



VisionLabs
MACHINES CAN SEE

VisionLabs FaceEngine Handbook

written for LUNA SDK Mobile Aurora version 5.25.0

Contents

Introduction	5
1 Core Concepts	6
1.1 SDK workflow	6
1.1.1 Object lifetime	6
1.1.2 Threading	7
1.1.3 Detailed constraints	8
1.2 Common Interfaces and Types	9
1.2.1 Reference Counted Interface	9
1.2.2 Automatic reference counting	10
1.2.2.1 Referencing - without acquiring ownership of object lifetime	10
1.2.2.2 Acquiring - own object lifetime	11
1.2.3 Serializable object interface	11
1.2.4 Auxiliary types	12
1.2.4.1 Image type	12
1.3 Beta Mode	12
2 FaceEngine Structure Overview	13
3 Core Facility	14
3.1 Common Interfaces	14
3.1.1 Face Engine Object	14
3.1.2 Settings Provider	14
3.2 Helper interfaces	14
3.2.1 Archive interface	14
3.3 Data Paths	15
3.3.1 Model Data	15
3.3.2 Configuration Data	15
4 Detection facility	16
4.1 Overview	16
4.2 Detection structure	16
4.3 Face Detection	16
4.3.1 Image coordinate system	16
4.3.2 Face detection	17
4.3.3 Redetect method	17
4.3.4 Face Alignment	17
4.3.4.1 Five landmarks	17

5	Image Warping	18
6	Parameter Estimation Facility	20
6.1	Overview	20
6.2	Best shot selection functionality	20
6.2.1	Eyes Estimation	20
6.2.2	BestShotQuality Estimation	22
6.3	Head Pose Estimation	26
6.4	Approximate Garbage Score Estimation (AGS)	28
6.4.1	LivenessOneShotRGB Estimation	29
6.5	Mouth Estimation Functionality	29
6.6	Face Occlusion Estimation Functionality	33
7	Descriptor processing facility	39
7.1	Overview	39
7.1.1	Person Identification Task	39
7.2	Descriptor	39
7.2.1	Descriptor Versions	40
7.2.2	Descriptor Batch	40
7.2.3	Descriptor Extraction	41
7.2.4	Descriptor Matching	42
8	System Requirements	44
8.1	Aurora installations	44
9	Hardware requirements	45
9.1	Mobile installations	45
9.1.1	CPU requirements	45
9.1.2	Memory requirements	45
9.1.3	Number of threads on mobile devices	46
10	Best practices	47
10.1	Thread pools	47
10.2	Estimator creation and inference	47
10.3	Forking process	47
10.4	Liveness estimator combination	48
10.4.1	Changing the threshold	48
10.4.2	Aggregating the scores	48
10.4.3	Recommended thresholds	48
10.4.4	Possible LivenessOneShotRGBEstimator model combinations	48

11 Device-specific constraints	50
11.1 Image constraints	50
12 Collecting information for Technical Support	51
12.1 Contact Technical Support	51
12.2 Specific error	51
12.3 Non-specific error	52
12.4 Unexpected Result	52
13 Appendix A. Specifications	54
13.1 Runtime performance for mobile environment	54
13.1.1 Aurora	54
13.1.1.1 Aurora environment. Matcher performance	54
13.1.1.2 Aurora environment. Extractor performance	54
13.1.1.3 Aurora environment. Detector performance	55
13.1.1.4 Aurora environment. Estimations performance with batch interface . .	55
13.1.1.5 Aurora environment. Estimations performance without batch interface .	56
13.2 Descriptor size	57
13.3 Feature matrix	57
14 Appendix B. Glossary	59
14.1 Descriptor	59
14.2 Cooperative Photoshooting and Recognition	59
14.3 Matching	59

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.

