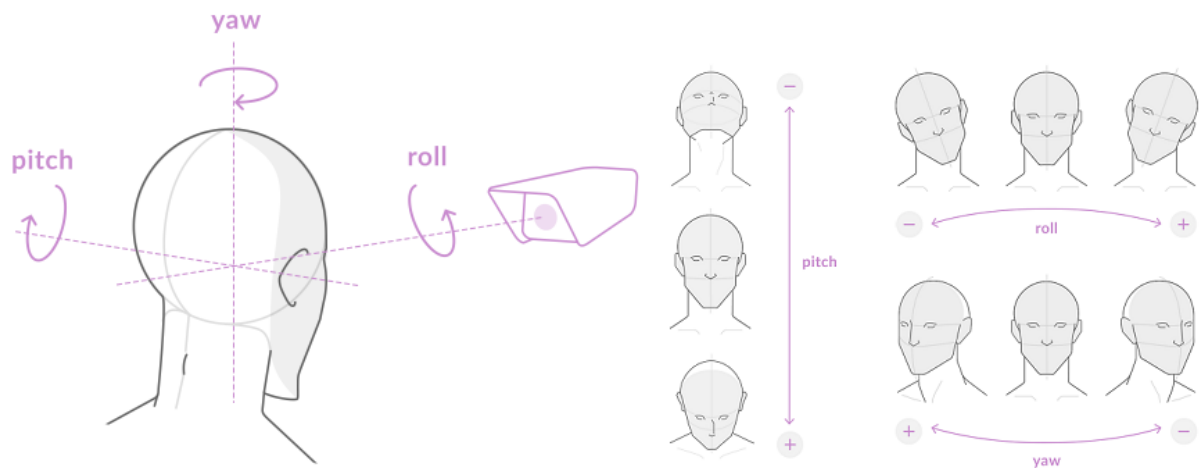


Figure 3: Head pose

Since 3D head translation is hard to determine reliably without camera-specific calibration, only 3D rotation component is estimated.

Head pose estimation characteristics:

- Units (degrees);
- Notation (Euler angles);
- Precision (see table below).

Implementation description:

The estimator (see `IBestShotQualityEstimator` in `IEstimator.h`):

- Implements the `estimate()` function that needs `fsdk::Image` in R8G8B8 format, `fsdk::Detection` structure of corresponding **source image** (see section “[Detection structure](#)” in chapter “Face detection facility”), `fsdk::IBestShotQualityEstimator::EstimationRequest` structure and `fsdk::IBestShotQualityEstimator::EstimationResult` to store estimation result;
- Implements the `estimate()` function that needs the span of `fsdk::Image` in R8G8B8 format, the span of `fsdk::Detection` structures of corresponding **source images** (see section “[Detection structure](#)” in chapter “Face detection facility”), `fsdk::IBestShotQualityEstimator::EstimationRequest` structure and span of `fsdk::IBestShotQualityEstimator::EstimationResult` to store estimation results.
- Implements the `estimateAsync()` function that needs `fsdk::Image` in R8G8B8 format, `fsdk::Detection` structure of corresponding source image (see section “[Detection structure](#)” in chapter “Face detection facility”), `fsdk::IBestShotQualityEstimator::EstimationRequest` structure;

Note: Method *estimateAsync()* is experimental, and it's interface may be changed in the future. **Note:** Method *estimateAsync()* is not marked as *noexcept* and may throw an exception.

Before using this estimator, user is free to decide whether to estimate or not some listed attributes. For this purpose, *estimate()* method takes one of the estimation requests:

- `fsdk::IBestShotQualityEstimator::EstimationRequest::estimateAGS` to make only AGS estimation;
- `fsdk::IBestShotQualityEstimator::EstimationRequest::estimateHeadPose` to make only Head Pose estimation;
- `fsdk::IBestShotQualityEstimator::EstimationRequest::estimateAll` to make both AGS and Head Pose estimations;

The **EstimationResult** structure contains results of the estimation:

```
struct EstimationResult {  
    Optional<HeadPoseEstimation> headPose;    //!< HeadPose estimation if  
        was requested, empty otherwise  
    Optional<float> ags;                      //!< AGS estimation if was  
        requested, empty otherwise  
};
```

Head Pose accuracy:

Prediction precision decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table below.

