VisionLabs FaceEngine Handbook

Contents

Introduction

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

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

What this book does, however, is this:

- It describes ideas behind resource management and gives a clue why one or another decision was made. With this in mind, you are ready to write efficient code with FaceEngine;

- It breaks down full face analysis and recognition pipeline in parts and shows how one part affects all the others. This information will help you to adapt FaceEngine to your needs, which is somewhat more productive than blindly following tutorials;

- It details things that are important and omits things that are obvious, so you get information that matters most.

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

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

1 Core Concepts

1.1 Common Interfaces and Types

1.1.1 Reference Counted Interface

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

- to notify the object that it is required by a client via *retaining* a reference to it;

- to notify the object that it is no longer required by *releasing* a reference to it;

- to get actual reference counter value.

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

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

1.1.2 Automatic reference counting

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

Two ways of working with Ref are possible:

1.1.2.1 Referencing - without acquiring ownership of object lifetime

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

```
  */

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

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

1.1.2.2 Acquiring - own object lifetime

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

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

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

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

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

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

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

As a side note, be informed that *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.

> 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 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 Sensor type

SensorType determines which type of camera sensor is used to perform estimation. Currently two types of SensorType are available: `Visible`, `NIR`. The user can indicate the required type of sensor when creating an object by passing the appropriate parameter.

3.4 Data Paths

3.4.1 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.

3.4.2 Configuration Data

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

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

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

4 Detection facility

4.1 Overview

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

4.2 Detection structure

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

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

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

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).

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

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

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

4.3.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.
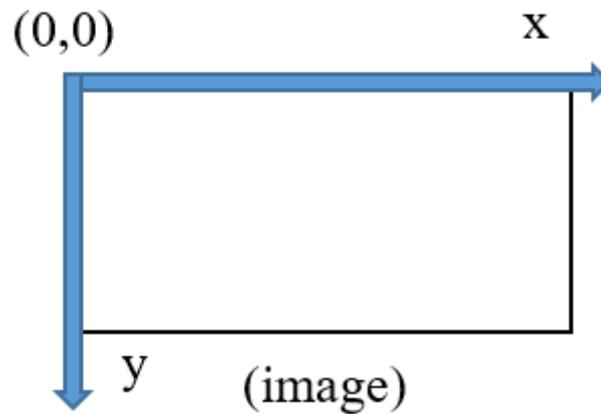
Figure 1: Source image coordinate system

4.3.2 Face detection

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

4.3.3 Redetect method

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

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

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

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

Detector works faster with larger value of `minFaceSize`.

4.3.4 Orientation Estimation

Name: OrientationEstimator

Algorithm description:

This estimator aims to detect an orientation of the input image. The next outputs are supported:

- The target image is normal oriented ;

- The target image is turned to the left by 90 deg;
- The target image is flipped upside-down;
- The target image is turned to the right by 90 deg.

Implementation description:

The estimator (see IOrientationEstimator in IOrientationEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format and returns the estimation result;

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

The **OrientationType enumeration** contains all possible results of the Orientation estimation:

