VisionLabs
MACHINES CAN SEE

VisionLabs FaceEngine Handbook

VisionLabs B.V.

Keizersgracht 311, 1016 EE, Amsterdam, the Netherlands

+31 20 369 04 93

info@visionlabs.ai

www.visionlabs.ai

Contents

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. Starting from chapter “Licensing” this book covers less code-oriented topics like licensing and system requirements.

Editions and Platforms

FaceEngine supports multiple platforms and comes in two editions: the frontend edition (or FE for short) and the complete edition. Supported software and hardware platforms differ depending on editions.

The Frontend edition is intended for lightweight software that does not need to implement searching functions. For example, these could be:

- Face detection applications that take a picture of the user and send it to a server for processing, such as verification;
- Face alignment applications, that only need the knowledge about head pose and facial shape;
- Simple ad-hoc analytics solutions like age & gender recognition for context-aware advertising;
- And so on.

The complete edition contains all the features of the frontend edition but adds face verification and identification features. That said, the complete edition is a more of a backend or server-oriented software. Still, it can run not only on powerful servers, but on ordinary PCs and even mobile devices as well.

This document covers the entire set of implemented features, that is - the complete edition. In the distribution section you can find a feature matrix that shows what exact algorithms are implemented in each edition and what platforms it supports.

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 *typedef*s for Ref's to all reference counted types are declared. All of them match the following naming convention: *InterfaceName*Ptr. 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 *SubImage*s longer that it is necessary to complete data modification. See the documentation of the *map()* function family for details.

The supports IO roitines 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 *Image*s 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.

Note: if you plan to use GPU acceleration, you should keep in mind CUDA runtime initialization and shutdown. Specifically, CUDA creates global runtime object with implicit lifetime; see [http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#initialization]. To prevent FaceEngine's runtime and lifetime mismatch, it is recommended to avoid creating static global instances of FaceEngine objects, as their destruction order is undetermined.

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. The data directory location is assumed to reside in:

- /opt/visionlabs/data on Linux
- ./data on Windows

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.

Special estimators exist to fulfill this task (see section "Image quality estimation" in chapter "Parameter estimation facility" for details).

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.

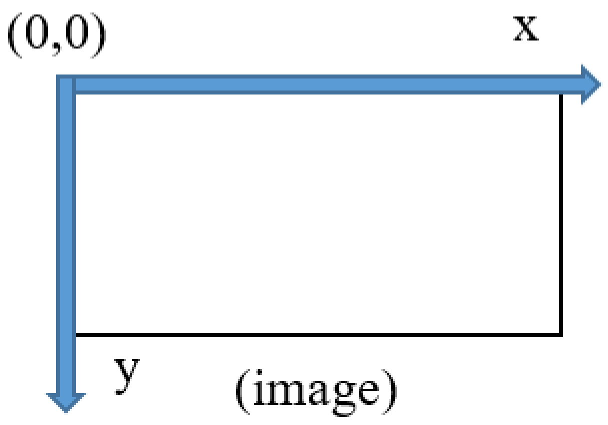

Figure 1: Source image coordinate system

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.

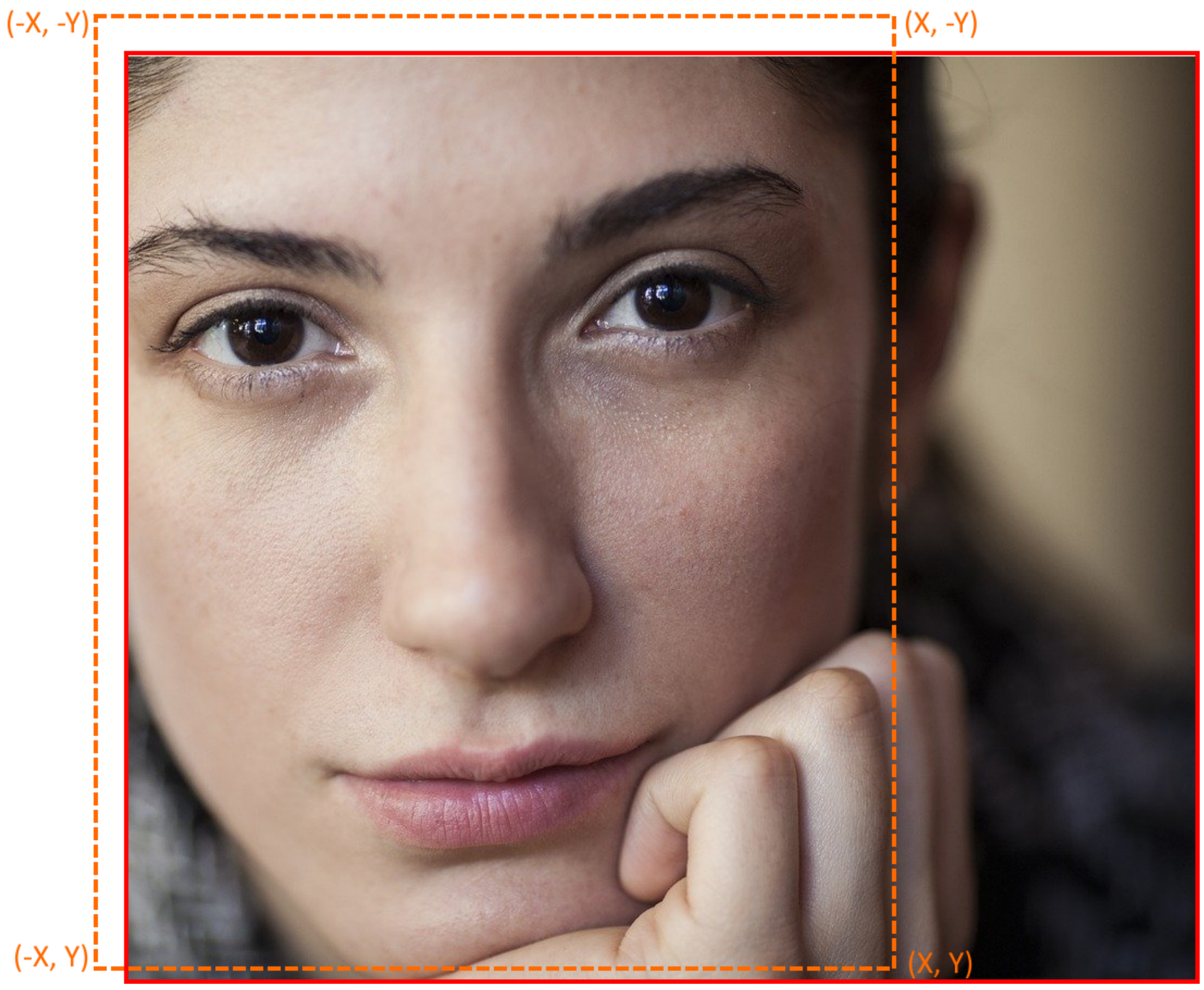

Figure 2: Upper left detection point is outside of the frame

- 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.

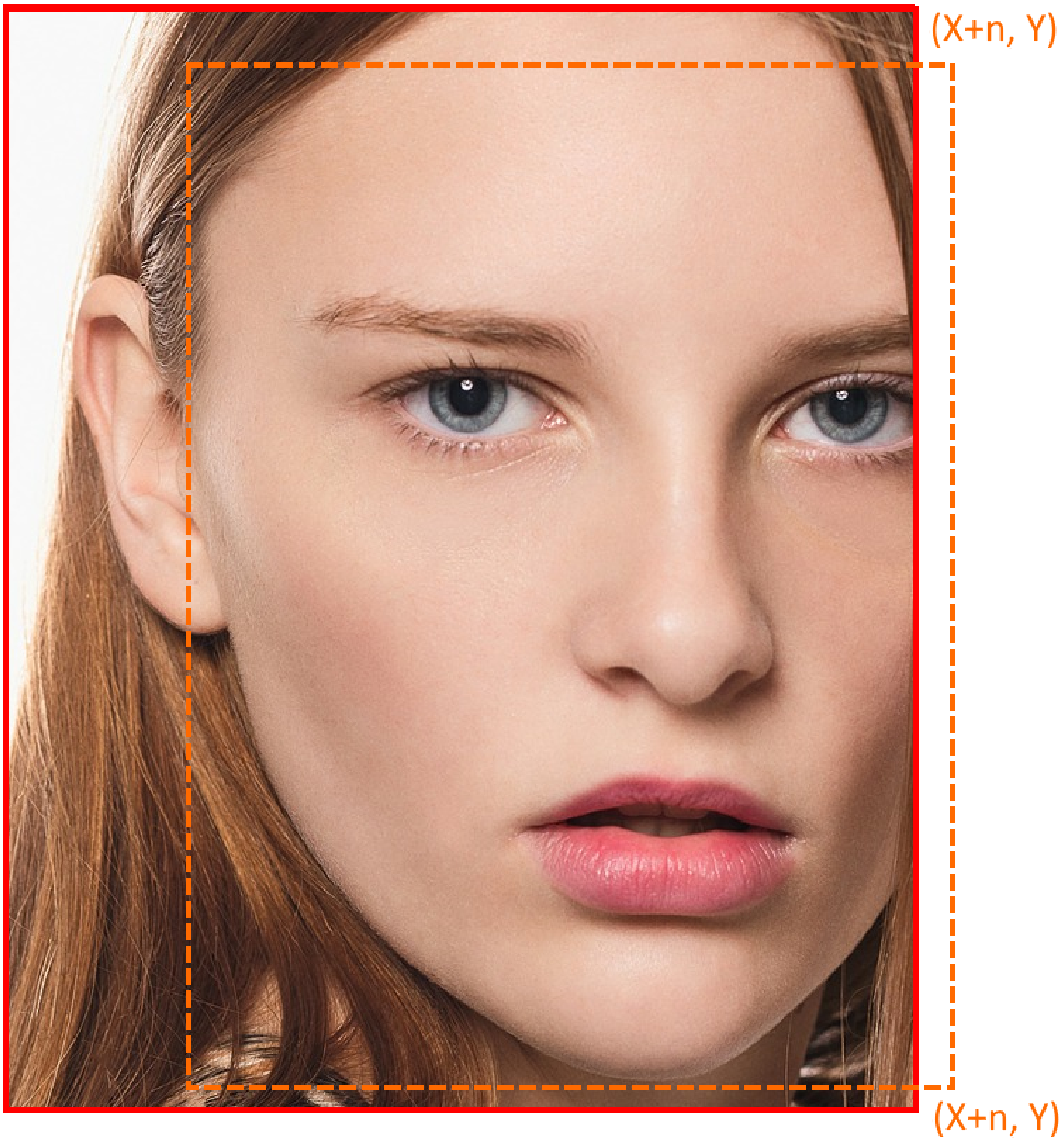

Figure 3: Lower right detection point is outside of the frame

> **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 Detector variants

Supported detector variants:

- FaceDetV1
- FaceDetV2
- FaceDetV3

There are two basic detector families. The first of them includes two detector variants: FaceDetV1 and FaceDetV2. The second family currently includes only one detector variant - FaceDetV3. FaceDetV3 is the latest and most precise detector. In terms of performance it is similar to FaceDetV1 detector. User code may specify necessary detector type while creating *IDetector* object using parameter.