Table 1: “Head pose prediction precision”

	Range	-45°...+45°	< -45° or > +45°
Average prediction error (per axis)	Yaw	±2.7°	±4.6°
Average prediction error (per axis)	Pitch	±3.0°	±4.8°
Average prediction error (per axis)	Roll	±3.0°	±4.6°

Zero position corresponds to a face placed orthogonally to camera direction, with the axis of symmetry parallel to the vertical camera axis.

API structure name:

`IBestShotQualityEstimator`

Plan files:

- `ags_angle_estimation_flwr_cpu.plan`

- `ags_angle_estimation_flwr_cpu-avx2.plan`
- `ags_angle_estimation_flwr_gpu.plan`

6.2.3 LivenessOneShotRGB Estimation

Name: LivenessOneShotRGBEstimator

Algorithm description:

This estimator shows whether the person's face is real or fake by the following types of attacks:

- Printed Photo Attack. One or several photos of another person are used.
- Video Replay Attack. A video of another person is used.
- Printed Mask Attack. An imposter cuts out a face from a photo and covers his face with it.
- 3D Mask Attack. An imposter puts on a 3D mask depicting the face of another person.

The requirements for the processed image and the face in the image are listed below.

Parameters	Requirements
Minimum resolution for mobile devices	720x960 pixels
Maximum resolution for mobile devices	1080x1920 pixels
Minimum resolution for webcams	1280x720 pixels
Maximum resolution for webcams	1920x1080 pixels
Compression	No
Image warping	No
Image cropping	No
Effects overlay	No
Mask	No
Number of faces in the frame	1
Face detection bounding box size	More than 200 pixels
Frame edges offset	More than 10 pixels
Head pose	-20 to +20 degrees for head pitch, yaw, and roll
Image quality	The face in the frame should not be overexposed, underexposed, or blurred.

See image quality thresholds in the [“Image Quality Estimation”](#) section.

Implementation description:

The estimator (see `ILivenessOneShotRGBEstimator` in `ILivenessOneShotRGBEstimator.h`):

- Implements the `estimate()` function that needs `fsdk::Image`, `fsdk::Detection` and `fsdk::Landmarks5` objects (see section [“Detection structure”](#) in chapter “Face detection facility”). Output estimation is a structure `fsdk::LivenessOneShotRGBEstimation`.
- Implements the `estimate()` function that needs the span of `fsdk::Image`, span of `fsdk::Detection` and span of `fsdk::Landmarks5` (see section [“Detection structure”](#) in chapter “Face detection facility”).

The first output estimation is a span of structure `fsdk::LivenessOneShotRGBEstimation`. The second output value (structure `fsdk::LivenessOneShotRGBEstimation`) is the result of aggregation based on span of estimations announced above. Pay attention the second output value (aggregation) is optional, i.e. `default` argument, which is `nullptr`.

The **LivenessOneShotRGBEstimation structure** contains results of the estimation:

```
struct LivenessOneShotRGBEstimation {
    enum class State {
        Alive = 0,    //!< The person on image is real
        Fake,         //!< The person on image is fake (photo, printed image)
        Unknown       //!< The liveness status of person on image is Unknown
    };

    float score;      //!< Estimation score
    State state;      //!< Liveness status
    float qualityScore; //!< Liveness quality score
};
```

Estimation score is normalized in range [0..1], where 1 - is real person, 0 - is fake.

Liveness quality score is an image quality estimation for the liveness recognition.

This parameter is used for filtering if it is possible to make bestshot when checking for liveness.

The reference score is 0,5.

The value of `State` depends on `score` and `qualityThreshold`. The value `qualityThreshold` can be given as an argument of method `estimate` (see `ILivenessOneShotRGBEstimator`), and in configuration file `faceengine.conf` (see [ConfigurationGuide LivenessOneShotRGBEstimator](#)).

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (`faceengine.conf`) in the `LivenessOneShotRGBEstimator::Settings` section. By default, these threshold values are

set to optimal.