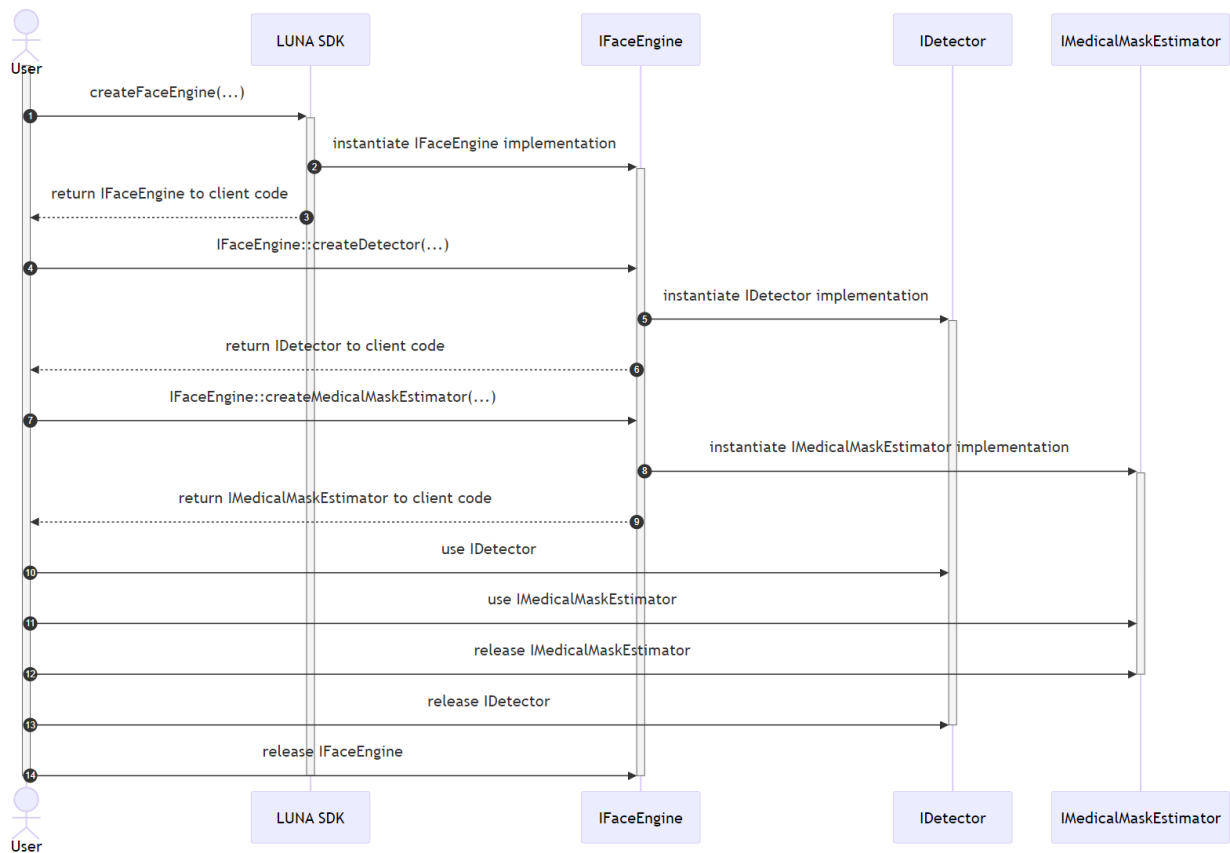
The opening chapter of this book is called “Core concepts”. It will tell you about memory management techniques, object creation and destruction strategies that are widely used across the entire FaceEngine. The following chapters catch up telling how higher level FaceEngine components are created from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

1 Core Concepts

1.1 SDK workflow

1.1.1 Object lifetime

Most of the SDK features are exposed via interfaces (C++ virtual classes) whose implementations must be obtained by calling factory functions. Some of the factories are C-functions, such as `createFaceEngine(...)`. The latter one produces a root object `IFaceEngine`, which in turn exposes many other factories of the `IFaceEngine::createXYZ(...)` form. A typical workflow consists of obtaining `IFaceEngine`, then calling its factories and using the produced child objects.



You do not destroy SDK objects directly, but instead deal with `fsdk::Ref<T>`, reference-counted smart pointers (see section [“Automatic reference counting”](#)) to SDK interfaces. You only need to release all shared references, at which point `fsdk::Ref<T>` destroys the underlying object.

In terms of lifetime, `IFaceEngine` should outlast all its child objects.

Holding `fsdk::Ref<T>` objects in global variables is error-prone. If the variables are in different translation units, their construction order is undefined, which means the destruction order is out of control, too. Viable approaches include gathering all `fsdk::Ref<T>` objects in a single class or using an explicit stack to store them, as well as storing all `fsdk::Ref<T>` as local variables on the call stack in simple projects. In the case when it is necessary to store `fsdk::Ref<T>` objects as global or static

variables, the correct order of releases should be guaranteed explicitly before the program ends:

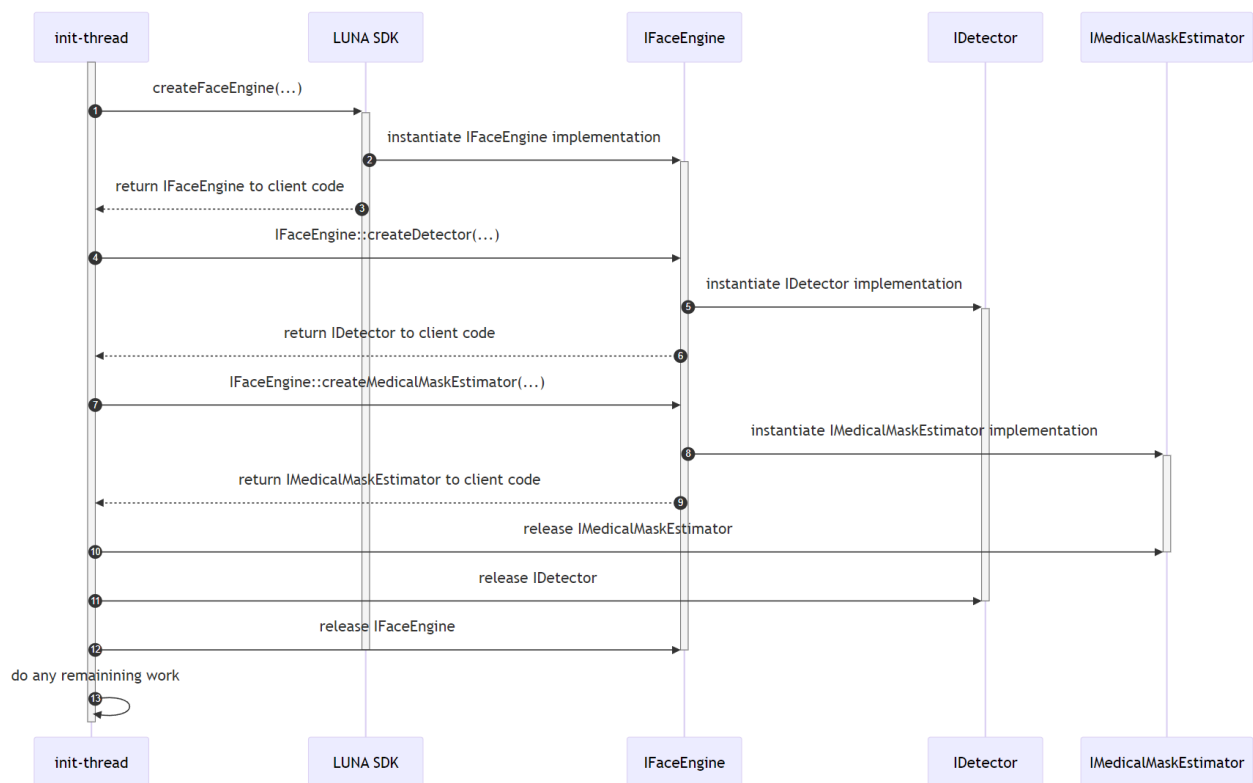
```
//warning: a correct, but not a good example due to these global variables
fsdk::IFaceEnginePtr faceEngine = fsdk::createFaceEngine("./data");
fsdk::IDetectorPtr detector = faceEngine->createDetector();
fsdk::IBestShotQualityEstimator bestShotQualityEstimator = faceEngine->
    createBestShotQualityEstimator();

int main() {
    // application code here

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

1.1.2 Threading

The part of the SDK that instantiates and destroys objects is not thread-safe. The SDK requires using one thread (let's call it `init-thread`) for calling all factory functions, as well as releasing the references to the produced objects. The SDK internally uses thread-local objects attached to `init-thread`, which makes `init-thread` special: as long as the SDK is alive, `init-thread` must be alive too. Therefore, there is a requirement that `init-thread` must outlast `IFaceEngine`.



Once SDK objects (such as detectors and estimators, but not `IFaceEngine`) have been created, they are thread-safe and can be used concurrently and on arbitrary threads. Before using an object concurrently on many threads, consider using asynchronous APIs of the SDK instead. For example, `IDetector` along with a synchronous `detect(...)` function also provides asynchronous `detectAsync(...)`.

It is required that an object cannot be destroyed while it has at least one incomplete call, synchronous or asynchronous, on any thread.

1.1.3 Detailed constraints

Here is a more detailed list of lifetime and threading constraints:

- There should be at most one `IFaceEngine` object per process simultaneously. You can create a new `IFaceEngine` object after destroying the previous one, just avoid holding multiple `IFaceEngine` objects at the same time.
- There should be at most one `ITrackEngine` object per process simultaneously. You can create a new `ITrackEngine` object after destroying the previous one, just avoid holding multiple `ITrackEngine` objects at the same time.
- All factory functions should be called on `init-thread` (the thread that calls `createFaceEngine()`). This also implies that factory code is not thread-safe and all factory calls should be serialized in time. Factory functions include:

- **C-style** functions of the form `createXYZ(...)` such as `createFaceEngine(...)`, `createTrackEngine(...)`
 - **member functions** such as `IFaceEngine::createXYZ(...)`, `ITrackEngine::createXYZ(...)`
- `activateLicense(...)` is not thread-safe. There should be at most one invocation of `activateLicense(...)` per process simultaneously.
 - `init-thread` should live no shorter than `IFaceEngine`.
 - `IFaceEngine` should live no shorter than `ITrackEngine`.
 - `IFaceEngine` should live no shorter than its child objects (algorithms/estimators/detectors). I.e., `IFaceEngine` should be the last destroyed SDK object.
 - `IFaceEngine` should be destroyed on `init-thread`.
 - Algorithms/estimators/detectors should be destroyed on `init-thread`.
 - Algorithms/estimators/detectors can be destroyed when there are no pending or unfinished invocations of member functions of those objects, synchronous or asynchronous, on any threads.
 - Track Engine requirements: all Track Engine streams should be stopped, then destroyed, then `ITrackEngine` itself should be stopped, then destroyed.
 - `ITrackEngine` and all its streams should be destroyed on `init-thread`.

The only part of the SDK that allows multithreading is using member functions of already instantiated algorithms/estimators/detectors, such as `IDetector::detect(...)` and `IAttributeEstimator::estimate(...)`. The member functions can be called on arbitrary threads and in parallel. Before resorting to this multithreaded scenario, please consider using asynchronous versions that accompany many synchronous functions of the SDK.

1.2 Common Interfaces and Types

1.2.1 Reference Counted Interface