FaceDetV3 supports orientation mode and can estimate orientation of whole image (Normal, Right90deg, Left90deg or Upsidesown). Config option `useOrientation` should be `1` (see Configuration guide). You can estimate orientation of image by calling method `estimateOrientation` of Detector. Or orientation will be automatically estimated while regular detection call if `useOrientation` was turned of. Detector estimate orientation in the beginning, then flip image if it necessary and detect on the correct oriented image. **Note:** Correct oriented image will be stored in Face.img field (use this field in future processing). Detection and landmarks coordinates are given in correct oriented image coordinates.

Note: FaceDetV1 and FaceDetV2 performance depends on number of faces on image and image complexity. FaceDetV3 performance depends only on the target image resolution.

Note: FaceDetV3 works faster with batched redetect.

4.3.5 FaceDetV1 and FaceDetV2 Configuration

FaceDetV1 detector is more precise and FaceDetV2 works two times faster (See “Appendix A. Specifications”).

FaceDetV1 and FaceDetV2 detector’s performance depend on number of faces in image. FaceDetV3 doesn’t depend on it, so it may be slower then FaceDetV1 on images with one face and much more faster on images with many faces.

4.3.6 FaceDetV3 Configurating

FaceDetV3 detects faces from `minFaceSize` to`maxFaceSize` (Note: `maxFaceSize` <=`minFaceSize` * 32). You can change the minimum and maximum sizes of the faces that will be searched in the photo from the `faceengine.conf` configuration.

For example:

```
config->setValue("FaceDetV3::Settings", "minFaceSize", 20);
```

The logic of the detector is very understandable. The smaller the face size we need to find the more time we need.

We recommend to use such meanings for `minFaceSize`: 20, 40 and 90. The size 90 pix is recommended for recognition. If you want to find faces with custom size value you will need to point with size with: `95% * value`. For example we want to find faces with size of 50 pix, it means that in config we should set: `50 * 0.95 ~ 47 pix`.

FaceDetV3 supports image orientation determining. Three main angles of image rotation are presented: 90, 180 and 270 degrees. In the case of rotated origin image the rectangles of detection and landmarks will be returned in origin coordinate system. For example if image was rotated on 90 degrees rectangles of detections and landmarks will be rotated on 90 degrees too. The total time for such detection will be 2 times longer comparably with detection without orientation defining. Mode of image orientation is switching on from `faceengine.conf` by setting `useOrientationMode`.

Note: FaceDetV3 may provide accurate *5 landmarks* only for faces with size greater then 40x40, for smaller faces it provides less accurate landmarks. If you have few faces on target images and target face sizes after resize will less then 40x40, it's recommended to require *68 landmarks*. If you have many faces on target image (greater then 7) it will be faster increase `minFaceSize` to have big enough faces for accurate landmarks estimation.

All last changes in Face Detection logic are described in `Handbook/Chapter 10 Migration guide`.

4.3.7 Face Alignment

4.3.7.1 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.

4.3.7.2 Sixty-eight landmarks

More advanced **68-points** face alignment is also implemented. Use this when you need precise information about face and its parts. The detected points look like in the image below.

The *68 landmarks* require additional computation time, so don't use it if you don't need precise information about a face. If you use *68 landmarks* , *5 landmarks* will be reassigned to more precise subset of *68 landmarks*.

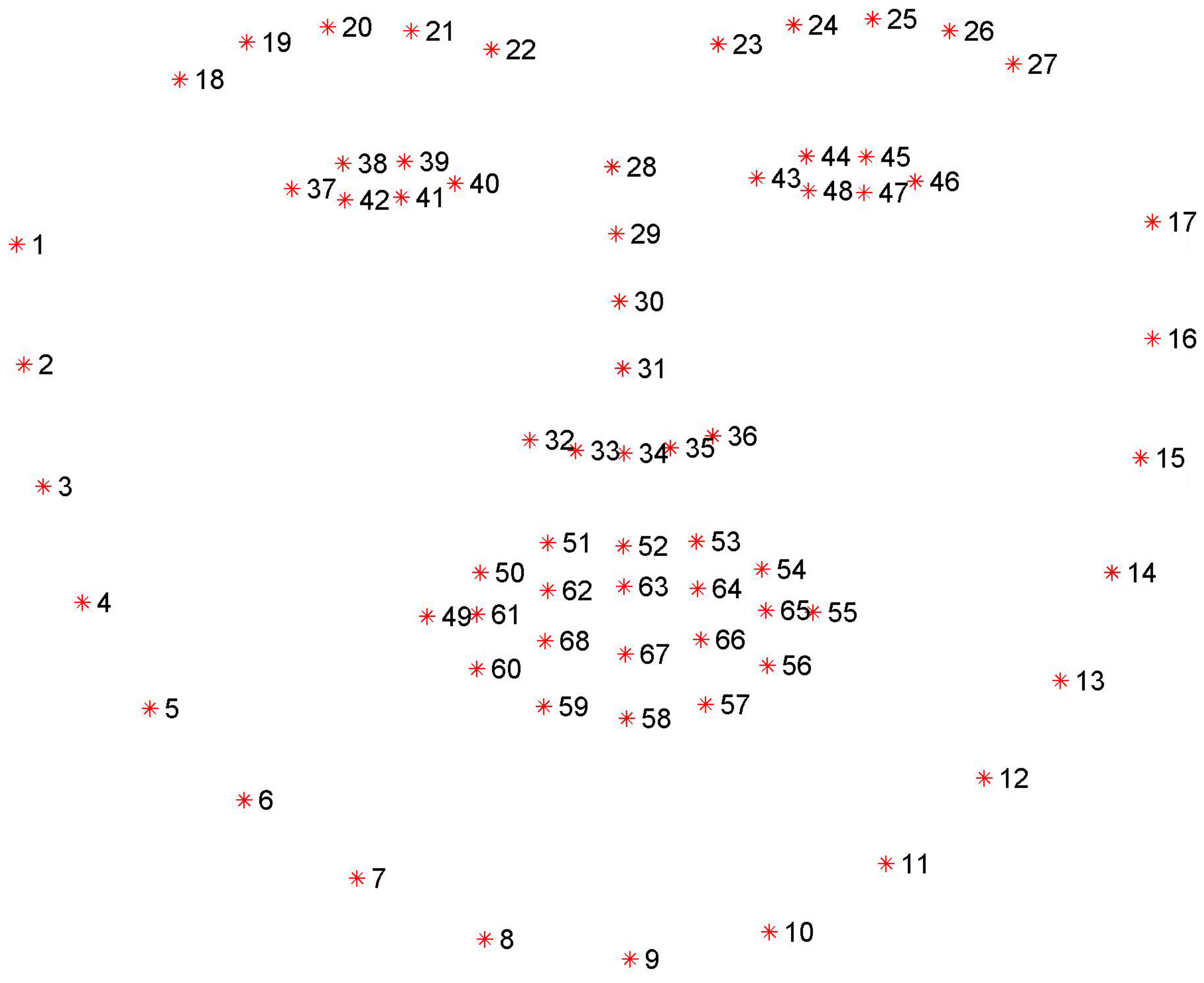

Figure 4: 68-point face alignment

The typical error for landmark estimation on a warped image (see Chapter "Image warping") is in the table below.

Table 1: "Average point estimation error per landmark"

Point	Error (pixels)	Point	Error (pixels)	Point	Error (pixels)	Point	Error (pixels)
1	±3,88	18	±3,77	35	±1,62	52	±1,65
2	±3,53	19	±2,83	36	±1,90	53	±2,01
3	±3,88	20	±2,70	37	±1,78	54	±2,00
4	±4,30	21	±3,06	38	±1,69	55	±1,93
5	±4,67	22	±3,92	39	±1,63	56	±2,18
6	±4,87	23	±3,46	40	±1,52	57	±2,17
7	±4,67	24	±2,59	41	±1,54	58	±1,99
8	±4,01	25	±2,53	42	±1,60	59	±2,32
9	±3,46	26	±2,95	43	±1,55	60	±2,33
10	±3,87	27	±3,84	44	±1,60	61	±2,06
11	±4,56	28	±1,88	45	±1,74	62	±1,97
12	±4,94	29	±1,75	46	±1,72	63	±1,56
13	±4,55	30	±1,92	47	±1,68	64	±1,86
14	±4,45	31	±2,20	48	±1,65	65	±1,94
15	±4,13	32	±1,97	49	±1,99	66	±2,00
16	±3,68	33	±1,70	50	±1,99	67	±1,70
17	±4,09	34	±1,73	51	±1,95	68	±2,12

Simple 5-point landmarks roughly correspond to:

- Average of positions 37, 40 for a left eye;
- Average of positions 43, 46 for a right eye;
- Number 31 for a nose tip;
- Numbers 49 and 55 for mouth corners.

The landmarks for both cases are output by the face detector via Landmarks5 and Landmarks68 structures. Note, that performance-wise 5-point alignment result comes free with a face detection, whereas 68-point result does not. So you should generally request the lowest number of points for your task.

Typical use cases for 68 landmarks:

- Segmentation;
- Head pose estimation.

4.4 Human Detection

This functionality enables you to detect human bodies in the image.

During the detection process we receive special points (called "landmarks" or exactly "HumanLandmarks17") for the body parts visible in the image. These landmarks represent the keypoints of a human body (see the Human keypoints section).

Human body detection is performed by the *IHumanDetector* object. The function of interest is *detect()*. It requires an image to detect on.

4.4.1 Image coordinate system

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

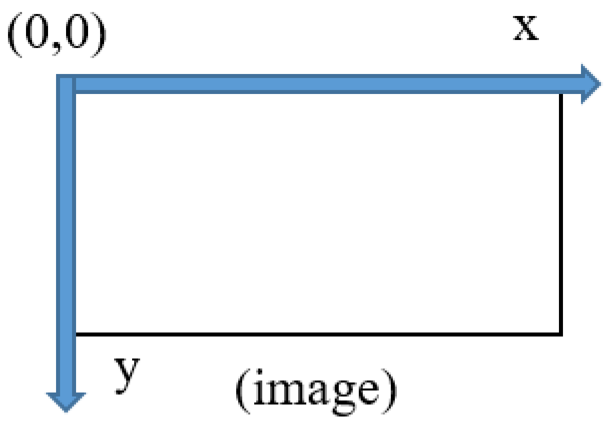

Figure 5: Source image coordinate system

4.4.2 Human body detection

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

When a part of a human body 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 human body is beyond the left or the upper border of the frame, the detection coordinates will have negative values;
- When the human body 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 upper left detection point and lower right points are outside of the frame.

The X and Y coordinates of the upper left detection point have negative values. The Y coordinate of the lower right detection point is equal to Y + n, where n is the length of the zone that exceeds the image frame size.