Table 3: “LivenessOneShotRGB estimator recommended thresholds”

Threshold	Recommended value
realThreshold	0.5
qualityThreshold	0.5
calibrationCoeff	0.67

Configurations:

See the “LivenessOneShotRGBEstimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

ILivenessOneShotRGBEstimator

Plan files:

- oslm_v4_model_1_cpu.plan
- oslm_v4_model_2_cpu.plan
- oslm_v4_model_1_arm.plan
- oslm_v4_model_2_arm.plan

7 Descriptor processing facility

7.1 Overview

The section describes descriptors and all the processes and objects corresponding to them.

Descriptors and extraction facility is available only in the Complete edition only!

Descriptor itself is a set of object parameters that are specially encoded. Descriptors are typically more or less invariant to various affine object transformations and slight color variations. This property allows efficient use of such sets to identify, lookup, and compare real-world objects images.

To receive a descriptor you should perform a special operation called descriptor *extraction*.

The general case of descriptors usage is when you compare two descriptors and find their similarity score. Thus you can identify persons by comparing their descriptors with your descriptors database.

All descriptor comparison operations are called *matching*. The result of the two descriptors matching is a distance between components of the corresponding sets that are mentioned above. Thus, from a magnitude of this distance, we can tell if two objects are presumably the same.

7.1.1 Person Identification Task

Facial recognition is the task of making an identification of a face in a photo or video image against a pre-existing database of faces. It begins with detection - distinguishing human faces from other objects in the image - and then works on the identification of those detected faces. To solve this problem, we use a face descriptor, which extracted from an image face of a person. A person's face is invariable throughout his life.

In a case of the face descriptor, the extraction is performed from object image areas around some previously discovered facial landmarks, so the quality of the descriptor highly depends on them and the image it was obtained from.

The process of face recognition consists of 4 main stages:

- face detection in an image;
- warping of face detection – compensation of affine angles and centering of a face;
- descriptor extraction;
- comparing of extracted descriptors (matching).

7.2 Descriptor

Descriptor object stores a compact set of packed properties as well as some helper parameters that were used to extract these properties from the source image. Together these parameters determine descriptor compatibility. Not all descriptors are compatible with each other. It is impossible to batch and match

incompatible descriptors, so you should pay attention to what settings do you use when extracting them. Refer to section [“Descriptor extraction”](#) for more information on descriptor extraction.

7.2.1 Descriptor Versions

Face descriptor algorithm evolves with time, so newer FaceEngine versions contain improved models of the algorithm.

Descriptors of different versions are **incompatible**! This means that you **cannot match descriptors with different versions**. This does not apply to base and mobilenet versions of the same model: they are compatible.

See chapter [“Appendix A. Specifications”](#) for details about performance and precision of different descriptor versions.

Descriptor version 59 is the best one by precision. And it works well with the personal protective equipment on face like medical mask.

Descriptor version may be specified in the configuration file (see section [“Configuration data”](#) in chapter [“Core facility”](#)).

7.2.2 Descriptor Batch

When matching significant amounts of descriptors, it is desired that they reside continuously in memory for performance reasons (think cache-friendly data locality and coherence). This is where descriptor batches come into play. While descriptors are optimized for faster creation and destruction, batches are optimized for long life and better descriptor data representation for the hardware.

A batch is created by the factory like any other object. Aside from type, a size of the batch should be specified. Size is a memory reservation this batch makes for its data. It is impossible to add more data than specified by this reservation.

Next, the batch must be populated with data. You have the following options:

- add an existing descriptor to the batch;
- load batch contents from an archive.

The following notes should be kept in mind:

- When adding an existing descriptor, its data is copied into the batch. This means that the descriptor object may be safely released.
- When adding the first descriptor to an empty batch, initial memory allocation occurs. Before that moment the batch does not allocate. At the same moment, internal descriptor helper parameters are copied into the batch (if there are any). This effectively determines compatibility possibilities of the batch. When the batch is initialized, it does not accept incompatible descriptors.

After initialization, a batch may be matched pretty much the same way as a simple descriptor.

Like any other data storage object, a descriptor batch implements the `::clear()` method. An effect of this method is the batch translation to a non-initialized state **except memory deallocation**. In other words, batch capacity stays the same, and no memory is reallocated. However, an actual number of descriptors in the batch and their parameters are reset. This allows re-populating the batch.