```
enum OrientationType : uint32_t {
    OT_NORMAL = 0,      //!< Normal orientation of image
    OT_LEFT = 1,        //!< Image is turned left by 90 deg
    OT_UPSIDE_DOWN = 2, //!< Image is flipped upsidedown
    OT_RIGHT = 3        //!< Image is turned right by 90 deg
};
```

API structure name:

IOrientationEstimator

Plan files:

- orientation_v2_cpu.plan
- orientation_v2_cpu-avx2.plan
- orientation_v2_gpu.plan

4.3.5 Detector variants

Supported detector variants:

- FaceDetV1
- FaceDetV2
- FaceDetV3
- FaceDetV3m

There are two basic detector families. The first of them includes two detector variants: FaceDetV1 and FaceDetV2. The second family also includes two detector variants: FaceDetV3 and FaceDetV3m. FaceDetV3 is the most precise detector. For this type of detector can be passed sensor type. In terms of performance FaceDetV3 is similar to FaceDetV1 detector. FaceDetV3m is the faster but less precise version of FaceDetV3.

User code may specify necessary detector type while creating *IDetector* object using parameter.

> FaceDetV1 and FaceDetV2 performance depends on number of faces on image and image complexity. FaceDetV3 and FaceDetV3m performance depends only on the target image resolution.

> FaceDetV3 and FaceDetV3m work faster with batched redetect.

> FaceDetV3 and FaceDetV3m support asynchronous methods for detection and redetection. FaceDetV1 and FaceDetV2 will return not implemented error.

4.3.6 FaceDetV1 and FaceDetV2 Configuration

FaceDetV1 detector is more precise and FaceDetV2 works two times faster (See appendix A chapter "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.