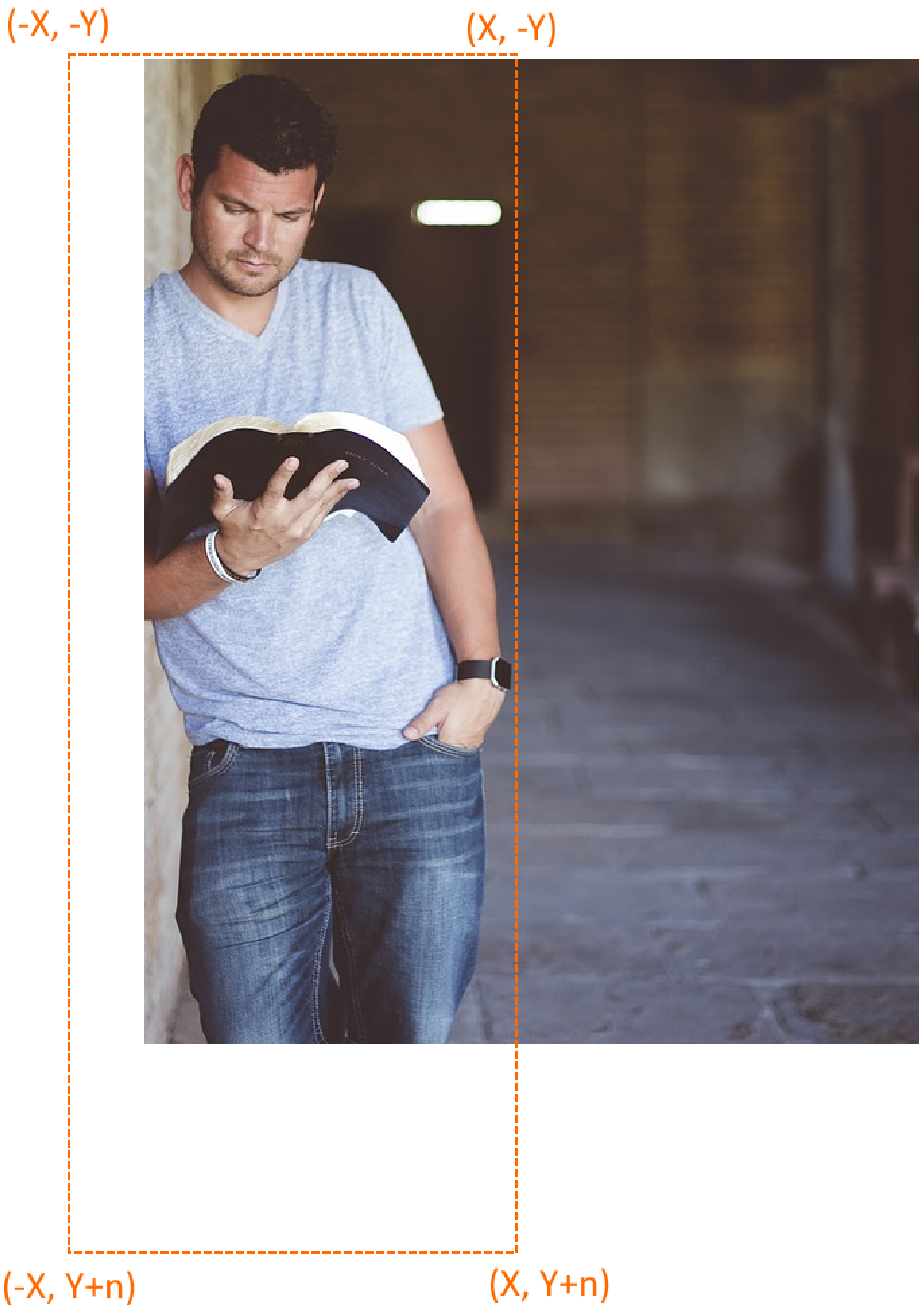

Figure 6: Detection points are outside of the frame

> **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 - human body detection. image - source image.

4.4.3 Constraints

Human body detection has the following constraints:

- Human body detector works correctly only with adult humans in an image;
- The detector may detect a body of size from 100 px to 640 px (in an image with a long side of 640 px). You may change the input image size in the config (see ./doc/ConfigurationGuide.pdf). The image will be resized to specified size by the larger side while maintaining the aspect ratio.

4.4.4 Camera position requirements

In general, you should locate the camera for human detection according to the image below.

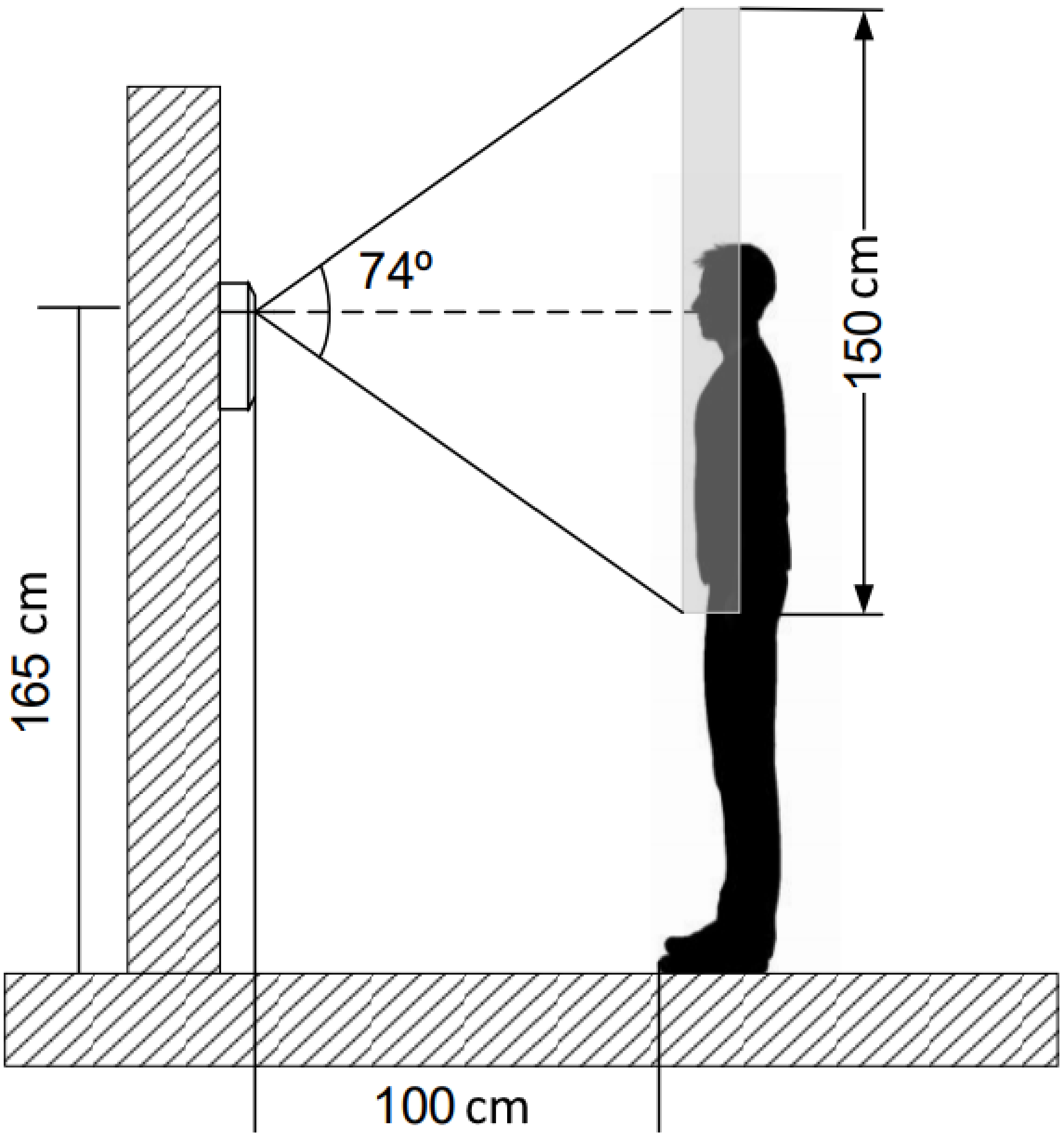

Figure 7: Camera position for human detection

Follow these recommendations to correctly detect human body and keypoints:

- A person's body should face the camera;
- Keep angle of view as close to horizontal as possible;
- There should be about 60% of the person's body in the frame (upper body);
- There must not be any objects that overlap the person's body in the frame;
- The camera should be located at about 165 cm from the floor, which corresponds to the average

height of a human.

The examples of wrong camera positions are shown in the image below.