Memory deallocation takes place when a batch is released.

Care should be taken when serializing and deserializing batches. When a batch is created, it is assigned with a fixed-size memory buffer. The size of the buffer is embedded into the batch BLOB when it is saved. So, when allocating a batch object for reading the BLOB into, make sure its size is at least the same as it was for the batch saved to the BLOB (even if it was not full at the moment). Otherwise, loading fails. Naturally, it is okay to deserialize a smaller batch into a larger another batch this way.

7.2.3 Descriptor Extraction

Descriptor extractor is the entity responsible for descriptor extraction. Like any other object, it is created by the factory. To extract a descriptor, aside from the source image, you need:

- a face detection area inside the image (see chapter “[Detection facility](#)”)
- a pre-allocated descriptor (see section “[Descriptor](#)”)
- a pre-computed landmarks (see chapter “[Image warping](#)”)

A descriptor extractor object is responsible for this activity. It is represented by the straightforward *IDescriptorExtractor* interface with only one method *extract()*. Note, that the descriptor object must be created prior to calling *extract()* by calling an appropriate factory method.

Landmarks are used as a set of coordinates of object points of interest, that in turn determine source image areas, the descriptor is extracted from. This allows extracting only data that matters most for a particular type of object. For example, for a human face we would want to know at least definitive properties of eyes, nose, and mouth to be able to compare it to another face. Thus, we should first invoke a feature extractor to locate where eyes, nose, and mouth are and put these coordinates into landmarks. Then the descriptor extractor takes those coordinates and builds a descriptor around them.

Descriptor extraction is one of the most computation-heavy operations. For this reason, threading might be considered. Be aware that descriptor extraction is not thread-safe, so you have to create an extractor object per a worker thread.

It should be noted, that the face detection area and the landmarks are required only for image warping, the preparation stage for descriptor extraction (see chapter “[Image warping](#)”). If the source image is already warped, it is possible to skip these parameters. For that purpose, the *IDescriptorExtractor* interface provides a special *extractFromWarpedImage()* method.

Descriptor extraction implementation supports execution on GPUs.

The *IDescriptorExtractor* interface provides *extractFromWarpedImageBatch()* method which allows you to extract batch of descriptors from the image array in one call. This method achieve higher utilization of GPU and better performance (see the “GPU mode performance” table in appendix A chapter “Specifications”).

Also *IDescriptorExtractor* returns *descriptor score* for each extracted descriptor. Descriptor score is normalized value in range [0,1], where 1 - face in the warp, 0 - no face in the warp. This value allows you filter descriptors extracted from false positive detections.

The *IDescriptorExtractor* interface provides *extractFromWarpedImageBatchAsync()* method which allows you to extract batch of descriptors from the image array asynchronously in one call. This method achieve higher utilization of GPU and better performance (see the “GPU mode performance” table in appendix A chapter “Specifications”).

Note: Method *extractFromWarpedImageBatchAsync()* is experimental, and it’s interface may be changed in the future.

Note: Method *extractFromWarpedImageBatchAsync()* is not marked as noexcept and may throw an exception.

7.2.4 Descriptor Matching

It is possible to match a pair (or more) previously extracted descriptors to find out their similarity. With this information, it is possible to implement face search and other analysis applications.

