

# VisionLabs FaceEngine Handbook

written for LUNA SDK Mobile Android version 4.9.0

## Contents

<b>Introduction</b>	<b>4</b>
<b>1 Core Concepts</b>	<b>5</b>
1.1 Common Interfaces and Types . . . . .	5
1.1.1 Reference Counted Interface . . . . .	5
1.1.2 Automatic reference counting . . . . .	5
1.1.3 Serializable object interface . . . . .	7
1.1.4 Auxiliary types . . . . .	7
1.1.4.1 Image type . . . . .	7
1.2 Beta Mode . . . . .	8
<b>2 FaceEngine Structure Overview</b>	<b>9</b>
<b>3 Core Facility</b>	<b>10</b>
3.1 Common Interfaces . . . . .	10
3.1.1 Face Engine Object . . . . .	10
3.1.2 Settings Provider . . . . .	10
3.2 Helper Interfaces . . . . .	10
3.2.1 Archive Interface . . . . .	10
3.3 Data Paths . . . . .	11
3.3.1 Model Data . . . . .	11
3.3.2 Configuration Data . . . . .	11
<b>4 Detection facility</b>	<b>12</b>
4.1 Overview . . . . .	12
4.2 Detection structure . . . . .	12
4.3 Face Detection . . . . .	12
4.3.1 Image coordinate system . . . . .	12
4.3.2 Face detection . . . . .	13
4.3.3 Redetect method . . . . .	15
4.3.4 Face Alignment . . . . .	15
4.3.4.1 Five landmarks . . . . .	15
<b>5 Parameter Estimation Facility</b>	<b>16</b>
5.1 Overview . . . . .	16
5.2 Eyes Estimation . . . . .	16
5.3 Head pose estimation . . . . .	17
5.4 Approximate Garbage Score Estimation (AGS) . . . . .	18
5.5 BestShotQuality Estimation . . . . .	18

<b>6</b>	<b>Image Warping</b>	<b>20</b>
<b>7</b>	<b>Descriptor Processing Facility</b>	<b>21</b>
7.1	Overview . . . . .	21
7.1.1	Person Identification Task . . . . .	21
7.2	Descriptor . . . . .	21
7.2.1	Descriptor Versions . . . . .	22
7.3	Descriptor Batch . . . . .	22
7.4	Descriptor Extraction . . . . .	23
7.5	Descriptor Matching . . . . .	24
<b>8</b>	<b>System Requirements</b>	<b>25</b>
8.0.1	Android installations . . . . .	25
8.1	Hardware requirements . . . . .	25
8.1.1	Mobile installations . . . . .	25
8.1.1.1	CPU requirements . . . . .	25
8.1.1.2	Memory requirements . . . . .	25
8.1.1.3	Number of threads on mobile devices . . . . .	26
<b>9</b>	<b>Appendix A. Specifications</b>	<b>27</b>
9.1	Runtime performance . . . . .	27
9.1.1	Mobile environment . . . . .	27
9.1.1.1	Android . . . . .	27
9.2	Descriptor size . . . . .	30
<b>10</b>	<b>Appendix B. Glossary</b>	<b>31</b>
10.1	Descriptor . . . . .	31
10.2	Cooperative Photoshooting and Recognition . . . . .	31
10.3	Matching . . . . .	31

## Introduction

This is a short guide that 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.

**Note:** 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.

**Note:** *Ref*<> always increments a reference counter by 1 during initialization. You may be not expecting such behavior from it in some first-time initialization scenarios. Consider a simple example:

```
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 = 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.

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.
*/

```

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

To simply make a reference to existing raw pointer, you may use the `::make_ref()` function pretty much alike to the `::acquire()` function.

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.