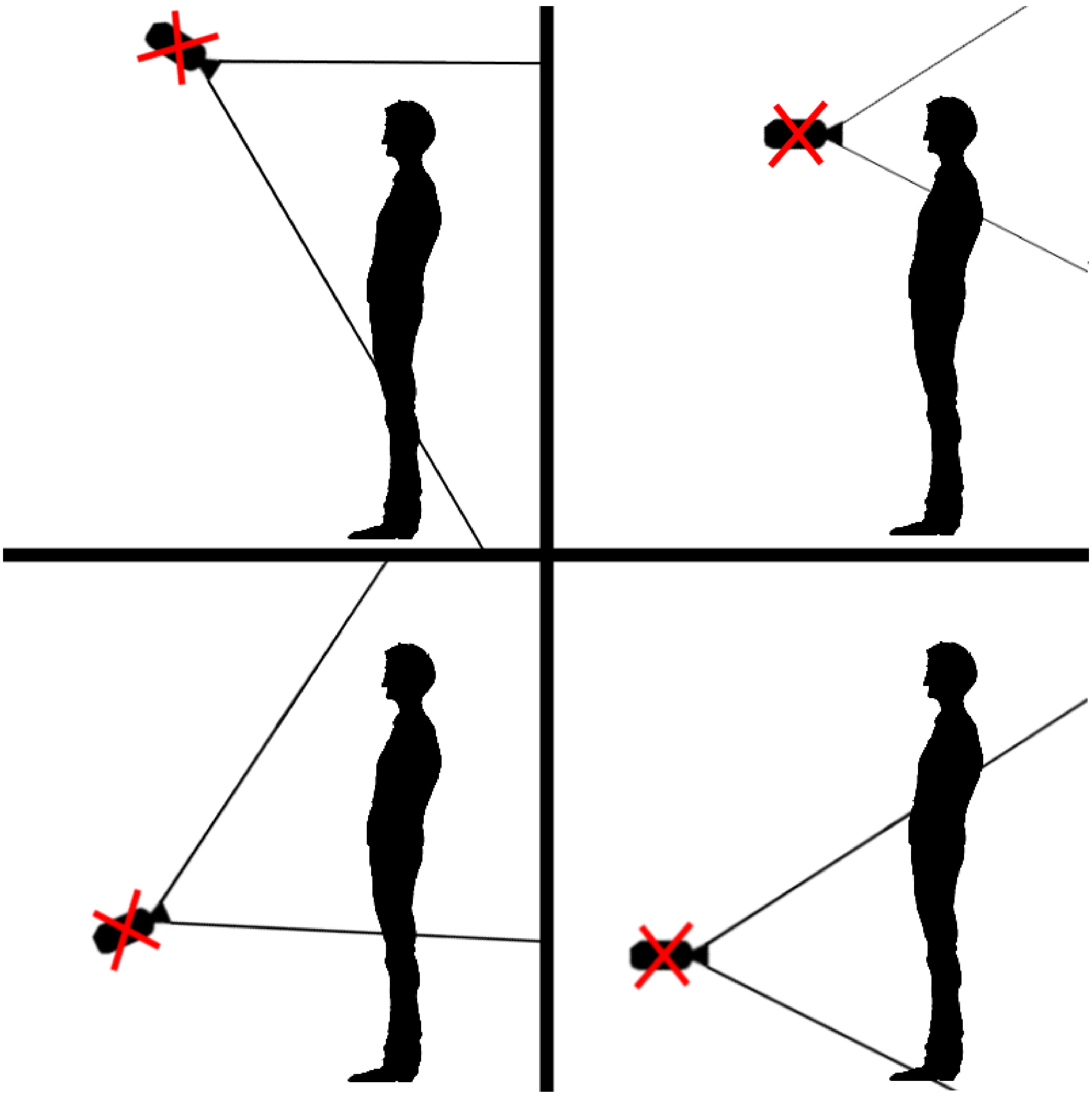

Figure 8: Wrong camera positions

4.4.5 Human body redetection

Like any other detector in Face Engine SDK, human detector also implements redetection model. The user can make full detection only in a first frame and then redetect the same human in the next "n" frames thereby boosting performance of the whole image processing loop.

User can use *redetectOne()* method if only a single human detection is required, for more complex use cases one should use *redetect()* which can redetect humans from multiple images.

Note: Detector give an opportunity to detect human body *keypoints* in an image.

4.4.6 Human Keypoints

The image below shows the keypoints detected for a human body.

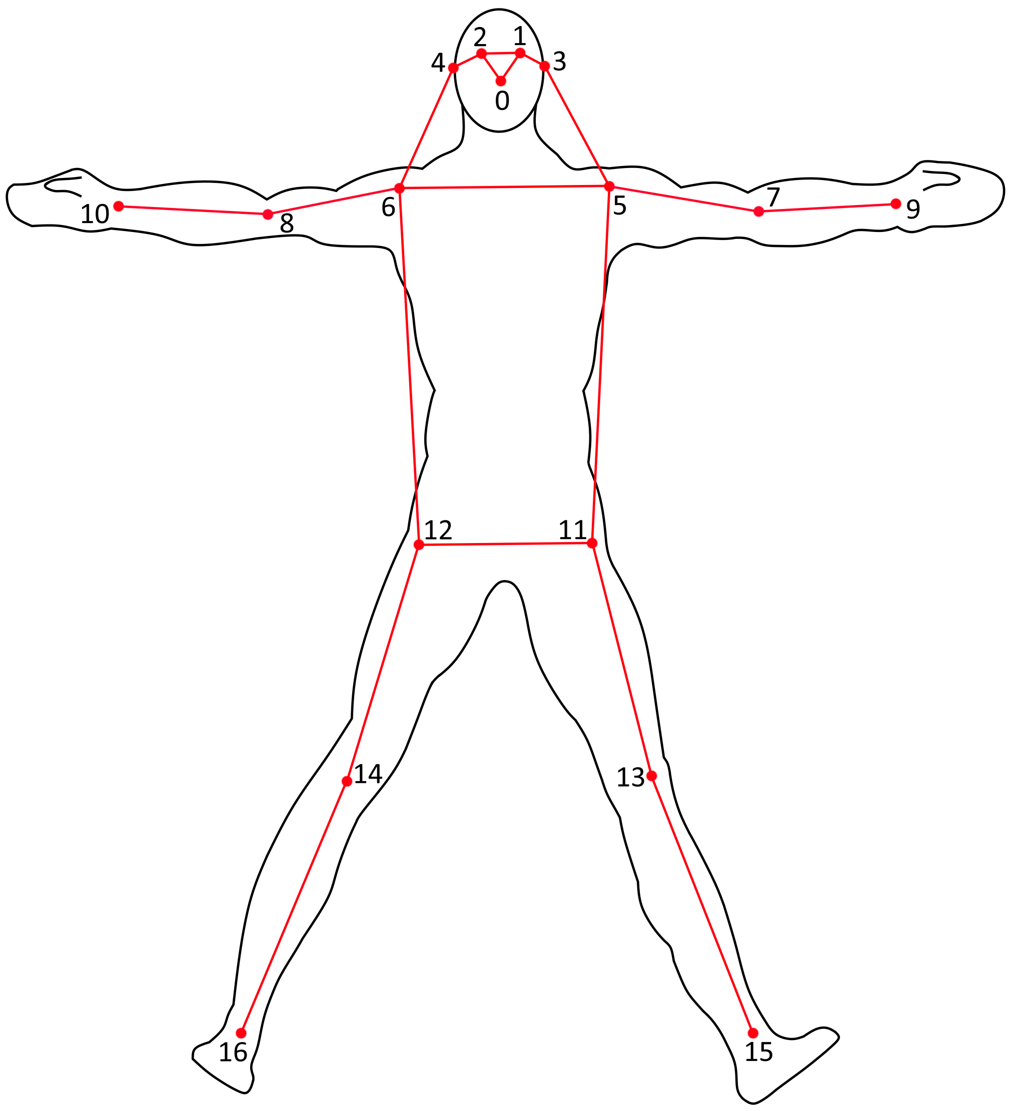

Figure 9: 17-points of human body

Point	Body Part	Point	Body Part
0	Nose	9	Left Wrist
1	Left Eye	10	Right Wrist

Point	Body Part	Point	Body Part
2	Right Eye	11	Left Hip
3	Left Ear	12	Right Hip
4	Right Ear	13	Left Knee
5	Left Shoulder	14	Right Knee
6	Right Shoulder	15	Left Ankle
7	Left Elbow	16	Right Ankle
8	Right Elbow		

4.4.7 Detection

To detect *Human Keypoints* call *detect()* using `fsdk::HumanDetectionType::DCT_BOX | fsdk::HumanDetectionType::DCT_POINTS` argument.

Note: Default is `fsdk::HumanDetectionType::DCT_BOX`.

4.4.8 Main Results of Each Detection

The main result of each detection is an array. Each array element consists of a point (fsdk:: Point2f) and a score. If the score value is less than the threshold, then the value of "x" and "y" coordinates will be equal to 0.

Note: see ConfigurationGuide.pdf ("HumanDetector settings" section) for more information about thresholds and configuration parameters.

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 Face Attribute Estimation

Note. The estimator is trained to work with warped images (see Chapter "Image warping" for details).

The Attribute estimator determines face attributes. Currently, the following attributes are available:

- Age: determines person's age;
- Gender: determinse person's gender;
- Ethnicity: determines ethnicity of a person.

Before using attribute estimator, user is free to decide whether to estimate or not some specific attributes listed above through *IAttributeEstimator::EstimationRequests* structure, which later get passed in main *estimate()* method. Estimator overrides *AttributeEstimationResults* output structure, which consists of optional fields describing results of user requested attributes.

- Age is reported in years:
  - For cooperative (see "Appendix B. Glossary") conditions: average error depends on person age, see the table below for additional details. Estimation precision is 2.3
- For gender estimation 1 means male, 0 means female.
  - Estimation precision in cooperative mode is 99.81% with the threshold 0.5;
  - Estimation precision in non-cooperative mode is 92.5%.
- Ethnicity estimation returns 4 float normalized values, each value describes probability of person's ethnicity.
  - Ethnicity Estimation precision in non-cooperative mode is 90.7%.
  - There are 4 types of races the estimator is currently able to distinguish: African American, Indian, Asian, Caucasian.
  - Estimates person's race depending on his/her appearance on a given image;
  - Outputs EthnicityEstimation structure with aforementioned data.

EthnicityEstimation displays races in scores that are presented as normalized float values in the range of [0..1]. The sum of scores always equals to 1. Each score stands for the probability of corresponding race.

Table 3: "Average age estimation error per age group for cooperative conditions"

Age (years)	Average error (years)
0-3	±3.3
4-7	±2.97
8-12	±3.06
13-17	±4.05
17-20	±3.89
20-25	±1.89
25-30	±1.88
30-35	±2.42
35-40	±2.65
40-45	±2.78
45-50	±2.88
50-55	±2.85
55-60	±2.86
60-65	±3.24
65-70	±3.85
70-75	±4.38
75-80	±6.79

Note In earlier releases of Luna SDK Attribute estimator worked poorly in non-cooperative mode (only 56% gender estimation precision), and did not estimate child's age. Having solved these problems average estimation error per age group got a bit higher due to extended network functionality.

5.3 Color/Monochrome Estimation

This estimator detects if an input image is grayscale or color. It implements *estimate()* function that accepts source *image* and outputs a Boolean, indicating if the image is grayscale (true) or not (false).

5.4 Image quality estimation

Note. The estimator is trained to work with warped images (see Chapter "Image warping" for details).

The general rule of thumb for quality estimation:

1. Detect a face, see if detection confidence is high enough. If not, reject the detection;
2. Produce a warped face image (see chapter "Descriptor processing facility") using a face detection and its landmarks;
3. Estimate visual quality using the estimator, finally reject low-quality images.

While the scheme above might seem a bit complicated, it is the most efficient performance-wise, since possible rejections on each step reduce workload for the next step.

At the moment estimator exposes two interface functions to predict image quality:

- **virtual Result estimate(const Image& warp, Quality& quality);**
- **virtual Result estimate(const Image& warp, SubjectiveQuality& quality);**

Each one of this functions use its own CNN internally and return slightly different quality criteria.

The first CNN is trained specifically on pre-warped human face images and will produce lower score factors if one of the following conditions are satisfied:

- Image is blurred;
- Image is under-exposured (i.e., too dark);
- Image is over-exposured (i.e., too light);
- Image color variation is low (i.e., image is monochrome or close to monochrome).

Each one of this score factors is defined in [0..1] range, where higher value corresponds to better image quality and vice versa.

Recommended thresholds for image quality of the first interface function are given below:

"saturationThreshold": 0.0; "blurThreshold": 0.93; "lightThreshold": 0.9; "darkThreshold": 0.9;

The second interface function output will produce lower factor if:

- The image is blurred;
- The image is underexposed (i.e., too dark);
- The image is overexposed (i.e., too light);
- The face in the image is illuminated unevenly (there is a great difference between light and dark regions);
- Image contains flares on face (too specular).

The estimator determines the quality of the image based on each of the aforementioned parameters. For each parameter, the estimator function returns two values: the quality factor and the resulting verdict.