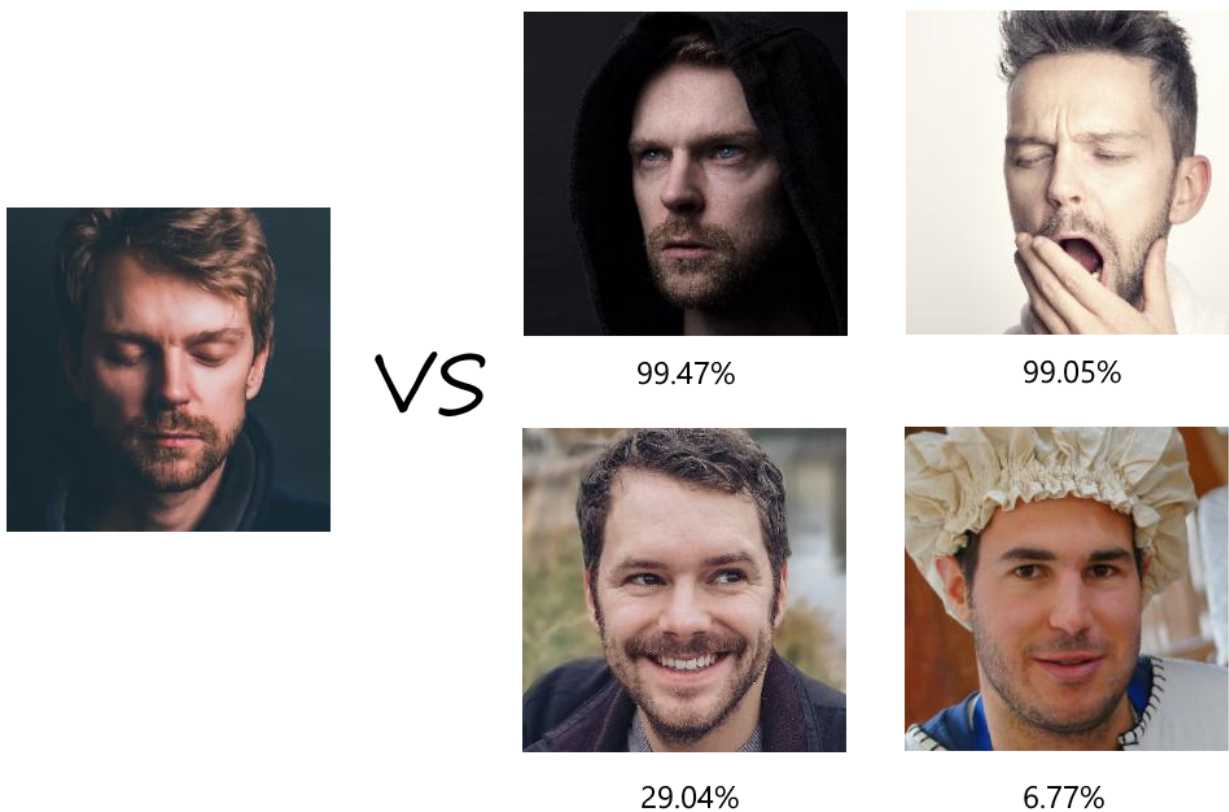


Figure 4: Matching

By means of *match* function defined by the *IDescriptorMatcher* interface it is possible to match a pair of descriptors with each other or a single descriptor with a descriptor batch (see section “[Descriptor batch](#)” for details on batches).

A simple rule to help you decide which storage to opt for:

- when searching among less than a hundred descriptors use separate *IDescriptor* objects;
- when searching among bigger number of descriptors use a batch.

When working with big data, a common practice is to organize descriptors in several batches keeping a batch per worker thread for processing.

Be aware that descriptor matching is not thread-safe, so you have to create a matcher object per a worker thread.

8 System Requirements

8.1 Aurora installations

We support AuroraOS-4.0.1.43-base-armv7hl

Supported compiler:

- GNU 8.3.0

Other versions were not tested.

Note 1: GPU computing is not supported.

9 Hardware requirements

9.1 Mobile installations

Table 4: Models provided in distribution package and supported devices.

Neural network	CPU	ARM
FaceDet_v2_<detector_type>first<device>.plan	yes	yes
FaceDet_v2_<detector_type>second<device>.plan	yes	yes
FaceDet_v2_<detector_type>third<device>.plan	yes	yes
ags_angle_estimation_flwr_<device>.plan	yes	yes
headpose_v3_<device>.plan	yes	yes
ags_v3_<device>.plan	yes	yes
eyes_estimation_flwr8_<device>.plan	yes	yes
eye_status_estimation_flwr_<device>.plan	yes	yes
cnn59m_<device>.plan	yes	yes
oslm_v4_model_<model_id>_<device>.plan	yes	yes

9.1.1 CPU requirements

Supported CPU architectures:

- armv7hl;

9.1.2 Memory requirements

RAM requirements are given for common for mobile platform verification pipeline.