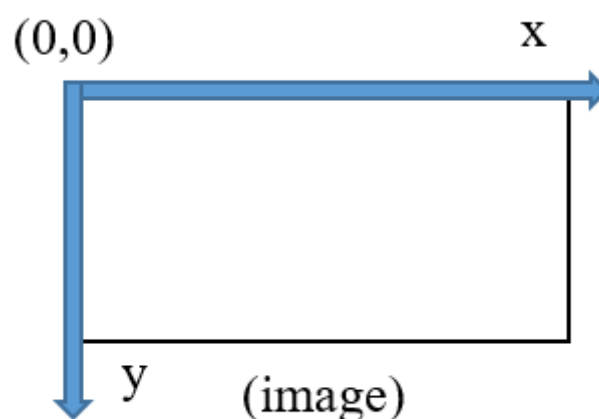
**Note:** Detection score is the measure of classification confidence and not the source image quality. While the score is related to quality (low-quality data generally results in a lower score), it is not a valid metric to estimate the visual quality of an image.

### 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).

#### 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.

When a part of a face is outside of the frame, the detection area will also be beyond the frame borders. Hence coordinates of the detection area may have the following values:

- When the face is beyond the left or the upper border of the frame, the detection coordinates will have negative values;

In the image below, the upper left detection point is outside of the frame. Hence the X and Y coordinates of the upper left detection point have negative values.