As with the first estimator function the second one will also return the quality factors in the range [0..1], where 0 corresponds to low image quality and 1 to high image quality. E. g., the estimator returns low quality factor for the Blur parameter, if the image is too blurry.

The resulting verdict is a quality output based on the estimated parameter. E. g., if the image is too blurry, the estimator returns "isBlurred = true".

The threshold can be specified for each of the estimated parameters. The resulting verdict and the quality factor are linked through this threshold. If the received quality factor is lower than the threshold, the image quality is low and the estimator returns "true". E. g., if the image blur quality factor is higher than the threshold, the resulting verdict is "false".

If the estimated value for any of the parameters is lower than the corresponding threshold, the image is considered of bad quality. If resulting verdicts for all the parameters are set to "False" the quality of the image is considered good.

Examples are presented in the images below. Good quality images are shown on the right.

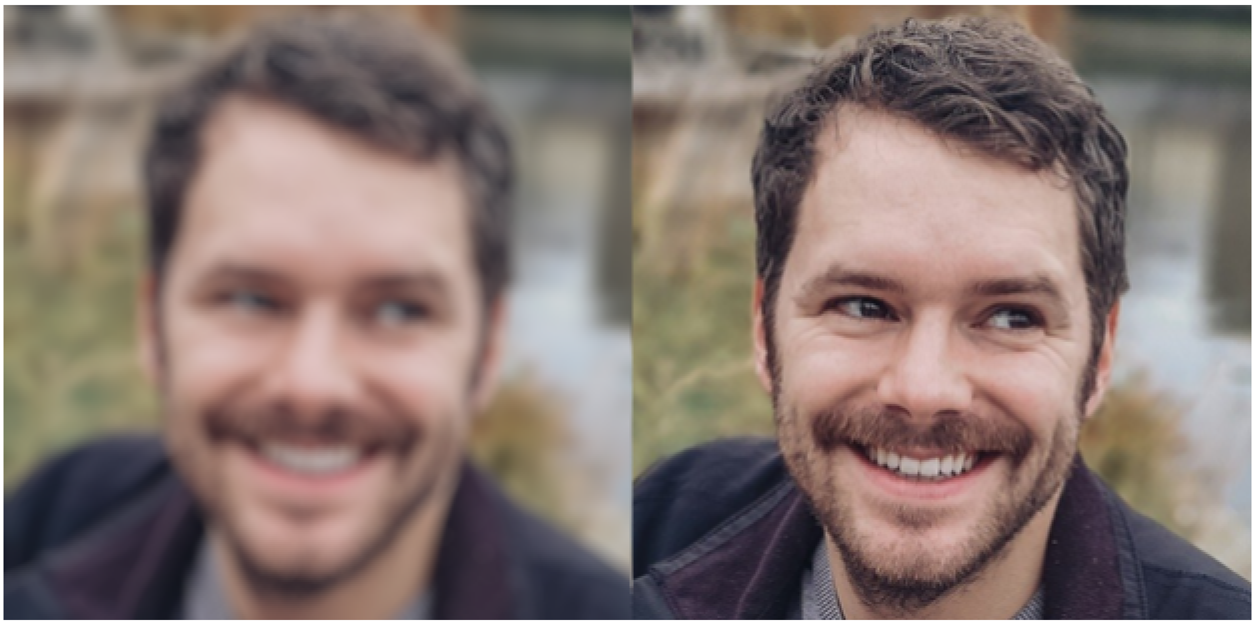

Figure 10: Blurred image (left), not blurred image (right)

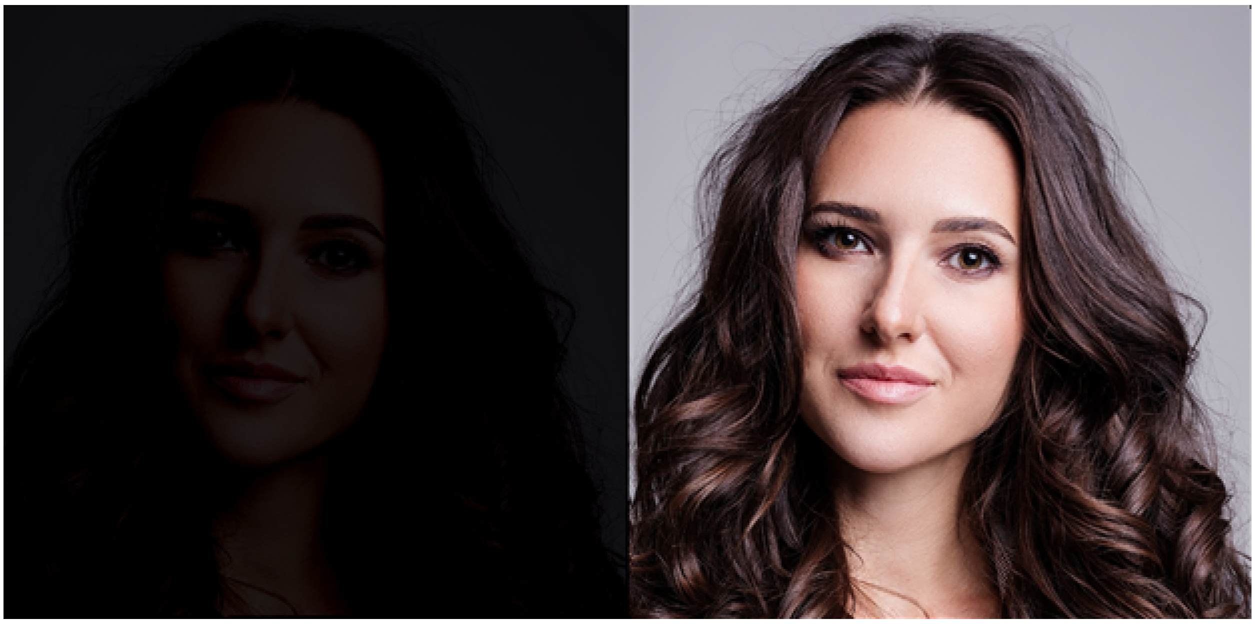

Figure 11: Dark image (left), good quality image (right)

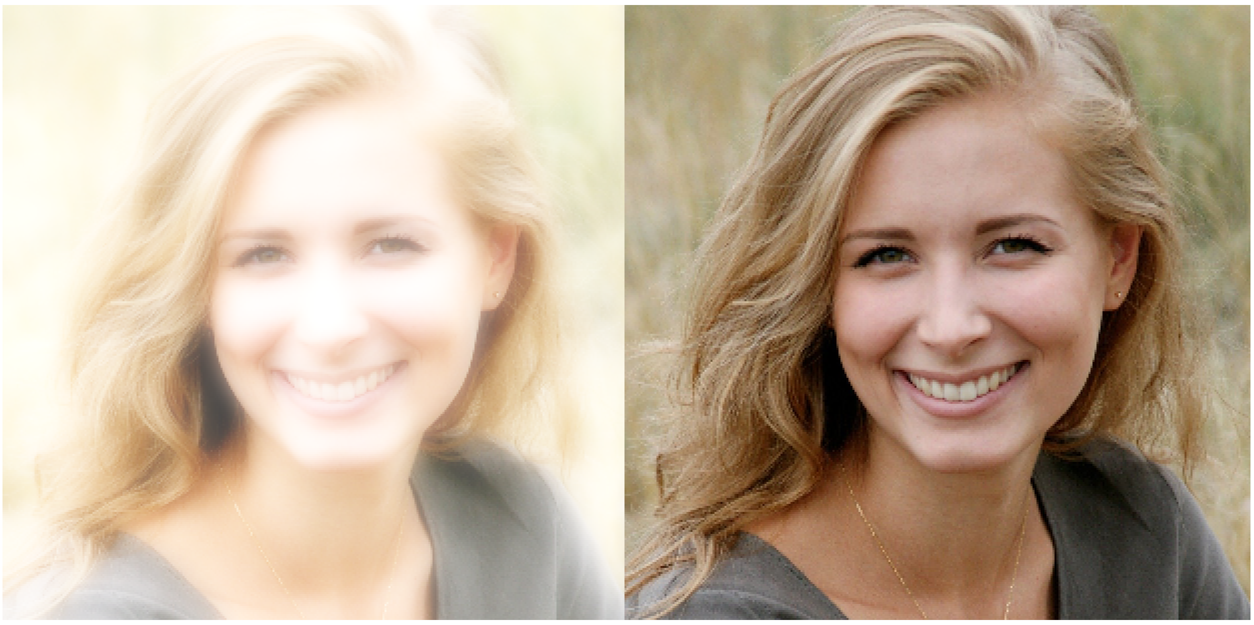

Figure 12: Light image (left), good quality image (right)

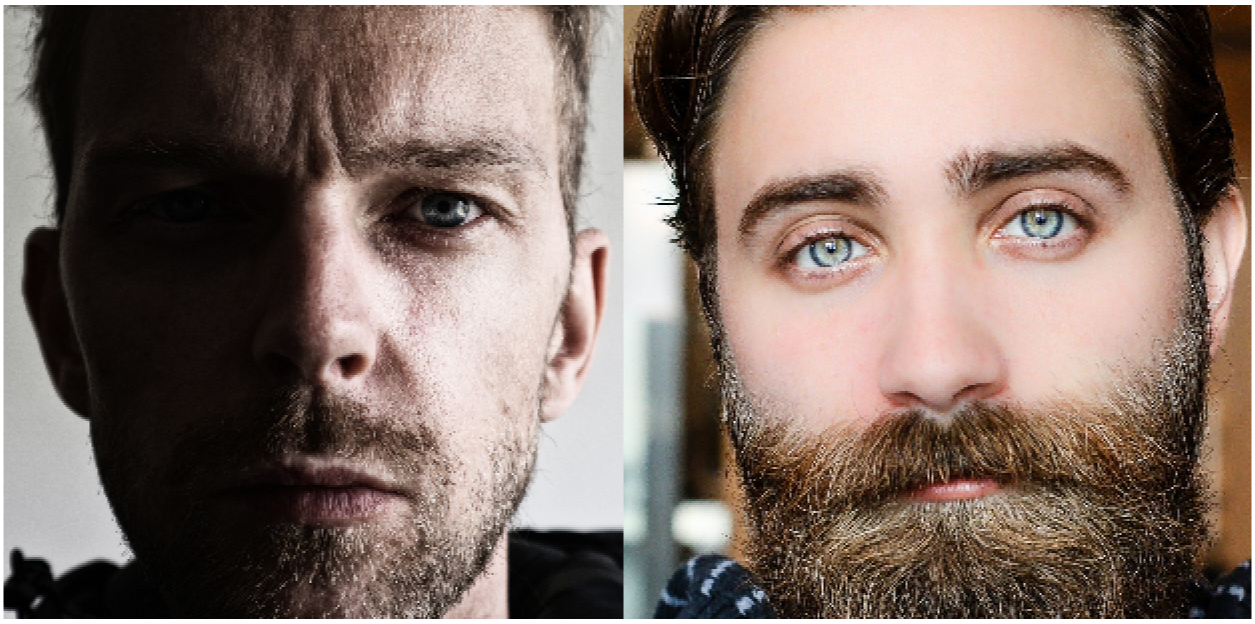

Figure 13: Image with uneven illumination (left), image with even illumination (right)