Everything in FaceEngine object system starts from here. The *RefCounted* 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, we do not recommend that you interact with the reference counting mechanism manually as doing so may be error-prone. Instead, we recommend that you use smart pointers that are specially designed to handle such objects and provided by FaceEngine. See section [“Automatic reference counting”](#) for details.

1.2.2 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:

1.2.2.1 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.

1.2.2.2 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`.

1.2.3 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*.

1.2.4 Auxiliary types

1.2.4.1 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.

1.3 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 an instance of it. 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.

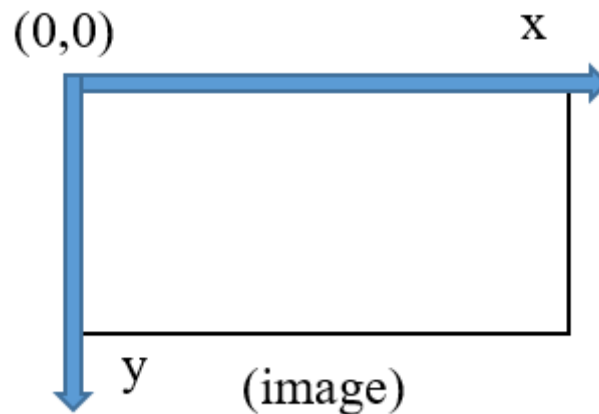


Figure 1: Source image coordinate system

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.

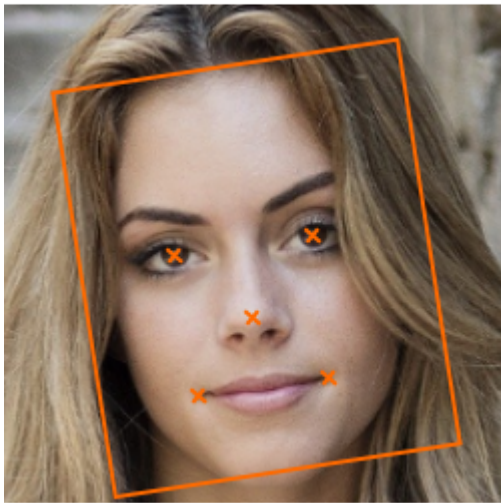


Figure 2: Face warping

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
- 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_cpu.plan
- eye_status_estimation_cpu-avx2.plan
- eye_status_estimation_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.

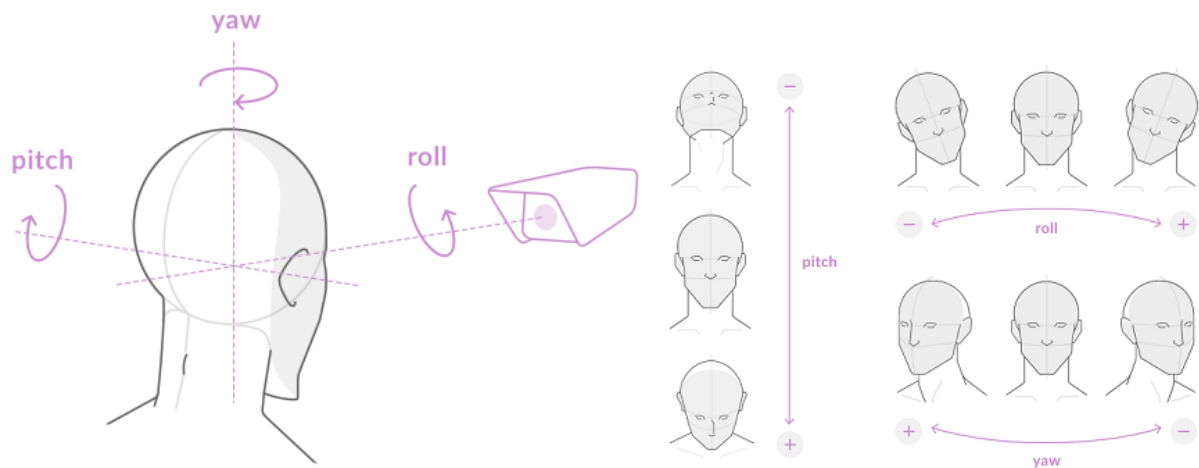


Figure 3: Head pose

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:

For more information see [Approximate Garbage Score Estimation \(AGS\)](#) and [Head Pose Estimation](#)

6.3 Head Pose Estimation

This estimator is designed to determine a camera-space head pose. Since the 3D head translation is hard to reliably determine without a camera-specific calibration, only the 3D rotation component is estimated.

There are two head pose estimation methods available:

- Estimate by 68 face-aligned landmarks. You can get it from the Detector facility, see Chapter “Face detection facility” for details.
- Estimate by the original input image in the RGB format.

An estimation by the image is more precise. If you have already extracted 68 landmarks for another facilities, you can save time and use the fast estimator from 68 landmarks.

By default, all methods are available to use in the faceengine.conf configuration file in section “HeadPoseEstimator”. You can disable these methods to decrease RAM usage and initialization time.

Estimation characteristics:

- Units (degrees)
- Notation (Euler angles)
- Precision (see table 2)

Note: Prediction precision decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table 2.

Table 2: “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 the camera direction, with the axis of symmetry parallel to the vertical camera axis. See figure 4 for a reference.

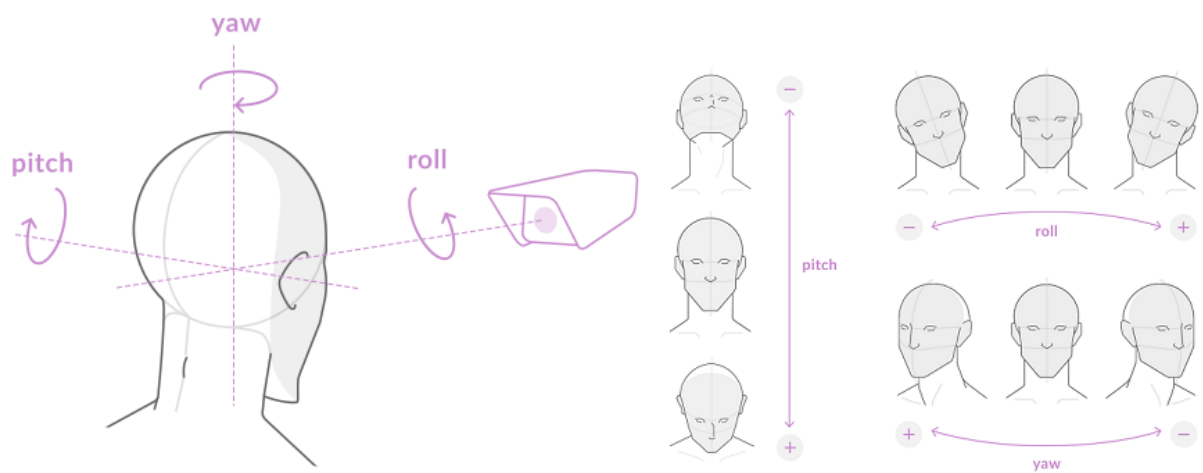


Figure 4: Head pose illustration

Note: In order to work, this estimator requires precise 68-point face alignment results, so familiarize with section “Face alignment” in the “Face detection facility” chapter, as well.

6.4 Approximate Garbage Score Estimation (AGS)

This estimator aims to determine the source image score for further descriptor extraction and matching. The higher the score, 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.

Contact VisionLabs for the recommended threshold value for this parameter.

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

- Implements the `estimate()` function that accepts the source image in the R8G8B8 format and the `fsdk::Detection` structure of corresponding source image. For details, see section “Detection structure” in chapter “Face detection facility”.
- Estimates garbage score of the input image.
- Outputs a garbage score value.

6.4.1 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.

6.5 Mouth Estimation Functionality

Name: MouthEstimator

Algorithm description:

This estimator is designed to predict person's mouth state.

Implementation description:

Mouth Estimation

It returns the following bool flags:

```
bool isOpened;    //!< Mouth is opened flag
bool isSmiling;   //!< Person is smiling flag
bool isOccluded;  //!< Mouth is occluded flag
```

Each of these flags indicate specific mouth state that was predicted.

The combined mouth state is assumed if multiple flags are set to true. For example there are many cases where person is smiling and its mouth is wide open.

Mouth estimator provides score probabilities for mouth states in case user need more detailed information:

```
float opened;     //!< mouth opened score
float smile;      //!< person is smiling score
float occluded;   //!< mouth is occluded score
```

Mouth Estimation Extended

This estimation is extended version of regular Mouth Estimation (see above). In addition, It returns the following fields:

```
SmileTypeScores smileTypeScores; //!< Smile types scores
SmileType smileType; //!< Contains smile type if person "isSmiling"
```

If flag `isSmiling` is true, you can get more detailed information of smile using `smileType` variable. `smileType` can hold following states:

```
enum class SmileType {
    None, //!< No smile
    SmileLips, //!< regular smile, without teeths exposed
    SmileOpen //!< smile with teeths exposed
};
```

If `isSmiling` is false, the `smileType` assigned to `None`. Otherwise, the field will be assigned with `SmileLips` (person is smiling with closed mouth) or `SmileOpen` (person is smiling with open mouth, with teeth's exposed).

Extended mouth estimation provides score probabilities for smile type in case user need more detailed information:

```
struct SmileTypeScores {
    float smileLips; //!< person is smiling with lips score
    float smileOpen; //!< person is smiling with open mouth score
};
```

`smileType` variable is set based on according scores hold by `smileTypeScores` variable - set based on maximum score from `smileLips` and `smileOpen` or to `None` if person not smiling at all.

```
if (estimation.isSmiling)
    estimation.smileType = estimation.smileTypeScores.smileLips >
        estimation.smileTypeScores.smileOpen ?
        fsdk::SmileType::SmileLips : fsdk::SmileType::SmileOpen;
else
    estimation.smileType = fsdk::SmileType::None;
```

When you use Mouth Estimation Extended, the underlying computation are exactly the same as if you use regular Mouth Estimation. The regular Mouth Estimation was retained for backward compatibility.

These estimators are trained to work with warped images (see Chapter [“Image warping”](#) for details).

Recommended thresholds:

The table below contains thresholds specified in the `MouthEstimator::Settings` section of the FaceEngine configuration file (*faceengine.conf*). By default, these threshold values are set to optimal.

Table 3: “Mouth estimator recommended thresholds”

Threshold	Recommended value
occlusionThreshold	0.5
smileThreshold	0.5
openThreshold	0.5

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

- Requirements for Detector:

Attribute	Minimum value
detection size	80

Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

- Requirements for `fsdk::MouthEstimator`:

Attribute	Acceptable values
headPose.pitch	[-20...20]
headPose.yaw	[-25...25]
headPose.roll	[-10...10]

Configurations:

See the “Mouth Estimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

IMouthEstimator

Plan files:

- `mouth_estimation_v4_arm.plan`
- `mouth_estimation_v4_cpu.plan`
- `mouth_estimation_v4_cpu-avx2.plan`
- `mouth_estimation_v4_gpu.plan`

6.6 Face Occlusion Estimation Functionality

Name: FaceOcclusionEstimator

Algorithm description:

This estimator is designed to predict occlusions in different parts of the face, such as the forehead, eyes, nose, mouth, and lower face. It also provides an overall occlusion score.

Implementation description:

Face Occlusion Estimation

The estimator returns the following occlusion states:

```
/**
 * @brief FaceOcclusionType enum.
 * This enum contains all possible facial occlusion types.
 * */
enum class FaceOcclusionType : uint8_t {
    Forehead = 0, //!< Forehead
    LeftEye,      //!< Left eye
    RightEye,     //!< Right eye
    Nose,         //!< Nose
    Mouth,        //!< Mouth
    LowerFace,    //!< Lower part of the face (chin, mouth, etc.)
    Count         //!< Total number of occlusion types
};

/**
 * @brief FaceOcclusionState enum.
 * This enum contains all possible facial occlusion states.
 * */
enum class FaceOcclusionState : uint8_t {
    NotOccluded = 0, //!< Face is not occluded
    Occluded,       //!< Face is occluded
    Count           //!< Total number of states
};

FaceOcclusionState states[static_cast<uint8_t>(FaceOcclusionType::Count)];
    //!< Occlusion states for each face region
float typeScores[static_cast<uint8_t>(FaceOcclusionType::Count)]; //!<
    Probability scores for occlusion types
FaceOcclusionState overallOcclusionState; //!< Overall occlusion state
float overallOcclusionScore;              //!< Overall occlusion score
float hairOcclusionScore;                  //!< Hair occlusion score
```

To get the occlusion score for a specific facial zone, you can use the following method:

```
float getScore(FaceOcclusionType type) const {  
    return typeScores[static_cast<uint8_t>(type)];  
}
```

To get the occlusion state for a specific facial zone, use the following:

```
FaceOcclusionState getState(FaceOcclusionType type) const {  
    return states[static_cast<uint8_t>(type)];  
}
```

This estimator is trained to work with warped images and Landmarks5 (see Chapter [“Image warping”](#) for details).

Recommended thresholds:

The table below contains thresholds specified in the FaceOcclusion::Settings section of the FaceEngine configuration file (faceengine.conf). These values are optimal by default.

Threshold	Recommended value
normalHairCoeff	0.15
overallOcclusionThreshold	0.07
foreheadThreshold	0.2
eyeThreshold	0.15
noseThreshold	0.2
mouthThreshold	0.15
lowerFaceThreshold	0.2

Configurations

See the “Face Occlusion Estimator settings” section in the “ConfigurationGuide.pdf” document.

Filtration parameters:

Name	Threshold
Face Size	>80px
Yaw, Pitch, Roll	±20
Blur (Subjective Quality)	>0.61

API structure name:

IFaceOcclusionEstimator

Plan files:

- face_occlusion_v1_arm.plan
- face_occlusion_v1_cpu.plan
- face_occlusion_v1_cpu-avx2.plan
- face_occlusion_v1_gpu.plan

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 width	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 9: “LivenessOneShotRGB estimator recommended thresholds”

Threshold	Recommended value
realThreshold	0.5
qualityThreshold	0.5
calibrationCoeff	0.89
calibrationCoeff	0.991

Configurations:

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

API structure name:

ILivenessOneShotRGBEstimator

Plan files:

- oneshot_rgb_liveness_v8_model_3_cpu.plan
- oneshot_rgb_liveness_v8_model_4_cpu.plan
- oneshot_rgb_liveness_v8_model_3_arm.plan
- oneshot_rgb_liveness_v8_model_4_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 62 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.

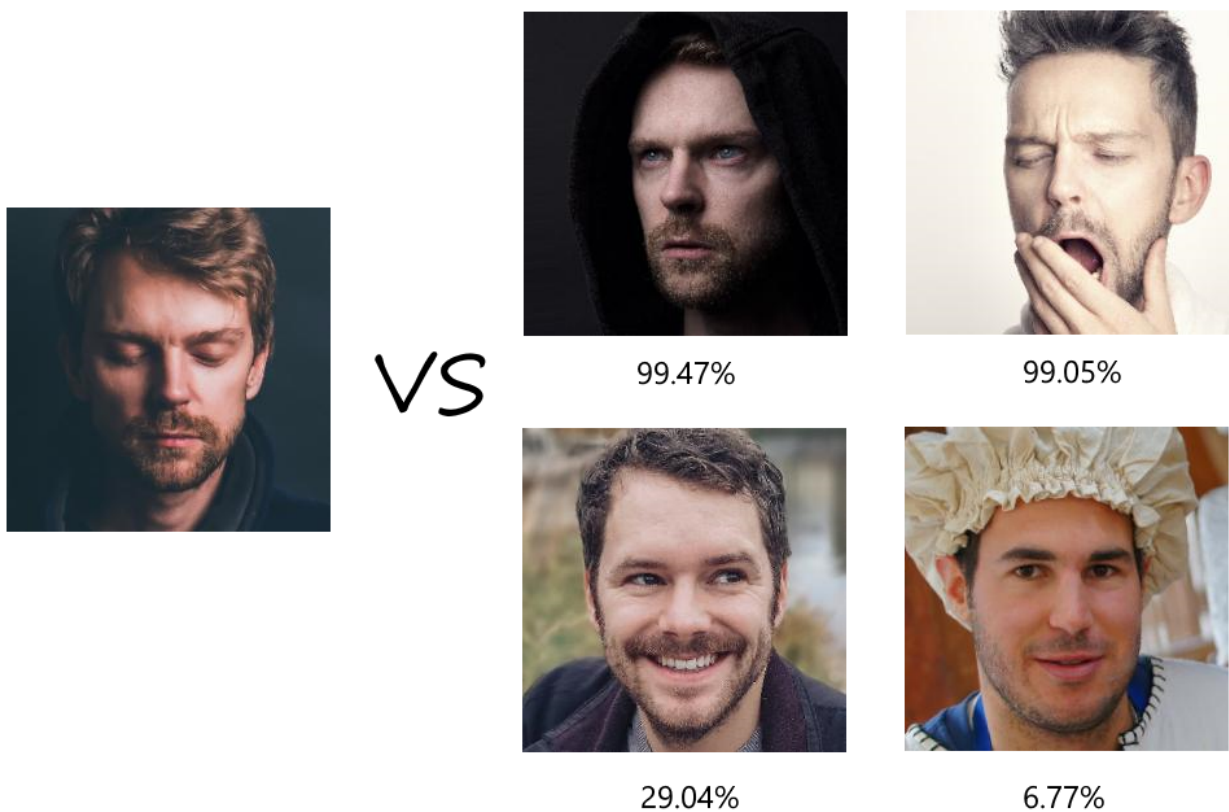


Figure 5: Matching

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 10: Models provided in distribution package and supported devices.

Neural network	CPU	ARM
FaceDet_v2_<detector_type> <i>first</i> <device>.plan	yes	yes
FaceDet_v2_<detector_type> <i>second</i> <device>.plan	yes	yes
FaceDet_v2_<detector_type> <i>third</i> <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_<device>.plan	yes	yes
mouth_estimation_v4_<device>.plan	yes	yes
face_occlusion_v1_<device>.plan	yes	yes
cnn59m_<device>.plan	yes	yes
oneshot_rgb_liveness_v8_model_<model_id>_<device>.plan	yes	yes
vlTracker_detection_<device>.plan	yes	yes
vlTracker_template_<device>.plan	yes	yes
vlTracker_update_<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 11: “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

This section provides a set of recommendations and performance tips that you should follow to get optimal performance when running the LUNA SDK algorithms on your target device.

10.1 Thread pools

We recommend that you use thread pools for user-created threads when running LUNA SDK algorithms in a multithreaded environment. 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, we recommend that you reuse threads from a pool to avoid caching new internal objects and to reduce penalty of creating or destroying new user threads.

10.2 Estimator 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.

10.3 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 one. 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.

Important: 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. For details, see <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.

10.4 Liveness estimator combination

Depending on your device and its camera, you can enhance the accuracy of the model by simultaneously using a combination of two universal liveness estimators. For example, you might use:

- LivenessDepthRGBEstimator and NIRLivenessEstimator
- LivenessDepthEstimator and LivenessOneShotRGBEstimator

To implement this, you need to aggregate the rates from each liveness estimator and adjust the thresholds in the `faceengine.conf` configuration file.

10.4.1 Changing the threshold

All models are calibrated so that the base threshold is 0.5 for any model of any modality.

If you need greater protection against hacking, then the threshold can be raised, and if the convenience of real users is more important, then lowered. We recommend that you configure specific values for changing the threshold in deviation from the basic one on a client basis.

10.4.2 Aggregating the scores

Any of two liveness modalities can be aggregated with each other. To do this, you need to multiply the speeds of the corresponding networks. The threshold in this case is also multiplied and becomes equal to 0.25.

10.4.3 Recommended thresholds

The recommended threshold is an optimal balance between TPR and FPR.

10.4.4 Possible LivenessOneShotRGBEstimator model combinations

You can use the LivenessOneShotRGBEstimator models in the following combinations:

- Use these models in the backend as an analogue of server LivenessOneShotRGBEstimator.
 - `oneshot_rgb_liveness_v8_model_1_cpu-avx2.plan`
 - `oneshot_rgb_liveness_v8_model_2_cpu-avx2.plan`
 - `oneshot_rgb_liveness_v8_model_3_cpu-avx2.plan`
 - `oneshot_rgb_liveness_v8_model_4_cpu-avx2.plan`
- Use these models on smartphones as an analogue of LivenessOneShotRGBEstimator.
 - `oneshot_rgb_liveness_v8_model_3_cpu-avx2.plan`
 - `oneshot_rgb_liveness_v8_model_4_cpu-avx2.plan`
- Use the below model on devices with Orbbec cameras, such as payment terminals (POS) and self-service cash registers (KCO):

- oneshot_rgb_liveness_v8_model_4_cpu-avx2.plan

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 Collecting information for Technical Support

To efficiently resolve a problem with LUNA SDK, collect all necessary information based on the error type and provide it to VisionLabs Technical Support. Possible error types include:

- Specific error
- Non-specific error
- Unexpected result

12.1 Contact Technical Support

You can contact our Technical Support in either of the following ways:

- Via email: support@visionlabs.ai
- Via Service Portal: <https://jira.visionlabs.ru/servicedesk/customer/portal/2>

12.2 Specific error

These errors usually occur when LUNA SDK is used incorrectly. Examples include:

- An estimator or detector does not work, resulting in an error when creating or using it.
- An error occurs when launching on a GPU device.
- A license error is received.

In such cases, study the full launch logs and understand what was launched and where.

To get detailed logging in LUNA SDK, follow these steps:

1❏ In the `luna-sdk/data/runtime.conf` configuration file, set the `verboseLogging` parameter to 4.