4.3.7 FaceDetV3 and FaceDetV3m Configuration

FaceDetV3 and FaceDetV3m detect faces from `minFaceSize` to `minFaceSize` * 32. You can change the minimum size 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 and FaceDetV3m 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 chapter "Migration guide".

4.3.8 Face Alignment

4.3.8.1 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.

4.3.8.2 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*.
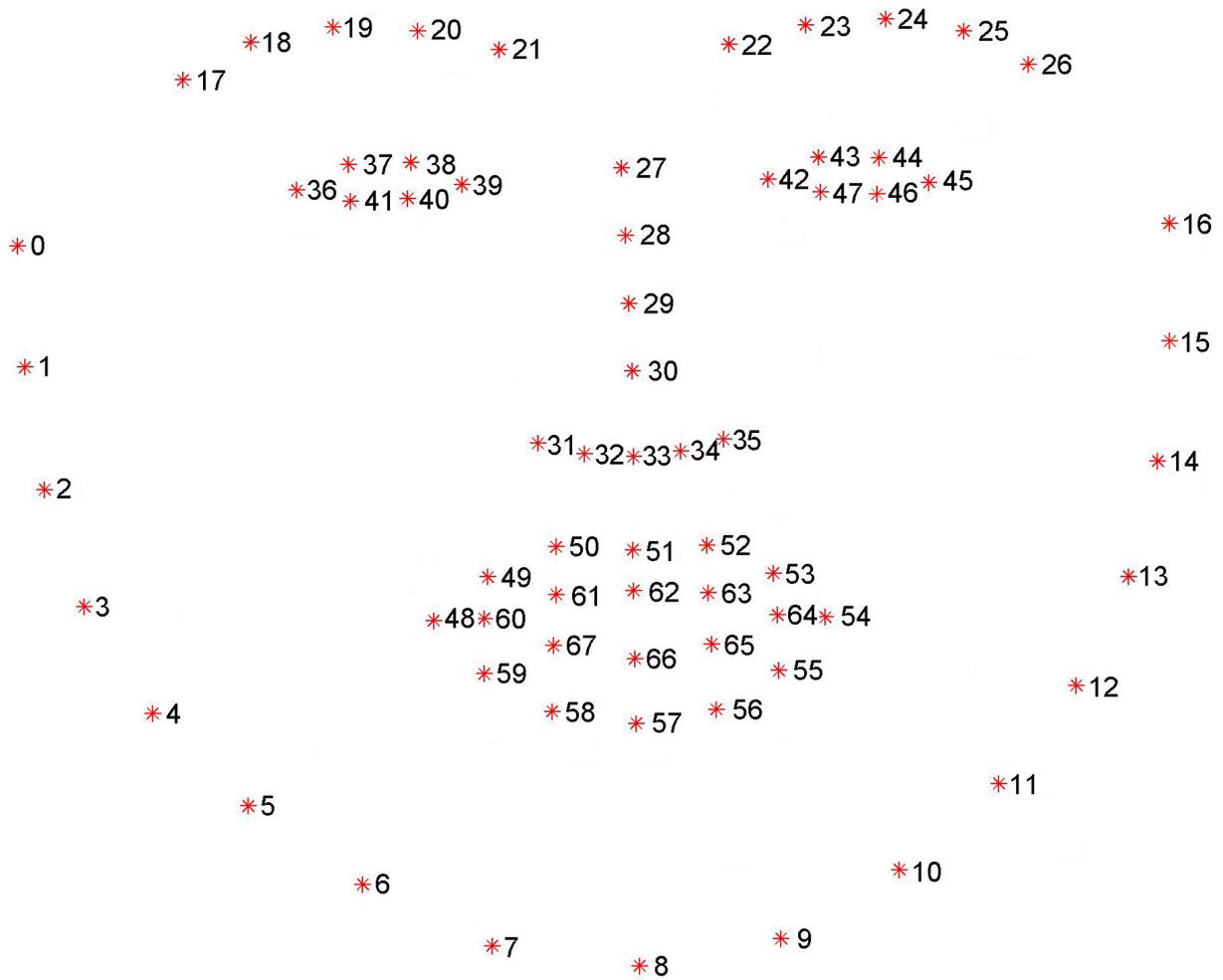
Figure 2: 68-point face alignment

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

Point	Error (pixels)	Point	Error (pixels)	Point	Error (pixels)	Point	Error (pixels)
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 Face Landmarks Detector

Every kind of detector provides an interface to find face landmarks. If you have a face detection without landmarks we provide additional interface to request them. The detection of landmarks is performed by the *IFaceLandmarksDetector* object. The functions of interest are *detectLandmarks5()* and *detectLandmarks68*. They need images and detections.

4.5 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.

Also, *IHumanDetector* implements *detectAsync()* which allows you to asynchronously detect human bodies parameters on multiple images.

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

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

4.5.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.
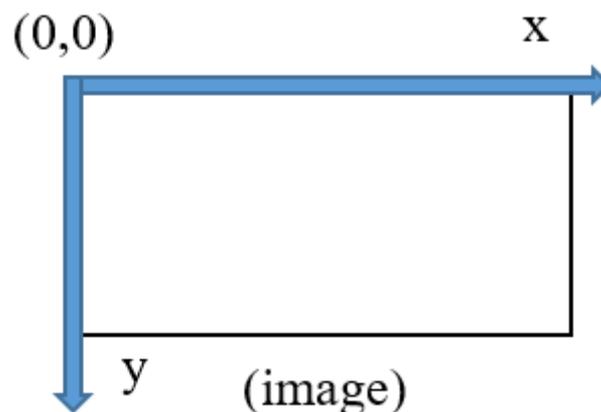
Figure 3: Source image coordinate system

4.5.2 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.