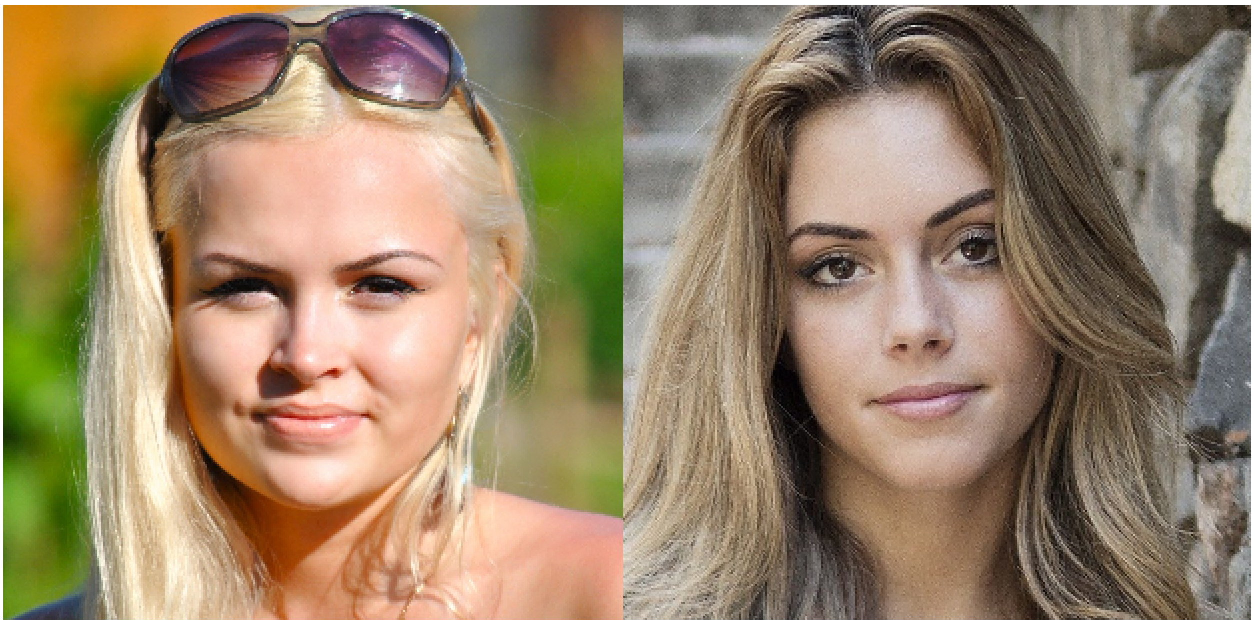

Figure 14: Image with specularity - image contains flares on face (left), good quality image (right)

The quality factor is a value in the range [0..1] where 0 corresponds to low quality and 1 to high quality.

Note. Illumination uniformity corresponds to the face illumination in the image. The lower the difference between light and dark zones of the face, the higher the estimated value. When the illumination is evenly distributed throughout the face, the value is close to "1".

Note. Specularity is a face possibility to reflect light. The higher the estimated value, the lower the specularity and the better the image quality. If the estimated value is low, there are bright glares on the face.

Table 4: Image quality parameters and their thresholds

Threshold	Estimated property	Recomended range	Default value
blurThreshold	Blur	[0.57..0.65]	0.61
darknessThreshold	Darkness	[0.45..0.52]	0.50
lightThreshold	Light	[0.44..0.61]	0.57
illuminationThreshold	Illumination uniformity	[0..0.3]	0.1
specularityThreshold	Specularity	[0..0.3]	0.1

The most important parameters for face recognition are "blurThreshold", "darknessThreshold" and "lightThreshold", so you should select them carefully.

You can select images of better visual quality by setting higher values of the "illuminationThreshold" and "specularityThreshold". Face recognition is not greatly affected by uneven illumination or glares.

5.5 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 xright < xleft.

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.

5.6 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 5: "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.

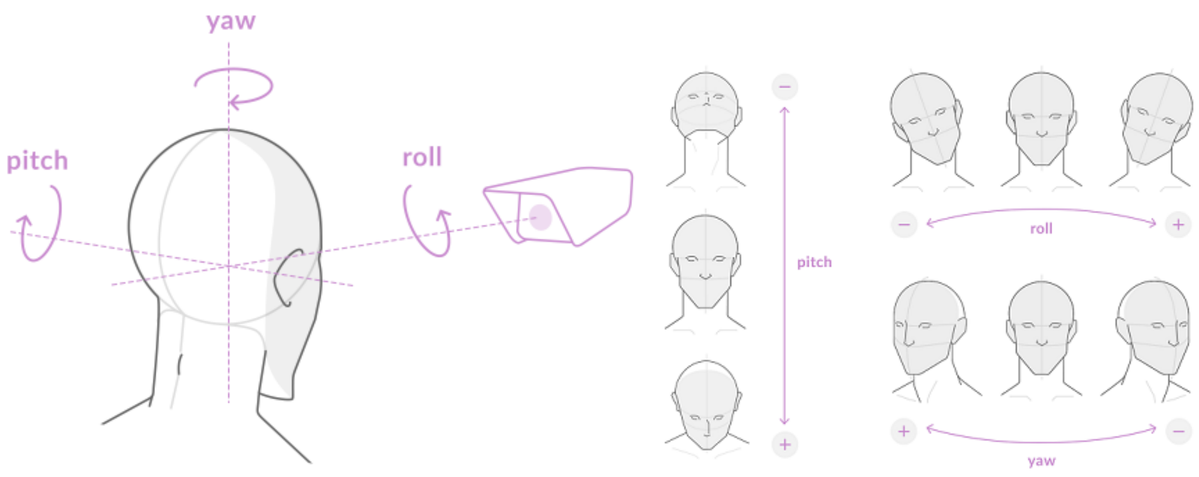

Figure 15: Head pose

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.

5.7 Gaze Estimation

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

Estimation characteristics:

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

Note. Roll angle is not estimated, prediction precision decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table below.

Table 6: "Gaze prediction precision"

	Range	-25°...+25°	-25° ... -45 ° or 25 ° ... +45°
Average prediction error (per axis)	Yaw	±2.7°	±4.6°
Average prediction error (per axis)	Pitch	±3.0°	±4.8°

Zero position corresponds to a gaze direction orthogonally to face plane, with the axis of symmetry parallel to the vertical camera axis. See figure Head pose estimation for a reference.

5.8 Smile Estimation

This estimator is designed to determine smile/mouth/occlusion probability using warped face image.

Smile estimation structure consists of:

- Smile score;
- Mouth score;
- Occlusion score.

Sum of scores always equals 1. Each score means probability of corresponding state. Smile score prevails in cases where smile was successfully detected. If there is any object on photo that hides mouth occlusion score prevails. Mouth score prevails in cases where neither a smile nor an occlusion was detected.

5.9 Mouth Estimation

This estimator is designed to predict person's mouth state. It returns the following bool flags:

- isOpened;
- isOccluded;
- isSmiling.

Each of this 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.

Similar to smile estimator mouth estimator provides score probabilities for mouth states in case user need more detailed information:

- Mouth opened score;

- Smile score;
- Occlusion score.

Note: This estimator is trained to work with warped images (see Chapter "Image warping" for details).

5.10 Emotions Estimation

Note: The estimator is trained to work with warped images (see Chapter "Image warping" for details).

This estimator aims to determine whether a face depicted on an image expresses the following emotions:

- Anger
- Disgust
- Fear
- Happiness
- Surprise
- Sadness
- Neutrality

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

The estimator (see IEmotionsEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts warped source image (see Chapter "Image warping"). Warped image is received from the warper (see IWarper::warp());
- Estimates emotions expressed by the person on a given image;
- Outputs EmotionsEstimation structure with aforementioned data.

EmotionsEstimation presents emotions as normalized float values in the range of [0..1] where 0 is lack of a specific emotion and 1 is the maximum intensity of an emotion.

5.11 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.

5.12 Glasses Estimation

Glasses estimator is designed to determine whether a person is currently wearing any glasses or not. There are 3 types of states estimator is currently able to estimate:

- NoGlasses state determines whether a person is wearing any glasses at all;
- EyeGlasses state determines whether a person is wearing eyeglasses;
- SunGlasses state determines whether a person is wearing sunglasses.

Note. Source input image must be warped in order for estimator to work properly (see Chapter "Image warping"). Quality of estimation depends on threshold values located in faceengine configuration file (faceengine.conf) in GlassesEstimator::Settings section. By default, these threshold values are set to optimal.

The table below contains true positive rates corresponding to selected false positive rates.

Table 7: "Glasses estimator TPR/FPR rates"

State	TPR	FPR
NoGlasses	0.997	0.00234
EyeGlasses	0.9768	0.000783
SunGlasses	0.9712	0.000383

5.13 Overlap Estimation

This estimator tells whether the face is overlapped by any object. It returns a structure with 2 fields. One is the value of overlapping in the range [0..1] where 0 is not overlapped and 1.0 is overlapped, the second is a Boolean answer. A Boolean answer depends on the threshold listed below. If the value is greater than the threshold, the answer returns true, else false.

The estimator (see IOverlapEstimator 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");
- Estimates whether the face is overlapped by any object on input image;
- Outputs structure with value of overlapping and Boolean answer.

5.14 Child Estimation

This estimator tells whether the person is child or not. Child is a person who younger than 18 years old. It returns a structure with 2 fields. One is the score in the range from 0.0 (is adult) to 1.0 (maximum, is child), the second is a boolean answer. Boolean answer depends on the threshold in config (faceengine.conf). If the value is more than the threshold, the answer is true (person is child), else - false (person is adult).

The estimator (see IChildEstimator in IEstimator.h):

- Implements the *estimate()* function accepts warped source image (see chapter "Image warping"). Warped image is received from the warper (see IWarper::warp());
- Estimates whether the person is child or not on input warped image;
- Outputs ChildEstimation structure. Structure consists of score of and boolean answer.

5.15 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.

5.16 HeadAndShouldersLiveness Estimation

This estimator tells whether the person's face is real or fake (photo, printed image) and confirms presence of a person's body in the frame. Face should be in the center of the frame and the distance between the face and the frame borders should be three times greater than space that face takes up in the frame. Both person's face and chest have to be in the frame. Camera should be placed at the waist level and directed from bottom to top. The estimator check for borders of a mobile device to detect fraud. So there should not be any rectangular areas within the frame (windows, pictures, etc.).

The estimator (see IHeadAndShouldersLiveness in IEstimator.h):

- Implements the *estimateHeadLiveness()* 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 whether it is a real person or not. Outputs float normalized score in range [0..1], 1 - is real person, 0 - is fake. Implements the *estimateShouldersLiveness()* 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 whether real person or not. Outputs float score normalized in range [0..1], 1 - is real person, 0 - is fake.