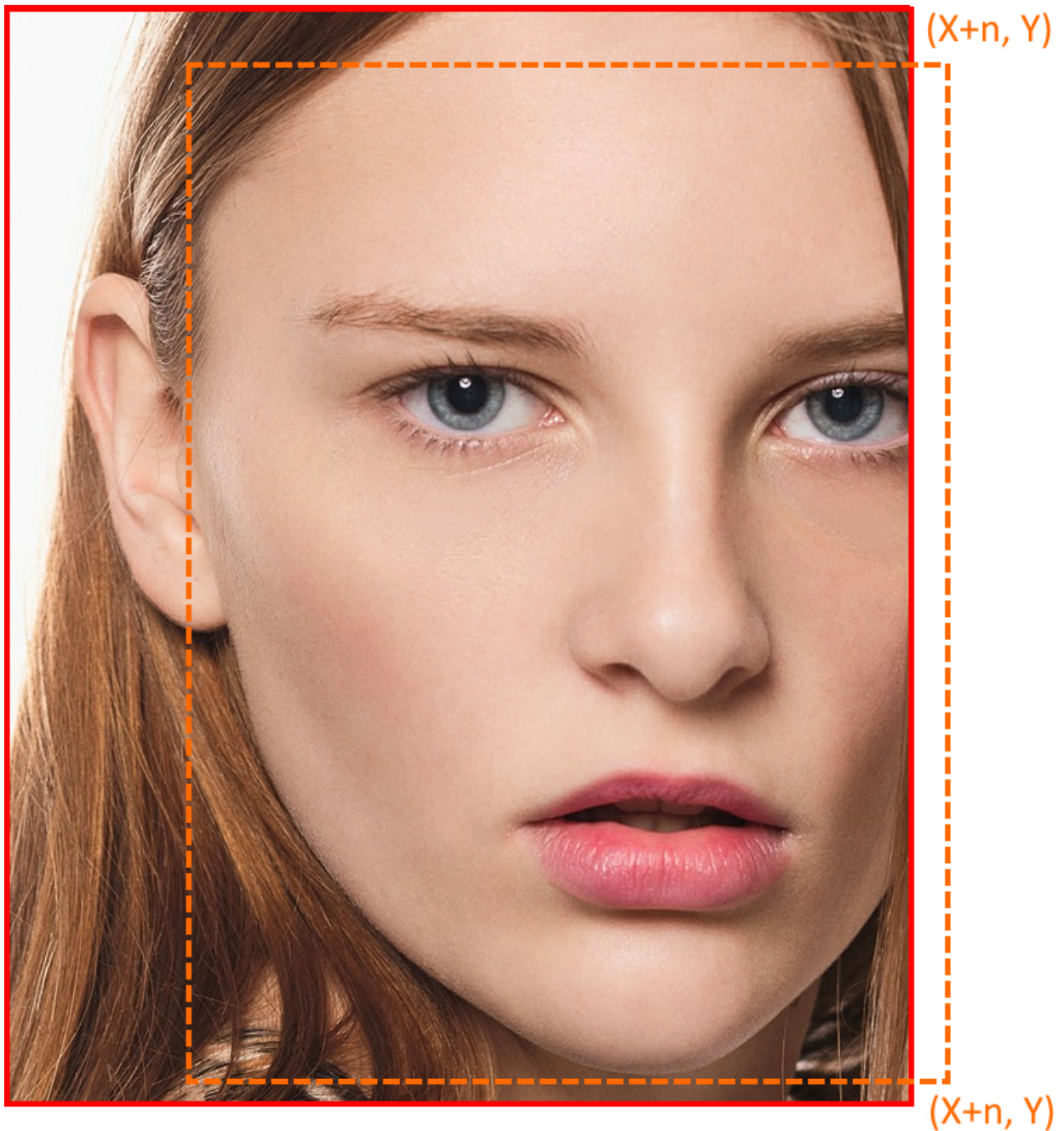


**Figure 2:** Upper left detection point is outside of the frame

- When the face is beyond the right or the lower border of the frame, the detection coordinates will have positive values, but their values will exceed the image size.



In the image below, the X coordinate is equal to  $X + n$ , where  $n$  is the length of the zone that exceeds the image frame size.



**Figure 3:** Lower right detection point is outside of the frame

**NOTE!** You must consider this feature when processing images to properly process the received coordinates.

A code example for detection cropping is given below.

```
const fsdk::Rect brect = detection.rect & image.getRect();
```

detection - face detection. image - source image.

### 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.

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 Parameter Estimation Facility

### 5.1 Overview

Estimation facility is the only multi-purpose facility in FaceEngine. It is designed as a collection of tools that help to estimate various image 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.

### 5.2 Eyes Estimation

**Note.** 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”.

The estimator:

- Implements the *estimate()* function that accepts warped source image (see Chapter “[Image warping](#)”) 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.

**Note.** 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.

### 5.3 Head pose estimation

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

There are two head pose estimation method available:

- Estimate by 68 face-aligned landmarks (you may get it from Detector facility, see Chapter “Detection facility”);
- Estimate by original input image in RGB format.

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

By default, all methods are available to use in config (faceengine.conf) in section “HeadPoseEstimator”. You may disable these methods to decrease RAM usage and initialization time.

Estimation characteristics:

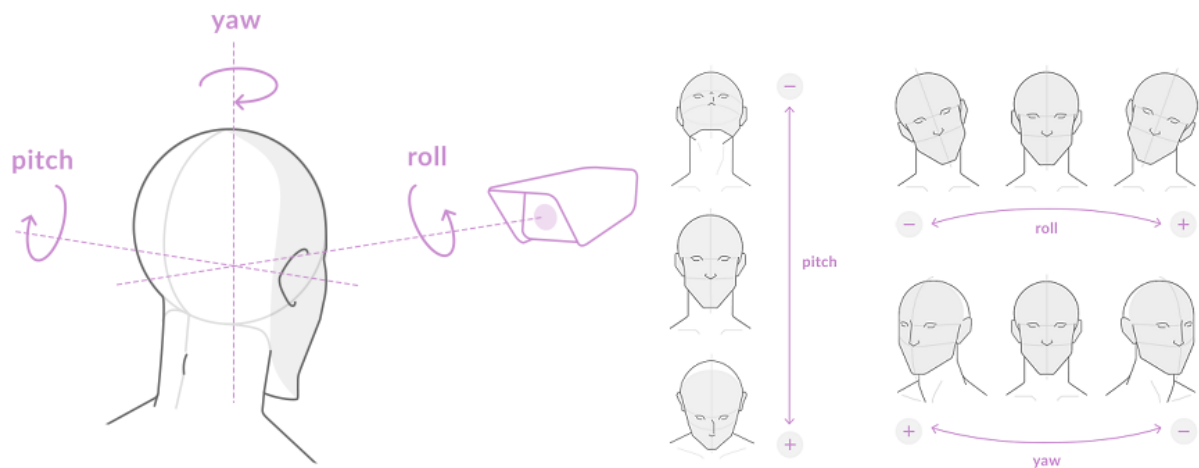
- Units (degrees);
- Notation (Euler angles);
- Precision (see the table below).