```
<param name="verboseLogging" type="Value::Int1" x="4" />
```

2❏ In the `luna-sdk/data/faceengine.conf` configuration file, set the `verboseLogging` parameter to 4.

```
<param name="verboseLogging" type="Value::Int1" x="4" />
```

3❏ In the `luna-sdk/data/trackengine.conf` configuration file, set the `severity` parameter to 0.

```
<param name="severity" type="Value::Int1" x="0" />
```

If you know which module the error occurs in, provide only that module's log by changing the value only in the relevant configuration file. If unsure, collect all logs.

12.3 Non-specific error

Examples of non-specific errors include:

- An application crashes at an uncertain time.
- An application freezes unexpectedly.
- There is a memory leak.

In such cases, you need to understand in detail the application operation scenario, including what is called and in what sequence.

Provide the following information:

- The exact version of LUNA SDK (e.g., v.5.22.2, build for CentOS 8).
- Information about the environment where the application runs (e.g., Docker container, launch via Python bindings).
- [Full launch logs](#).
- Additional information like crash dumps, reports from third-party utilities, and system logs.
- Code reproducing the problem, if any.

12.4 Unexpected Result

Unexpected results may occur due to:

- Incorrect use of LUNA SDK
- Algorithm errors
- Launching in unexpected conditions

Examples include:

- A face is present in a photo or video, but the detector doesn't see it.
- A person is smiling, but the emotion estimator indicates sadness.

Reasons for unexpected results vary, such as:

- Incorrect use of LUNA SDK, for example, a wrong threshold in a configuration file.
- Incorrect input data, such as a poor-quality video or heavily compressed frames.
- Occasional algorithm errors.
- New data for the algorithm.

To understand and address the issue, provide:

- [Full launch logs](#).
- All configuration files used during the launch:
 - luna-sdk/data/runtime.conf
 - luna-sdk/data/faceengine.conf
 - luna-sdk/data/trackengine.conf

- An estimate of how often the unexpected result occurs, for example, every frame or once in a thousand frames.
- Examples of data that produce unexpected results.

13 Appendix A. Specifications

13.1 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;

13.1.1 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.

13.1.1.1 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

13.1.1.2 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

13.1.1.3 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

13.1.1.4 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	404.98
Eyes (RGB, useStatusPlan=0)	auto	1	298.04
Eyes (RGB, useStatusPlan=0)	auto	8	212.24
Eyes (RGB, useStatusPlan=1)	1	1	404.41
Eyes (RGB, useStatusPlan=1)	auto	1	394.3
Eyes (RGB, useStatusPlan=1)	auto	8	212.19
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	443.88
Glasses	auto	1	1298.73
Glasses	auto	8	621.69
Mouth	1	1	4751.0
Mouth	auto	1	3200.3
Mouth	auto	8	2410.8
FaceOcclusion	1	1	3580.21
FaceOcclusion	auto	1	3824.73
FaceOcclusion	auto	4	3292.99
FaceOcclusion	auto	8	2808.52

13.1.1.5 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

13.2 Descriptor size

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

Table 17: “Descriptor size”

Descriptor version	Data size (bytes)	Metadata size (bytes)	Total size
CNN 59 (60)	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.

13.3 Feature matrix

Mobile versions come only in the complete edition.

The table below shows FaceEngine features supported by the complete edition for mobile platforms.

Table 18: “Feature matrix”

Facility	Module	Complete
Core		Yes
Face detection & alignment	Face detector	Yes
Parameter estimation	BestShotQuality estimation	Yes
	Color estimation	Yes
	Eye estimation	Yes
	Head pose estimation	Yes
	AGS estimation	Yes
	LivenessOneShotRGB estimation	Yes

Facility	Module	Complete
	Medical Mask estimation	Yes
	Quality estimation	Yes
	Mouth estimation	Yes
	Glasses estimation	Yes
Face descriptors	Descriptor extraction	Yes
	Descriptor matching	Yes
	Descriptor batching	Yes
	Descriptor search acceleration	Yes

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

14 Appendix B. Glossary

Table 19: 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

14.1 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.

14.2 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).

14.3 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.