Table 5: “Memory requirements”

Requirements for	Aurora
RAM	400 MB
Storage Full	350 MB
Storage Frontend	300 MB

9.1.3 Number of threads on mobile devices

The description of according settings you can find in “Configuration Guide - Runtime settings”. The setting `<param name="numThreads" type="Value::Int1"x="-1"/>` means that will be taken the maximum number of available threads. This number of threads is equal to according the number of available processor cores. We strongly recommend you to follow this recommendation; otherwise, performance can be significantly reduced.

10 Best practices

10.1 Overview

The following chapter provides a set of recommendations that user should follow in order to get optimal performance when running Luna SDK algorithms on their target device. Over time this list will be populated with more recommendations and performance tips.

10.1.1 Creation and deletion order

All Luna SDK objects should be destroyed in the order reversal to their creation order. This implies the following:

- at first FaceEngine object should be created (using createFaceEngine method)
- after that all child objects, such as detectors, estimators etc, can be created
- at the end of the work all these child objects should be deleted in the first place
- and only after that the main FaceEngine object can be deleted

It is not recommended to use FaceEngine objects as globals (or static objects), because in this case their deletion order could be undefined. In the case when such a usage is necessary, the correct deletion order should be guaranteed via explicit deletion of all objects in the correct order, before the end of the program. For instance:

```
fsdk::IFaceEnginePtr faceEngine = fsdk::createFaceEngine("./data");
fsdk::IDetectorPtr detector = faceEngine->createDetector();
fsdk::IBestShotQualityEstimator bestShotQualityEstimator = faceEngine->
    createBestShotQualityEstimator();

int main() {
    // application code here

    detector.reset();
    bestShotQualityEstimator.reset();
    faceEngine.reset();
    return 0;
}
```

10.1.2 Multithread scenario

Creation and destroying Luna SDK algorithms from the different threads is prohibited due to internal implementation restrictions. All objects of the FaceEngine class and all objects of algorithms (for

example, detectors, estimators, extractors and others) must be created and destroyed by the same thread. A typical scenario is as follows: Thread 1 (may be a main thread) creates the FaceEngine object and all needed algorithms (for example, IDetector). Threads 2..N (maybe several) use those objects for any purpose. Thread 1 destroys the FaceEngine object and all algorithms after all work is complete.

10.1.3 Thread pools

When running Luna SDK algorithms in a multithreaded environment it is highly recommended to use thread pools for user-created threads. For each thread Luna SDK caches some amount of thread local objects under the hood in order to make its algorithms run faster next time the same thread is used at the cost of higher memory footprint. For this reason, it is recommended to reuse threads from a pool in order to avoid caching new internal objects and to reduce penalty of creating/destroying new user threads.

10.1.4 Estimators. Creation and Inference

Create face engine objects once and reuse them when you need to make a new estimate to reduce RAM usage and increase performance. The reason is that recreating of estimators leads to reopen the corresponding plan file every time. These plan files are cached separately for every load and will be removed only when they are flushed from the cache or after calling the destructor of FaceEngine root object.

10.1.5 Forking process

UNIX-like operating systems implement a mechanism to duplicate a process. It creates a new child process and copies its parents' memory space into the child's. This is typically done programmatically by calling the `fork()` system function in the parent process. Care should be taken when forking a process running the SDK. Always fork before the first instance of `IFaceEngine` is created! This is because the SDK internally maintains a pool of worker threads, which is created lazily at the time the very first `IFaceEngine` object is born and destroyed right after the last `IFaceEngine` object is released. When using GPU or NPU devices, their runtime is initialized and shut down in the same manner. The hazard comes from the fact that while `fork()` copies process memory, it only creates just one thread - the main thread (refer to man pages for details: <https://man7.org/linux/man-pages/man2/fork.2.html>). As a result, if at least one `IFaceEngine` object is alive at the time the process is being forked, the child processes will inherit the knowledge of the object, and therefore, the implicit thread pool (and device runtime, when appropriate). But there will be no worker threads actually running (in both, the inherited pool and the runtime, when appropriate) and attempting to call certain SDK functions will cause a deadlock.

11 Device-specific constraints

11.1 Image constraints

When memory is allocated for Image pixel data storage, the following constraints are enforced depending on the requested memory residence:

- Image::MemoryResidence::CPU: base address alignment is 32 bytes;
- Image::MemoryResidence::GPU: base address alignment is 128 bytes;
- Image::MemoryResidence::NPU: base address alignment is 128 bytes;
- Image::MemoryResidence::NPU_DPP: base address alignment is 128 bytes.