**Note.** 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. See the image below for a reference.



**Figure 4:** Head pose

**Note.** In order to work, this estimator requires precise 68-point face alignment results, so familiarize with section “Face alignment” in the “[Detection facility](#)” chapter as well.

## 5.4 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.

Consult VisionLabs about the recommended threshold value for this parameter.

The estimator (see IAGSEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts source image in R8G8B8 format and fsdk::Detection structure of corresponding source image (see section “[Detection structure](#)” in chapter “[Detection facility](#)”);
- Estimates garbage score of input image;
- Outputs garbage score value.

## 5.5 BestShotQuality Estimation

The BestShotQuality estimator represents a collection of estimator functionalities unified for end-user convenience.

Estimation types that were merged into this estimator are described in the following list:

- AGS: image quality score (see section “[Approximate garbage score estimation \(AGS\)](#)” for more details);

- HeadPose: determines person head rotation angles in 3D space, namely pitch, yaw and roll (see section [Head pose estimation](#) for more details).

Before using this estimator, user is free to decide whether to estimate or not some specific attributes listed above through *IBestShotQualityEstimator::EstimationRequests* structure, which later get passed in main *estimate()* method.

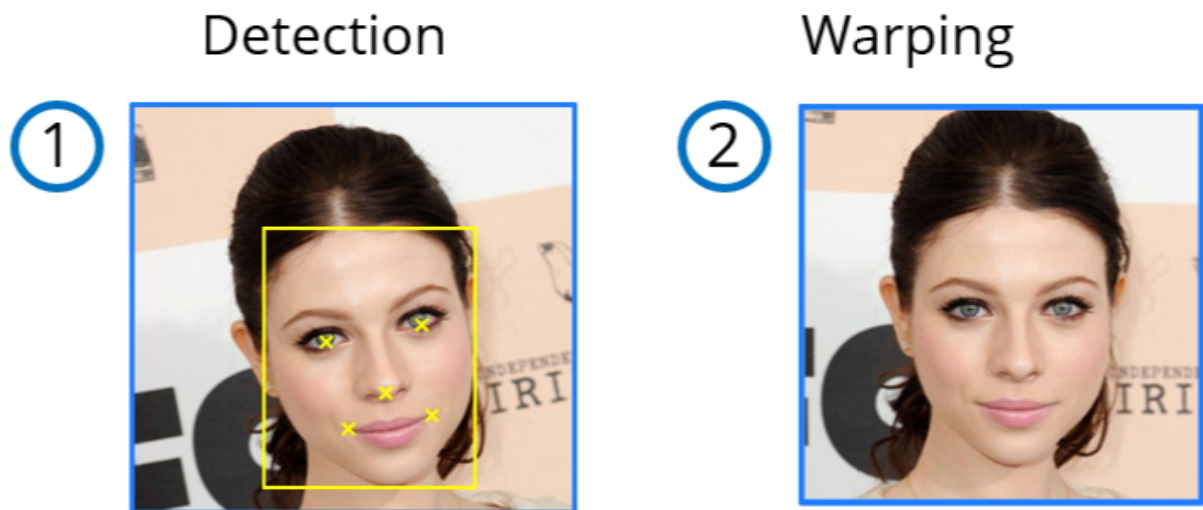
Estimator overrides *AQEEstimationResults* output structure, which consists of optional fields describing results of user requested attributes.