5.17 LivenessFlyingFaces Estimation

This estimator tells whether the person’s face is real or fake (photo, printed image).

The estimator (see ILivenessFlyingFacesEstimator in IEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Face` with valid image in R8G8B8 format and detection structure of corresponding source image (see section “Detection structure” in chapter “Detection facility”). This method estimates whether a real person or not. Output estimation with float score is normalized in range [0..1], where 1 - is real person, 0 - is fake.
- Implements the *estimate()* function that needs the span of `fsdk::Faces` with valid source images in R8G8B8 formats and fsdk::Detection structures of corresponding source images (see section “Detection structure” in chapter “Detection facility”). Each element of span of `fsdk::Face` must be with valid image and detection. This method estimates whether different persons are real or not. Corresponding estimation output with float scores which are normalized in range [0..1], where 1 - is real person, 0 - is fake.

5.18 LivenessRGBM Estimation

This estimator tells whether the person’s face is real or fake (photo, printed image).

The estimator (see ILivenessRGBMEstimator in IEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Face` with valid image in R8G8B8 format, detection structure of corresponding source image (see section “Detection structure” in chapter “Detection facility”) and `fsdk::Image` with accumulated background. This method estimates whether a real person or not. Output estimation structure contains the float score and boolean result. The float score normalized in range [0..1], where 1 - is real person, 0 - is fake. The boolean result has value true for real person and false otherwise.
- Implements the *update()* function that needs the `fsdk::Image` with current frame , number of that image and previously accumulated background. The accumulated background will be overwritten by this call.

5.19 Medical Mask Estimation

This estimator aims to detect a medical face mask on the face in the source image. It can return the next results:

- A medical mask is on the face (see MedicalMask::Mask field in the MedicalMask enum);
- There is no medical mask on the face (see MedicalMask::NoMask field in the MedicalMask enum);
- The face is occluded with something (see MedicalMask::OccludedFace field in the MedicalMask enum);

The estimator (see IMedicalMaskEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts source warped image in R8G8B8 format and MedicalMaskEstimation structure to return results of estimation;
- Implements the *estimate()* function that accepts source image in R8G8B8 format, face detection to estimate and MedicalMaskEstimation structure to return results of estimation;
- Implements the *estimate()* function that accepts fsdk::Span of the source warped images in R8G8B8 format and fsdk::Span of the MedicalMaskEstimation structures to return results of estimation;
- Implements the *estimate()* function that accepts fsdk::Span of the source images in R8G8B8 format, fsdk::Span of face detections and fsdk::Span of the MedicalMaskEstimation structures to return results of the estimation.

The estimator was implemented for two use-cases:

1. When the user already has warped images. For example, when the medical mask estimation is performed right before (or after) the face recognition;
2. When the user has face detections only.

Note: Calling the *estimate()* method with warped image and the *estimate()* method with image and detection for the same image and the same face could lead to different results.

5.19.1 MedicalMaskEstimator thresholds

The estimator returns several scores - one for each possible result. The final result calculated based on that scores and thresholds. If some score is above the corresponding threshold, that result is estimated as final. The default values for all thresholds are taken from the configuration file. See Configuration guide for details.

5.19.2 MedicalMask enumeration

The MedicalMask enumeration contains all possible results of the MedicalMask estimation:

```
    enum class MedicalMask {
        Mask = 0,                //!< medical mask is on the face
```

```
            NoMask,                     //!< no medical mask on the face
            OccludedFace                //!< face is occluded by something
        };
```

5.19.3 MedicalMaskEstimation structure

The MedicalMaskEstimation structure contains results of the estimation:

```
        struct MedicalMaskEstimation {
            MedicalMask result;          //!< estimation result (@see MedicalMask
               enum)
            // scores
            float maskScore;            //!< medical mask is on the face score
            float noMaskScore;          //!< no medical mask on the face score
            float occludedFaceScore;    //!< face is occluded by something score
        };
```

There are two groups of the fields:

1. The first group contains only the result enum:

```
            MedicalMask result;
```

Result enum field MedicalMaskEstimation contain the target results of the estimation.

2. The second group contains scores:

```
            float maskScore;            //!< medical mask is on the face score
            float noMaskScore;          //!< no medical mask on the face score
            float occludedFaceScore;    //!< face is occluded by something score
```

The scores group contains the estimation scores for each possible result of the estimation. All scores are defined in [0,1] range. They can be useful for users who want to change the default thresholds for this estimator. If the default thresholds are used, the group with scores could be just ignored in the user code.

5.20 Credibility Check Estimation

This estimator estimates reliability of a person.

The estimator (see ICredibilityCheckEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts warped image in R8B8G8 format and `fsdk::CredibilityCheckEstimation` structure.

- Implements the *estimate()* function that accepts span of warped images in R8B8G8 format and span of `fsdk::CredibilityCheckEstimation` structures.

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

Table 8: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-20…20]
yaw	[-20…20]
roll	[-20…20]

Table 9: "Requirements for `fsdk::SubjectiveQuality`"

Attribute	Minimum value
blur	0.61
light	0.57

Table 10: "Requirements for `fsdk::AttributeEstimationResult`"

Attribute	Minimum value
age	18

Table 11: "Requirements for `fsdk::OverlapEstimation`"

Attribute	State
overlapped	false

Table 12: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	100

Note. Detection size is detection width.

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

5.21 LivenessIREstimation Estimation

This estimator tells whether the person's face is real or fake (photo, printed image).

The estimator (see ILivenessIREstimator in IEstimator.h):

- Implements the *estimate()* function that accepts warped image in R8G8B8, R16 or IR_X8X8X8 format and `fsdk::IREstimation` structure.
- Implements the *estimate()* function that accepts span of warped images in R8G8B8, R16 or IR_X8X8X8 format and span of `fsdk::IREstimation` structures.

Note. The estimator for `Verme` camera is trained to work with face images that meet the following requirements:

Table 13: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-20…20]
yaw	[-20…20]
roll	[-20…20]

Table 14: "Requirements for `fsdk::SubjectiveQuality`"

Attribute	Minimum value
blur	0.4
light	0.1

Attribute	Minimum value
darkness	0.3
illumination	0.6
specularity	0.5

Table 15: "Requirements for `fsdk::EyesEstimation`"

Attribute	State
left eye	`fsdk::EyesEstimation::EyeAttributes::State::Open`
right eye	`fsdk::EyesEstimation::EyeAttributes::State::Open`

Table 16: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	160

Note. Detection size is detection width.

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

5.22 Facial Hair Estimation

This estimator aims to detect a facial hair type on the face in the source image. It can return the next results:

- There is no hair on the face (see FacialHair::NoHair field in the FacialHair enum);
- There is stubble on the face (see FacialHair::Stubble field in the FacialHair enum);
- There is mustache on the face (see FacialHair::Mustache field in the FacialHair enum);
- There is beard on the face (see FacialHair::Beard field in the FacialHair enum);

The estimator (see IFacialHairEstimator in IFacialHairEstimator.h):

- Implements the *estimate()* function that accepts source warped image in R8G8B8 format and FacialHairEstimation structure to return results of estimation;

- Implements the *estimate()* function that accepts fsdk::Span of the source warped images in R8G8B8 format and fsdk::Span of the FacialHairEstimation structures to return results of estimation.

5.22.1 FacialHair enumeration

The FacialHair enumeration contains all possible results of the FacialHair estimation:

```
        enum class FacialHair {
            NoHair = 0,                 //!< no hair on the face
            Stubble,                    //!< stubble on the face
            Mustache,                   //!< mustache on the face
            Beard                       //!< beard on the face
        };
```

5.22.2 FacialHairEstimation structure

The FacialHairEstimation structure contains results of the estimation:

```
        struct FacialHairEstimation {
            FacialHair result;          //!< estimation result (@see FacialHair
                enum)
            // scores
            float noHairScore;          //!< no hair on the face score
            float stubbleScore;         //!< stubble on the face score
            float mustacheScore;        //!< mustache on the face score
            float beardScore;           //!< beard on the face score
        };
```

There are two groups of the fields:

1. The first group contains only the result enum:

```
            FacialHair result;          //!< estimation result (@see FacialHair
                enum)
```

Result enum field FacialHairEstimation contain the target results of the estimation.

2. The second group contains scores:

```
            float noHairScore;          //!< no hair on the face score
            float stubbleScore;         //!< stubble on the face score
            float mustacheScore;        //!< mustache on the face score
            float beardScore;             //!< beard on the face score
```

The scores group contains the estimation scores for each possible result of the estimation. All scores are defined in [0,1] range. Sum of scores always equals 1.

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

Table 17: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-40…40]
yaw	[-40…40]
roll	[-40…40]

Table 18: "Requirements for `fsdk::MedicalMaskEstimation`"

Attribute	State
result	fsdk::MedicalMask::NoMask

Table 19: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	40

Note. Detection size is detection width.

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

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.

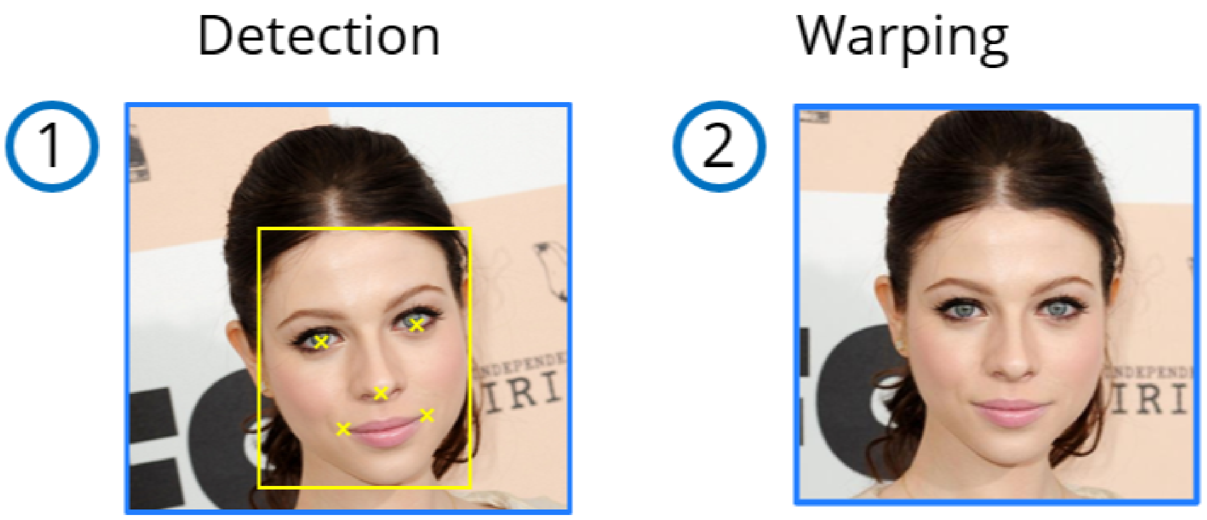

Figure 16: Face warping

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.