4.5.3 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.

4.5.4 Camera position requirements

In general, you should locate the camera for human detection according to the image below.
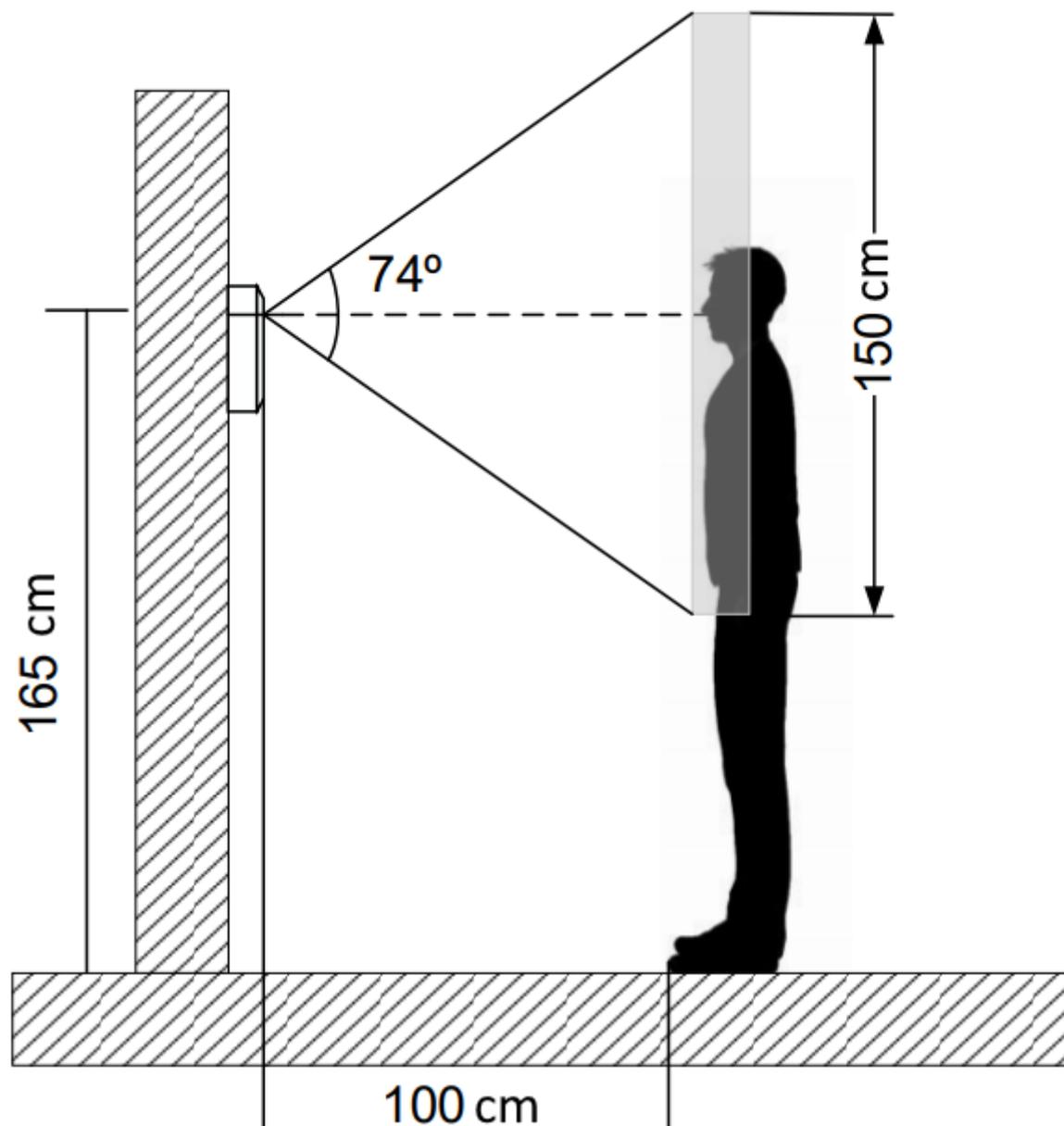
Figure 4: Camera position for human detection

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.
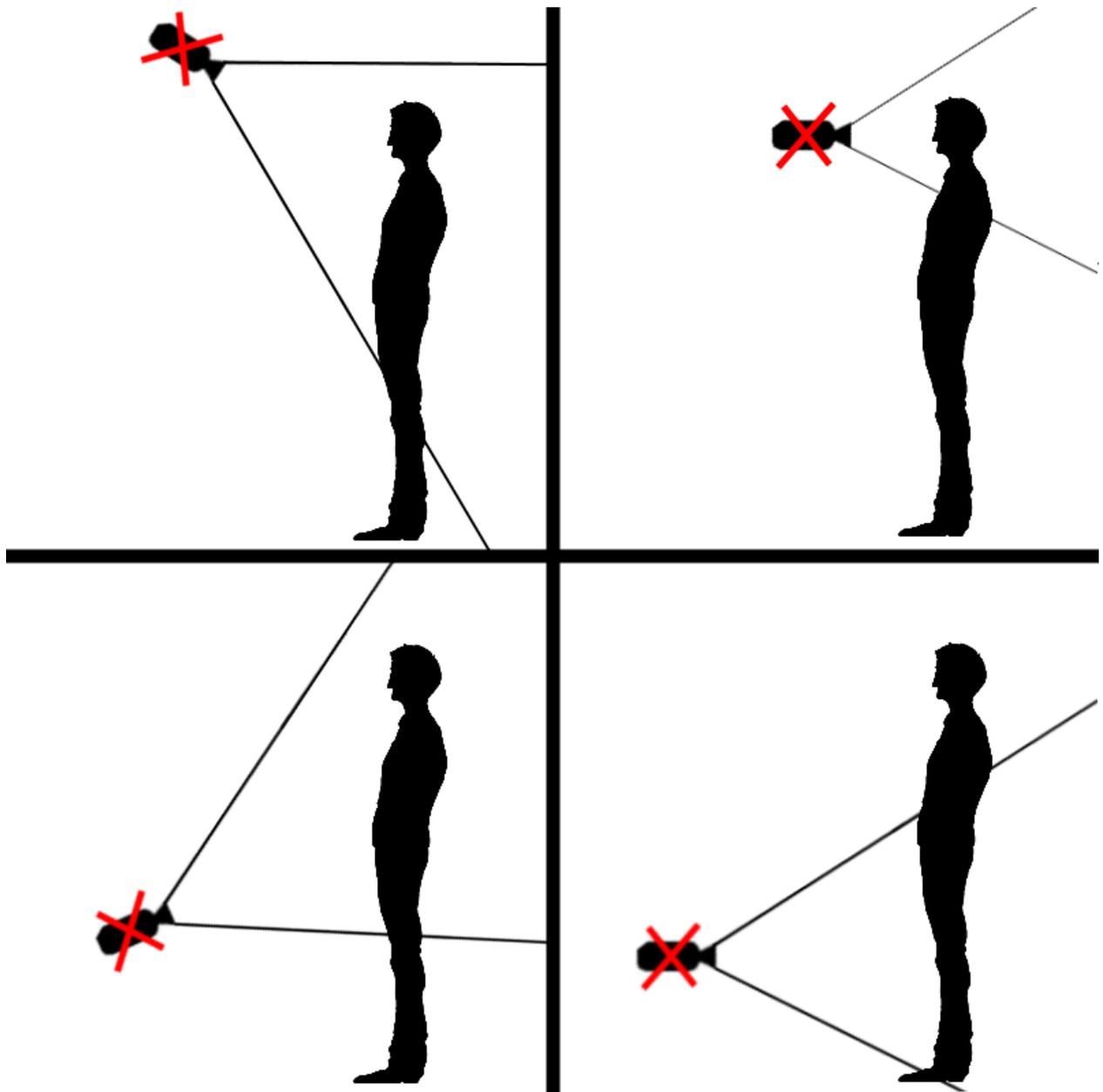
Figure 5: Wrong camera positions

4.5.5 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.

Also, *IHumanDetector* implements *redetectAsync()* which allows you to asynchronously redetect human bodies parameters on multiple images.

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

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

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

4.5.6 Human Keypoints

The image below shows the keypoints detected for a human body.
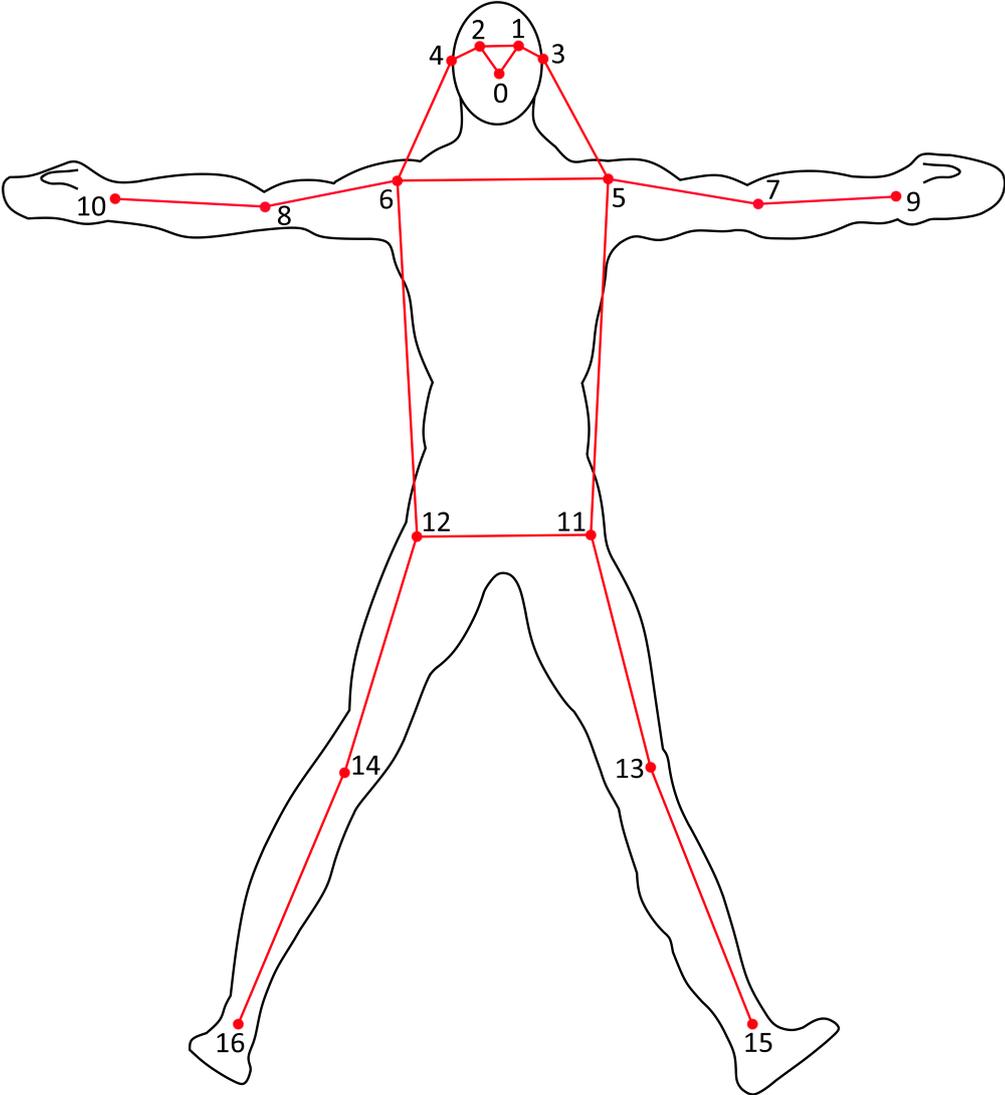
Point	Body Part	Point	Body Part
0	Nose	9	Left Wrist
1	Left Eye	10	Right Wrist

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		

Cases that increase the probability of error:

- Non-standard poses (head below the shoulders, vertical twine, lying head to the camera, etc.);
- Camera position from above at a large angle;
- Sometimes estimator predicts invisible points with high score, especially for points of elbows, wrists, ears.

Human detector provides an interface to find human landmarks. If you have a human structure without landmarks we provide additional interface to request them. The detection of landmarks is performed by the *IHumanLandmarksDetector* object. The function of interest is *detectLandmarks()*. It needs images and detections.

4.5.7 Detection

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

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

4.5.8 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.

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

4.6 HumanFace Detection. Face to body association

This functionality enables you to detect the bodies and faces of people and perform an association between them, determining whether the detected face and body belong to the same person.

This detector contains the implementation of both Human and Face(FaceDetV3) detectors. This means that all the requirements, constraints and recommendations for quality improvement imposed for these detectors will be relevant for the HumanFace detector.

Detector operation algorithm:

- human detection
- face detection
- determination of an association for each detection

Figure 7: HumanFace detection

4.6.1 HumanFace redetection

To perform redetection, you need to separately redetect body and face.

4.6.2 Performance

User can skip computation of associations by selecting according `HumanFaceDetectionType` for `detect()` method, if he doesn't need this functionality. In such case, we estimate performance gain about 5% on cpu and about 20% on gpu devices. The more faces and bodies represented in image, the more gain user will enjoy after association skip.

4.6.3 Main results

There are two output structures:

- **HumanFaceBatch**
- **HumanFaceAssociations**

The **HumanFaceBatch** contains three arrays - face detections, human detections and associations:

```
struct IHumanFaceBatch : public IRefCounted {
    virtual Span<const Detection> getHumanDetections(size_t index = 0)
        const noexcept = 0;
    virtual Span<const Detection> getFaceDetections(size_t index = 0)
        const noexcept = 0;
    virtual Span<const HumanFaceAssociation> getAssociations(size_t
        index = 0) const noexcept = 0;
};
```

The **HumanFaceAssociation structure** contains results of the association:

```
struct HumanFaceAssociation {
    uint32_t humanId;
    uint32_t faceId;
    float score;
};
```

There are two groups of fields:

1. The first group contains body and face detection indexes:

```
    uint32_t humanId;
    uint32_t faceId;
```

2. The second group contains association score:

```
    float score;
```

The score is defined in [0,1] range.

Associations and detections whose scores are lower than the threshold will be rejected and not returned in the results.

> See ConfigurationGuide.pdf ("HumanFace settings" section) for more information about thresholds and configuration parameters.

4.6.4 minFaceSize

This detector could detect faces with size 20 px and more (`minFaceSize` parameter) and humans with size 100 px and more. In case if such small faces and humans are not required, user could change the `minFaceSize` parameter in the config.

Before processing, the images will be resized by `minFaceSize/20` times. For example, if the value is `minFaceSize=50`, then the image will be additionally resized by `minFaceSize=50/20=2.5` times.

Detector works faster with larger value of `minFaceSize`.

4.7 Head Detection

This functionality enables you to detect the heads of people.

This detector implementation is similar to Face(FaceDetV3) detectors. This means that all the requirements, constraints and recommendations for quality improvement imposed for this detector will be relevant for the Head detector.

Object detection is performed by the *IHeadDetector*. 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 heads only in the given location).

4.7.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.
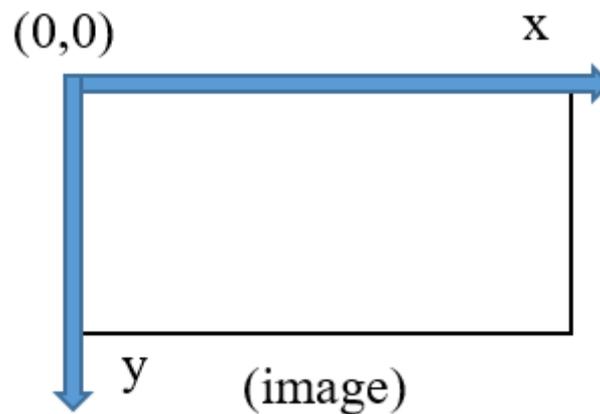
Figure 8: Source image coordinate system

4.7.2 Main results

Output structures:

- **DetectionBatch**

The **DetectionBatch** contains an array of head detections:

```
struct IDetectionBatch : public IRefCounted {

    virtual size_t getSize() const noexcept = 0;
```

```
        virtual Span<const Detection> getDetections(size_t index = 0) const
            noexcept = 0;