## 6 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.



**Figure 5:** Face warping

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

## 7 Descriptor Processing Facility

### 7.1 Overview

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

Note: 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.

**Note.** *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 the performance and precision of different descriptor versions.

Descriptor version 59 is the best one by precision. And it works well 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”](#)).

## 7.3 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.

## 7.4 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.

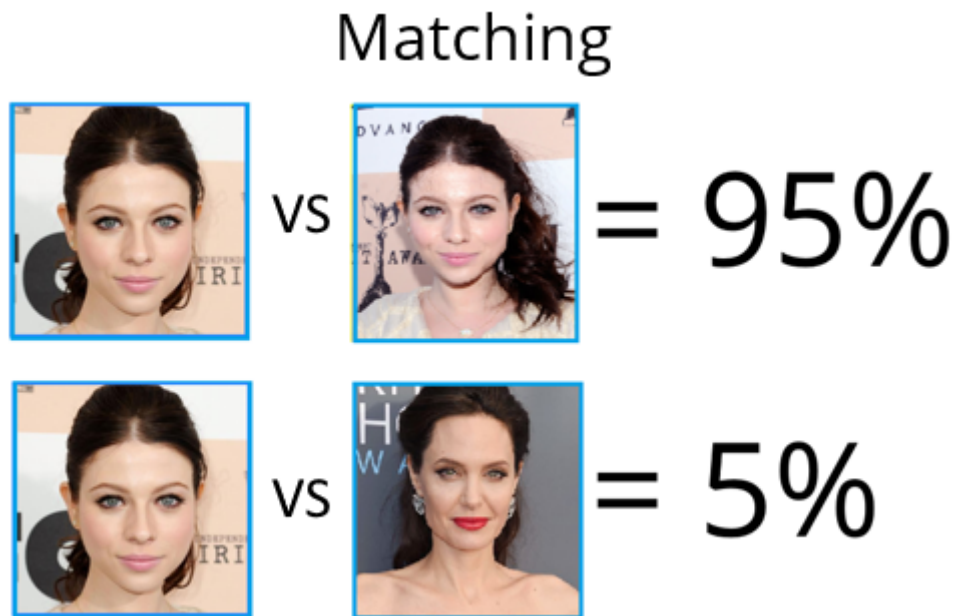
It should be noted, that the face detection area and the landmarks are required only for image warping, the preparation stage for descriptor extraction (see section “[Image warping](#)”). If the source image is already warped, it is possible to skip these parameters. For that purpose, the *IDescriptorExtractor*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

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.

## 7.5 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 6:** 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

### 8.0.1 Android installations

FaceEngine requires:

- Android version 4.4.4 or newer.

For development:

- Android SDK 21;
- Android NDK 21 {Pkg.Revision = 21.0.6113669}.

**Note:** Android development dependencies listed above can be downloaded directly from SDK manager in Android Studio IDE or via SDK manager command line tool. For more information, please visit <https://developer.android.com/studio/command-line/sdkmanager>.

## 8.1 Hardware requirements

### 8.1.1 Mobile installations

#### 8.1.1.1 CPU requirements

Supported CPU architectures:

- x86;
- x86\_64;
- armeabi-v7a;
- arm64-v8a.

**Note:** Per-abi libraries are provided for Android.

#### 8.1.1.2 Memory requirements

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

Storage is amount of space specific version of installation takes on device. For Android app Gradle build system strips symbols from all the dynamic libraries when building a release *.apk*. As the result *.so* files in your final app archive will occupy (up to 30-60%, depending on platform) less storage space compared to ones found in the distribution.

**Table 2:** “Memory requirements”

Requirements for	Android
RAM	400 MB
Storage Full	350 MB

Requirements for	Android
Storage Frontend	300 MB

#### 8.1.1.3 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 number of available processor cores. We strongly recommend you to follow this recommendation; otherwise, performance can be significantly reduced.