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.

There are two different tasks solved using descriptors: person identification and person reidentification.

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).

Note: Additionally you can extract face features (gender, age, emotions, etc) or image attributes (light, dark, blur, specularity, illumination, etc.).

7.1.2 Person Reidentification Task

Note! This functionality is experimental.

The person reidentification enables you to detect a person who appears on different cameras. For example, it is used when you need to track a human, who appears on different supermarket cameras. Reidentification can be used for:

- building of human traffic warm maps;
- analysing of visitors movement;
- tracking of visitors;
- etc.

For reidentification purposes, we use so-called human descriptors. The extraction of the human descriptor is performed using the detected area with a person's body on an image or video frame. The descriptor is a unique data set formed based on a person's appearance. Descriptors extracted for the same person in different clothes will be significantly different.

> The face descriptor and the human descriptor are almost the same from the technical point of view, but they solve fundamentally different tasks.

The process of reidentifications consists of the following stages:

- human detection in an image;
- warping of human detection – centering and cropping of the human body;
- descriptor extraction;
- comparing of extracted descriptors (matching).

> The human descriptor does not support the *descriptor score* at all. The returned value of the descriptor score is always equal to 1.0.

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 may be specified in the configuration file (see section “Configuration data” in chapter “Core facility”).

7.2.1.1 Face descriptor

Currently next versions are available: 46, 52, 54, 56, 57 and 58. Descriptors have **backend** and **mobilenet** implementations. Versions 57 and 58 supports only **backend** implementation. Backend versions more precise, but mobilenet faster and have smaller model files (See Appendix A). Versions 57, 56, 54 and 52 more precise then 46, but version 46 works very fast on GPU. Version 58 is the most precise.
See Appendix A.1 and A.2 for details about performance and precision of different descriptor versions.

Note Version 46 and 52 are deprecated since LUNA SDK release v.4.1.0. Use 54, 56 or 57 versions in new projects.

7.2.1.2 Human descriptor

Currently, only one version of human descriptor is available: 101.

To create a human descriptor, human batch, human descriptor extractor, human descriptor matcher you must pass the minimum human descriptor version (DV_MIN_HUMAN_DESCRIPTOR_VERSION) which equals 101.

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.

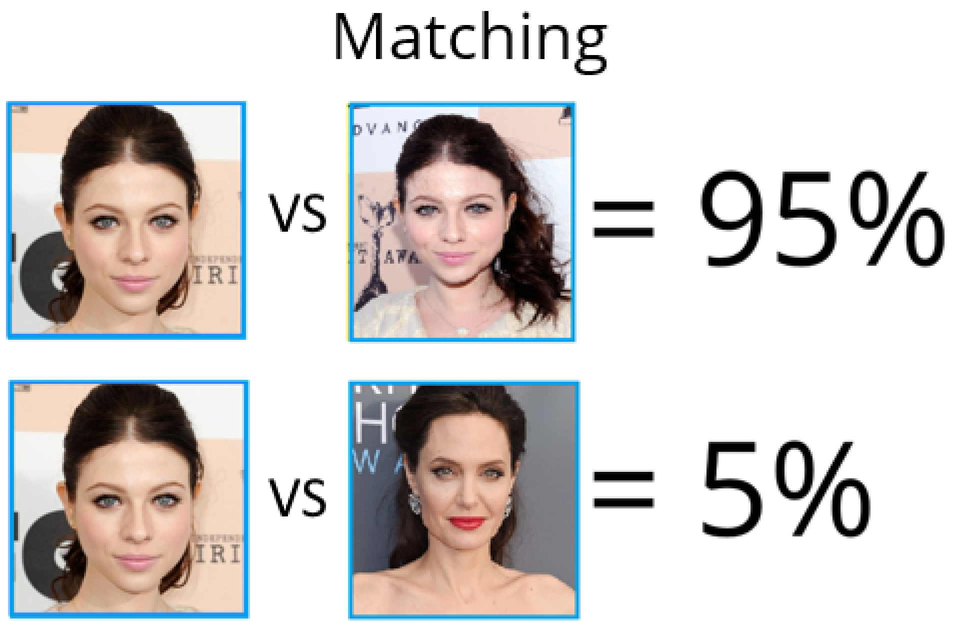

Figure 17: Matching

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.

7.6 Descriptor Indexing

7.6.1 Using HNSW

In order to accelerate the descriptor matching process, a **special index** may be created for a descriptor batch. With the index, matching becomes a two-stage process:

- First, you need to build indexed data structure - index - using *IIndexBuilder*. This is quite slow process so it is not supposed to be done frequently. You build it by appending *IDescriptor* objects or *IDescriptorBatch* objects and finally using build method - *IIndexBuilder::buildIndex*;
- Once you have index, you can use it to search nearest neighbors for passed descriptor very fast.

There are two types of indexes: *IDenseIndex* and *IDynamicIndex*. The interface difference is very simple: dense index is read only and dynamic index is editable: you can append or remove descriptors.

You can only build a dynamic index. So how can you get a dense index? The answer is through deserialization. Imagine you have several processes that might need to search in index. One option is for every one of them to build index separately, but as mentioned before building of index is very slow and you probably don't want to do it more than needed. So the second option is to build it once and serialize it to file. This is where the dense and dynamic difference arises: formats used to store these two types of index are different. From the user's point of view, the difference is that dense index loads faster, but it is read only. Once loaded, there are no performance difference in terms of searching on these two types of indexes.

To serialize index use *IDynamicIndex::saveToDenseIndex* or *IDynamicIndex::saveToDynamicIndex* methods. To deserialized use *IFaceEngine::loadDenseIndex* or *IFaceEngine::loadDynamicIndex*.

Note: Index files are not cross-platform. If you serialize index on some platform, it's only usable on that exact platform. Not only the operating system breaks compatibility, but also different architecture of CPU might break it.

Note: HNSW index isn't supported on embedded and 32-bit desktop platforms.

8 Licensing

8.1 Server platforms

FaceEngine supports per-features node-locked licensing for every supported platform. This means that by activating and deactivating the licensed features a final customer version of FaceEngine might be customized. For that reason, not all algorithms and modules described in this book might be available to you. Each SDK instance should be activated on every device. License, which was activated for one device could not be used on some other device. Interface for License objects ILicense (see file ILicense.h) gives you possibility to:

- check if license is already activated;
- save license for next usage to some file;
- load license from file;
- check if some feature of FaceEngine is available for this license;
- check the expiration date for each feature in this license.

Typical usage scenario:

- Create a IFaceEngine object (see file FaceEngine.h);
- Get license pointer through fsdk::IFaceEngine::getLicense.
- Make activation for that license object through fsdk::activateLicense. This method requires full or relative path to the license.conf file.

NOTE: Descriptor license feature allows creating no more than 100000 descriptors on server platforms.

NOTE: If the feature is not available for the current license or the feature has expired, an attempt to use the corresponding functionality will result in an error.

8.1.1 License features

To work with license features in code the ***LicenseFeature*** enum should be used (see file ILicense.h). Some features are not available for some platforms.

Full list of features and according list of estimators:

- Detection - allows to create IDetector instance to find person face on a frame;
- BestShot - allows to create IAGSEstimator, IBestShotQualityEstimator, IHeadPoseEstimator, IQualityEstimator instances;
- Attributes - allows to create IAttributeEstimator, IChildEstimator, ICredibilityCheckEstimator instances;
- Emotions - allows to create IEmotionsEstimator, IMouthEstimator, ISmileEstimator instances;
- FaceFeatures - allows to create IEyeEstimator, IGazeEstimator, IGlassesEstimator, IMedicalMaskEstimator, IOverlapEstimator instances;

- Liveness - allows to create ILivenessDepthEstimator, ILivenessFlowEstimator, IHeadPoseAndShouldersLiveness ILivenessIREstimator, ILivenessFlyingFacesEstimator, ILivenessRGBMEstimator instances;
- Descriptor - allows to create IDescriptor, IDescriptorBatch, IDescriptorExtractor, IDescriptorMatcher instances to extract face features of a concrete person and work with them;
- DescriptorIndex - allows to create IIndexBuilder, IDenseIndex, IDynamicIndex instances;
- LivenessEngine - allows to create ILivenessEngine instance to check real person or not on video;
- TrackEngine - allows to create ITrackEngine instance to track person on video;
- HumanDetection - allows to create IHumanDetector instance for human bodies detection.
- MedicalMaskDetection - allows to create IMedicalMaskEstimator instance to detect a medical face mask on the face in the source image.

If the license for the selected feature is invalid, the factory instantiation method will return nullptr. For example, method: ***IFaceEngine::createAGSEstimator*** will return ***nullptr*** in case if LicenseFeature::BestShot in not available.

9 System Requirements

9.0.1 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.

9.1 Hardware requirements

9.1.1 Embedded installations

9.1.1.1 CPU requirements

Supported CPU architectures:

- ARMv7-A;
- ARMv8-A (ARM64).

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.

11 Appendix C. FAQ

Q: This document contains high-level descriptions and no code examples nor reference. Where can one find them?

A: The complete type and function reference are provided as an interactive web-based documentation; see the *doc/fsdk/index.html* inside the LUNA SDK package. The examples are located in the */examples* folder and "ExamplesGuide.pdf" is located in /doc folder of LUNA SDK package.

Q: Does FaceEngine support multicore / multiprocessor systems?

A: Yes, all internal algorithm implementations are multithreaded by design and take advantage of multi-core systems. The number of threads may be controlled via the configuration file; see configuration manual "ConfigurationGuide.pdf" or comments in the configuration file for details.

Q: What is the state of GPU support?

A: As of version 2.7 the GPU support is implemented for face detection and descriptor extraction algorithms. Starting from version 2.9 GPU implementations are considered stable.

Q: What speedup may be expected from GPUs?

A: Typically GPUs allow accelerating algorithms by the factor of 2-4 times depending on microprocessor architecture and input data.

Q: Are there any official bindings/wrappers for other languages (C#, Java)?

A: No, such bindings are not provided. FaceEngine officially implements C++ API only, bindings to other languages should be created by users themselves. There are tools to automate this process, like, e.g., SWIG.

Q: Does FaceEngine support DBMS systems?

A: No, FaceEngine implements just computer vision algorithms. Users should implement DBMS communication themselves using serialization methods described in section "Serializable object interface" of chapter "Core concepts" and section "Archive interface" of chapter "Core facility".

Q: What image formats does FaceEngine support?

A: FaceEngine does not implement image format encoding functions. If such functions are required, one should use a third-party library, e.g., FreeImage.

FaceEngine functions typically expect image data in the form of uncompressed unencoded pixel data (RGB color 24 bits per pixel or grayscale 8 bits per pixel).

FaceEngine implements convenience functions like RGB -> grayscale and RGB<-> BGR color conversions. The rationale of this design is explained in section "Image type" of chapter "Core concepts".