Also, in case of Image::MemoryResidence::NPU_DPP image width must be multiple of 16 and image height must be multiple of 2.

When Image is initialized as a wrapper for a user-provided memory block, whose residence is said to be Image::MemoryResidence::NPU or Image::MemoryResidence::NPU_DPP, the above requirements are checked upon the initialization.

Image class implements limited functionality for device-side data. Only the following operations are supported:

- construction (both with Image-owned memory and as a wrapper for a user-defined memory) and assignment (including deep copy);
- destruction;
- set() family of functions (functionally the same as construction/assignment);
- convert() function, but only in transfer mode; This means that both source and destination formats must match, only memory residency may differ. This function supports only synchronous memory transfers in the following directions:
 - host <-> GPU
 - GPU <-> GPU
 - host <-> NPU
 - NPU <-> NPU.

Full range of functionality (including format conversions) is currently only available for Images with host memory data residence.

The following operations are **NOT** supported:

- compressed format encoding/decoding;
- format/color space conversion;
- subimage views (i.e. map() function);
- padding and cropping (i.e. extract() function);
- manipulation (e.g. getPixel(), setPixel(), etc.).

12 Appendix A. Specifications

12.1 Runtime performance for mobile environment

Face detection performance depends on input image parameters such as resolution and bit depth as well as the size of the detected face. The Aurora platform uses mobilenet by default.

Input data characteristics:

- Image resolution: 640x480px;
- Image format: 24 BPP RGB;

12.1.1 Aurora

The number of threads auto means that will be taken the maximum number of available threads. For this mode use the -1 value for the numThreads parameter in the runtime.conf. This number of threads is equal to according number of available processor cores. We strongly recommend you to follow this recommendation; otherwise, performance can be significantly reduced. Description of according settings you can find in “Configuration Guide - Runtime settings”.

The performance measurements are presented for device with configurations as below:

Architecture: armv7l Byte Order: Little Endian CPU(s): 4 On-line CPU(s) list: 0-3 Thread(s) per core: 1 Core(s) per socket: 4 Socket(s): 1 Vendor ID: ARM Model: 5 Model name: Cortex-A7 Stepping: r0p5 CPU max MHz: 1267.2000 CPU min MHz: 200.0000 BogoMIPS: 38.40 Flags: swp half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 evtstrm

The number of threads you can find in tables below.

12.1.1.1 Aurora environment. Matcher performance

The table below shows the performance of Matcher on the Aurora environment.

Type	Model	CPU threads	Batch Size	Average (matches/sec)
Matcher	59	1	1000	16597.5 K

12.1.1.2 Aurora environment. Extractor performance

The table below shows the performance of Extractor on the Aurora environment.

Measurement	Model	Threads	Batch Size	Average (ms)
Extractor	59	1	1	11817.6
Extractor	59	auto	1	1462.37

Measurement	Model	Threads	Batch Size	Average (ms)
Extractor	59	auto	4	1518.88
Extractor	59	auto	8	2132.2

12.1.1.3 Aurora environment. Detector performance

The table below shows the performance of Detector on the Aurora environment.

Measurement	Threads	Average (ms)
Detector FaceDetV2	1	1191.31 / 1034.12 / 5707.42
(Easy/complex/6 faces)	auto	210.81 / 230.13 / 834.58

12.1.1.4 Aurora environment. Estimations performance with batch interface

The table below shows the performance of Estimations on the Aurora environment for estimators that have a batch interface.