## 9 Appendix A. Specifications

### 9.1 Runtime performance

#### 9.1.1 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 Android platform uses mobilenet by default.

Input data characteristics:

- Image resolution: 640x480px;
- Image format: 24 BPP RGB;
- Typical face size: ~260x260px.

##### 9.1.1.1 Android

Performance measurements are presented for ARM of Samsung SM-G930F and Samsung SM-J730FM in tables below. Measured values are averages of at least 100 experiments. Mobilenet is used by default. 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 accoding settings you can find in “Configuration Guide - Runtime settings”.

**Table 3:** “Performance of Samsung SM-G930F. Extractor and matcher”

Measurement	Model	Threads	Average	Units
Extractor	56	1	622.4	ms
	56	auto	336.4	
Matcher	56	-	60 K	matches/sec

**Table 4:** “Performance of Samsung SM-G930F. Extractor batch”

Measurement	Model	Threads	Average (ms)	Batch Size
Extractor Batch	56	auto	230.0	1
	56	auto	301.4	4
	56	auto	276.1	8

**Table 5:** “Performance of Samsung SM-G930F. Detection and estimation”

Measurement	Threads	Average (ms)	BatchSize
Detector (FaceDetV2)	1	78.9 / 70.8 / 286.0	-
(Easy/complex/6 faces)	auto	80.2 / 70.8 / 149.5	-
Warper	1	8.4	-
	auto	8.2	-
Head Pose by Image	1	6.5	-
	auto	12.3	-
Head Pose Batch	auto	12.0	1
	auto	5.0	4
	auto	3.7	8
Eyes	1	24.3	-
	auto	26.9	-
Eyes Batch	auto	26.5	1
	auto	16.9	4
	auto	12.9	8
AGS	1	6.4	-
	auto	10.0	-
AGS Batch	auto	10.1	1
	auto	4.4	4
	auto	3.5	8
BestShot Quality	1	12.8	-
	auto	23.3	-
BestShot Quality Batch	auto	21.7	1
	auto	9.5	4
	auto	7.3	8

**Table 6:** “Performance of Samsung SM-J730FM. Extractor and matcher”

Measurement	Model	Threads	Average	Units
Extractor	56	1	1092.4	ms
	56	auto	290.0	
Matcher	56	-	60 K	matches/sec

**Table 7:** “Performance of Samsung SM-J730FM. Extractor batch”

Measurement	Model	Threads	Average (ms)	Batch Size
Extractor Batch	56	auto	159.5	1
	56	auto	164.2	4
	56	auto	162.2	8

**Table 8:** “Performance of Samsung SM-J730FM. Detection and estimation”

Measurement	Threads	Average(ms)	BatchSize
Detector (FaceDetV2)	1	89.6 / 82.2 / 321.3	-
(Easy/complex/6 faces)	auto	100.2 / 82.5 / 144.2	-
Warper	1	13.4	-
	auto	13.4	-
Head Pose by Image	1	6.2	-
	auto	9.0	-
Head Pose Batch	auto	7.6	1
	auto	4.4	4
	auto	2.9	8
Eyes	1	32.2	-
	auto	28.4	-
Eyes Batch	1	34.1	1
	auto	38.7	8

Measurement	Threads	Average(ms)	BatchSize
AGS	auto	7.4	-
	auto	4.0	-
	auto	2.9	-
AGS Batch	1	25.2	1
	auto	17.9	8
BestShot Quality	auto	14.1	1
	auto	7.0	1
	auto	5.8	1
BestShot Quality Batch	auto	28.6	1
	auto	17.3	4
	auto	11.8	8

## 9.2 Descriptor size

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

**Table 9:** “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.

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

## 10 Appendix B. Glossary

### 10.1 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.

### 10.2 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).

### 10.3 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.