Measurement	Threads	BatchSize	Average (ms)
HeadPose	1	1	72.46
HeadPose	auto	1	12.16
HeadPose	auto	8	9.97
Eyes (RGB, useStatusPlan=0)	1	1	411.33
Eyes (RGB, useStatusPlan=0)	auto	1	67.54
Eyes (RGB, useStatusPlan=0)	auto	8	59.57
Eyes (RGB, useStatusPlan=1)	1	1	427.3
Eyes (RGB, useStatusPlan=1)	auto	1	72.67
Eyes (RGB, useStatusPlan=1)	auto	8	57.98
AGS	1	1	83.23
AGS	auto	1	173.76
AGS	auto	8	78.04
BestShotQuality	1	1	73.43
BestShotQuality	auto	1	12.18

Measurement	Threads	BatchSize	Average (ms)
BestShotQuality	auto	8	9.72
MedicalMaskBatch	1	1	2948.68
MedicalMaskBatch	auto	1	406.63
MedicalMaskBatch	auto	8	372.05
LivenessOneShotRGBEstimatorBatch	1	1	40971.3
LivenessOneShotRGBEstimatorBatch	auto	1	25502.6
LivenessOneShotRGBEstimatorBatch	auto	8	24494.2
Glasses	1	1	1121.78
Glasses	auto	1	1822.48
Glasses	auto	8	1277.76

12.1.1.5 Aurora environment. Estimations performance without batch interface

The table below shows the performance of Estimations on the Aurora environment for estimators that do not have a batch interface.

Measurement	Threads	Average (ms)
Warper	1	26.2
Warper	auto	33.15
Quality	1	632.68
Quality	auto	138.09

12.2 Descriptor size

The table below shows size of serialized descriptors to estimate memory requirements.

Table 11: “Descriptor size”

Descriptor version	Data size (bytes)	Metadata size (bytes)	Total size
CNN 54	512	8	520

Metadata includes signature and version information that may be omitted during serialization if the *NoSignature* flag is specified.

When estimating individual descriptor size in memory or serialization storage requirements with default options, consider using values from the “Total size” column.

When estimating memory requirements for descriptor batches, use values from the “Data size” column instead, since a descriptor batch does not duplicate metadata per descriptor and thus is more memory-efficient.

These numbers are for approximate computation only, since they do not include overhead like memory alignment for accelerated SIMD processing and the like.

12.3 Feature matrix

Mobile versions come in two editions: the frontend edition (or FE for short) and the complete edition.

The table below shows FaceEngine features supported by different editions of mobile platform.

Table 12: “Feature matrix”

Facility	Module	Complete	Frontend
Core		Yes	Yes
Face detection & alignment	Face detector	Yes	Yes
Parameter estimation	BestShotQuality estimation	Yes	Yes
	Color estimation	Yes	Yes
	Eye estimation	Yes	Yes
	Head pose estimation	Yes	Yes
	AGS estimation	Yes	Yes
	LivenessOneShotRGB estimation	Yes	Yes
	Medical Mask estimation	Yes	Yes
	Quality estimation	Yes	Yes
	Glasses estimation	Yes	Yes
Face descriptors	Descriptor extraction	Yes	No
	Descriptor matching	Yes	No
	Descriptor batching	Yes	No
	Descriptor search acceleration	Yes	No

See file “doc/FeatureMapMobile.htm” for more details.

13 Appendix B. Glossary

Table 13: Glossary

Term	Description
Host memory	Computer system RAM
Device memory	On-board RAM of GPU or NPU card
Memory transfer	Operation that copies memory from host to device or vice-versa

13.1 Descriptor

A set of features meant to describe a real-world object (e.g., a person's face). Computed by means of computer vision algorithms, such features are typically matched to each other to determine the similarity of represented objects.

13.2 Cooperative Photoshooting and Recognition

A procedure of taking person face photograph characterized by person awareness of the matter and his/her will to assist.

Typical highlights:

- Close to frontal head pose;
- Neutral facial expression;
- No occlusions (i.e., hair, hats, non-transparent eyewear, hands, other objects obscuring the face);
- No extreme lighting conditions (i.e., reasonable illuminance, no direct sunlight);
- Steady and well-tuned optics (i.e., no motion blur, depth of field, digital post-processing except noise cancellation).

Cooperative photoshooting is opposite to the so-called “in the wild” photoshooting, which is also called non-cooperative shooting (or recognition).

13.3 Matching

The process of descriptors comparison. Matching is usually implemented as a distance function applied to the feature sets and distances comparison later on. The smaller the distance, the closer are descriptors, hence, the more similar are the objects.

For convenience, helper functions exist to convert distance to a normalized similarity score, where 100% means completely identical, and 0% means completely different.