    };
```

4.7.3 minHeadSize

This detector could detect heads with size 20 px and more (`minHeadSize` parameter). In case if such small heads, user could change the `minHeadSize` parameter in the config.

Before processing, the images will be resized by `minHeadSize/20` times. For example, if the value is `minHeadSize=50`, then the image will be additionally resized by `minHeadSize=50/20=2.5` times.

Detector works faster with larger value of `minHeadSize`.

5 Image Warping

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

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

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

The warper (see IWarper in IWarper.h):

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

Note: Method *warpAsync()* is experimental, and it's interface may be changed in the future. **Note:** Method *warpAsync()* is not marked as noexcept and may throw an exception.
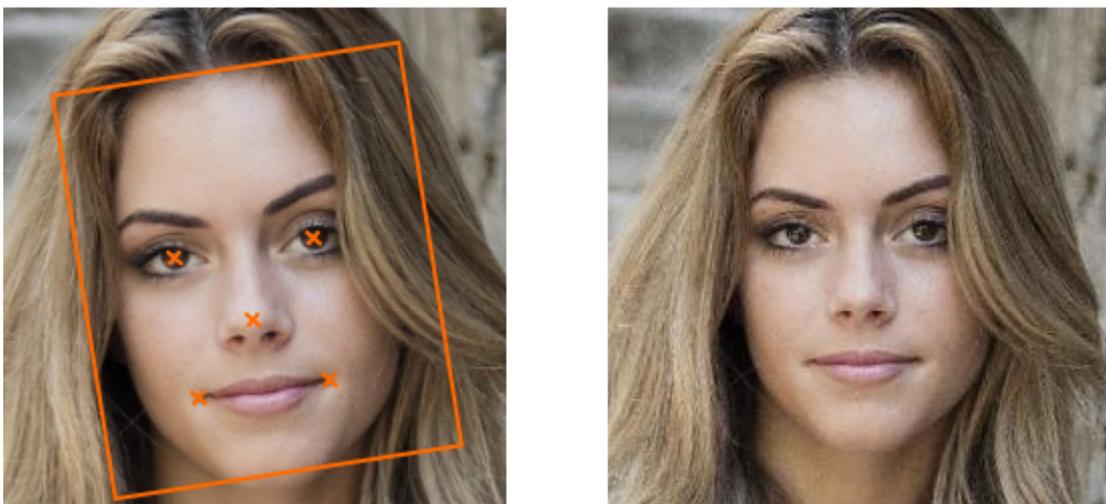
Figure 9: Face warping

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

6 Parameter Estimation Facility

6.1 Overview

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

6.2 Use cases

6.2.1 ISO estimation

LUNA SDK provides algorithms for image check according to the requirements of the ISO/IEC 19794-5:2011 standard and compatible standards.

The requirements can be found on the official website: https://www.iso.org/obp/ui/#iso:std:iso-iec:19794:-5:en.

The following algorithms are provided:

- Head rotation angles (pitch, yaw, and roll angles). According to section "7.2.2 Pose" in the standard, the angles should be +/- 5 degrees from frontal in pitch and yaw, less than +/- 8 degrees from frontal in roll. See additional information about the algorithm in section "Head Pose".

- Gaze. See section "7.2.3 Expression" point "e" of the standard. See additional information about the algorithm in section "Gaze Estimation".

- Mouth state (opened, closed, occluded) and additional properties for smile (regular smile, smile with teeths exposed) See section "7.2.3 Expression" points "a", "b", and "c" of the standard. See additional information about the algorithm in section "Mouth Estimation".

- Quality of the image:
  - Contrast and saturation (insufficient or too large exposure). See sections "7.2.7 Subject and scene lighting" and "7.3.2 Contrast and saturation" of the standard.
  - Blurring. See section "7.3.3 Focus and depth of field" of the standard.
  - Specularity. See section "7.2.8 Hot spots and specular reflections" and "7.2.12 Lighting artefacts" of the standard.
  - Uniformity of illumination. See sections "7.2.7 Subject and scene lighting" and "7.2.12 Lighting artefacts" of the standard.

  See additional information about the algorithm in section "Image Quality Estimation".

- Glasses state (no glasses, glasses, sunglasses). See section "7.2.9 Eye glasses" of the standard. See additional information about the algorithm in section "Glasses Estimation".

- Eyes state (for each eye: opened, closed, occluded). See sections "7.2.3 Expression" point "a", "7.2.11 Visibility of pupils and irises" and "7.2.13 Eye patches" of the standard. See additional information about the algorithm in section "Eyes Estimation".

- Natural light estimation. See section "7.3.4 Unnatural colour" of the standard. See additional information about the algorithm in section "Natural Light Estimation".

- Eybrows state: neutral, raised, squinting, frowning. See section "7.2.3 Expression" points "d", "f", and "g" of the standard. See additional information about the algorithm in section "Eyebrows estimation".

- Position of a person's shoulders in the original image: the shoulders are parallel to the camera or not. See section "7.2.5 Shoulders" of the standard. See additional information about the algorithm in section "Portrait Style Estimation".

- Headwear. Checks if there is a headwear on a person or not. Several types of headwear can be estimated. See section "B.2.7 Head coverings" of the standard. See additional information about the algorithm in section "Headwear Estimation".

- Red eyes estimation. Checks if there is a red eyes effect. See section "7.3.4 Unnatural colour" of the standard. See additional information about the algorithm in section "Red Eyes Estimation".

- Radial distortion estimation. See section "7.3.6 Radial distortion of the camera lens" of the standard. See additional information about the algorithm in section "Fish Eye Estimation".

- Image type estimation: color, grayscale, infrared. See section "7.4.4 Use of near infra-red cameras" of the standard. See additional information about the algorithm in section "Grayscale, color or infrared Estimation".

- Background estimation: background uniformity and if a background is too light or too dark. See section "B.2.9 Backgrounds" of the standard. See additional information about the algorithm in section "Background Estimation".

6.3 Best shot selection functionality

6.3.1 BestShotQuality Estimation

Name: BestShotQualityEstimator

Algorithm description:

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

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

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

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

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

Head Pose determines person head rotation angles in 3D space, namely pitch, yaw and roll.
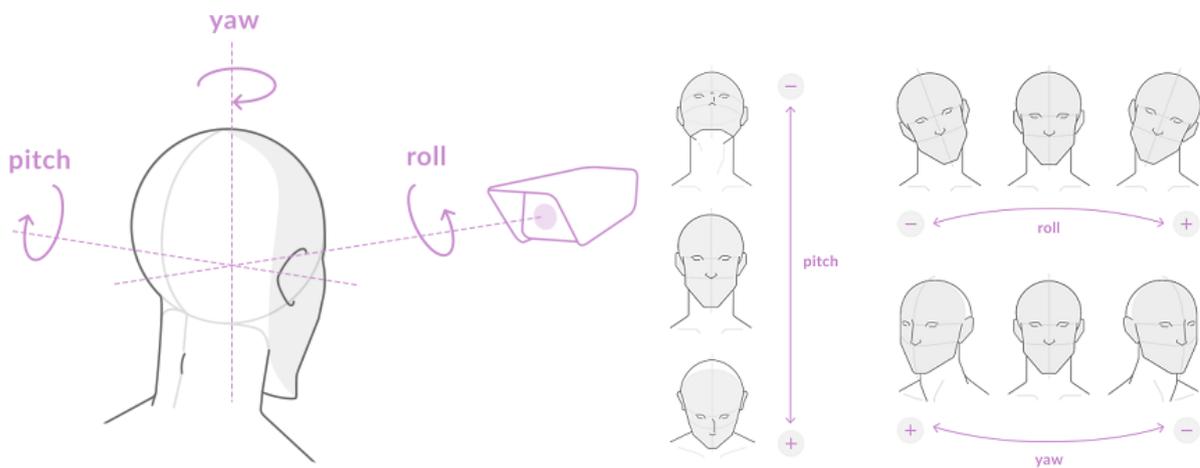
Figure 10: Head pose

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

Head pose estimation characteristics:

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

Implementation description:

The estimator (see IBestShotQualityEstimator in IEstimator.h):

- Implements the *estimate()* function that needs fsdk::Image in R8G8B8 format, fsdk::Detection structure of corresponding **source image** (see section "Detection structure" in chapter "Face detection facility"), fsdk::IBestShotQualityEstimator::EstimationRequest structure and fsdk::IBestShotQualityEstimator::EstimationResult to store estimation result;

- Implements the *estimate()* function that needs the span of fsdk::Image in R8G8B8 format, the span of fsdk::Detection structures of corresponding **source images** (see section "Detection structure" in chapter "Face detection facility"), fsdk::IBestShotQualityEstimator::EstimationRequest structure and span of fsdk::IBestShotQualityEstimator::EstimationResult to store estimation results.

- Implements the *estimateAsync()* function that needs fsdk::Image in R8G8B8 format, fsdk::Detection structure of corresponding source image (see section "Detection structure" in chapter "Face detection facility"), fsdk::IBestShotQualityEstimator::EstimationRequest structure;

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

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

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

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

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

Head Pose accuracy:

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

Table 3: "Head pose prediction precision"

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

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

API structure name:

IBestShotQualityEstimator

Plan files:

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

6.3.2 Image Quality Estimation

Name: QualityEstimator

Algorithm description:

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

This estimator is designed to determine the image quality. You can estimate the image according to the following criteria:

- 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).

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

Figure 11: Blurred image (left), not blurred image (right)

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

Figure 13: Light image (left), good quality image (right)

Figure 14: Image with uneven illumination (left), image with even illumination (right)

Figure 15: Image with specularity - image contains flares on face (left), good quality image (right)

Implementation description:

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-exposed (i.e., too dark);
- Image is over-exposed (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.

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 (see below) 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.

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

> 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".

> 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.

The **Quality structure** contains results of the estimation made by first CNN. Each estimation is given in normalized [0, 1] range:

```
struct Quality {
    float light;    //!< image overlighting degree. 1 - ok, 0 -
        overlighted.
    float dark;     //!< image darkness degree. 1 - ok, 0 - too dark.
    float gray;     //!< image grayness degree 1 - ok, 0 - too gray.
    float blur;     //!< image blur degree. 1 - ok, 0 - too blured.
    inline float getQuality() const noexcept;   //!< complex estimation
        of quality. 0 - low quality, 1 - high quality.
};
```

The **SubjectiveQuality structure** contains results of the estimation made by second CNN. Each estimation is given in normalized [0, 1] range:

```
struct SubjectiveQuality {
    float blur;         //!< image blur degree. 1 - ok, 0 - too blured.
    float light;        //!< image brightness degree. 1 - ok, 0 - too
        bright;
    float darkness;     //!< image darkness degree. 1 - ok, 0 - too dark
        ;
    float illumination; //!< image illumination uniformity degree. 1 -
        ok, 0 - is too illuminated;
    float specularity;  //!< image specularity degree. 1 - ok, 0 - is
        not specular;
    bool isBlurred;     //!< image is blurred flag;
    bool isHighlighted; //!< image is overlighted flag;
    bool isDark;        //!< image is too dark flag;
    bool isIlluminated; //!< image is too illuminated flag;
    bool isNotSpecular; //!< image is not specular flag;
```

```
        inline bool isGood() const noexcept;     //!< if all boolean flags
            are false returns true - high quality, else false - low quality.
    };
```

Recommended thresholds:

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

Table 4: "Image quality estimator recommended thresholds"

Threshold	Recommended value
blurThreshold	0.61
darknessThreshold	0.50
lightThreshold	0.57
illuminationThreshold	0.1
specularityThreshold	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.

Configurations:

See the "Quality estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IQualityEstimator

Plan files:

- model_subjective_quality_v2_cpu.plan
- model_subjective_quality_v2_cpu-avx2.plan
- model_subjective_quality_v2_gpu.plan

6.4 Attributes estimation functionality

6.4.1 Face Attribute Estimation

Name: AttributeEstimator

Algorithm description:

> 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: determines person's gender;

The Attribute estimator returns `Ethnicity estimation structure`. Each estimation is given in normalized [0, 1] range.

The Ethnicity estimation structure looks like the struct below:

```
struct EthnicityEstimation {
 float africanAmerican;
 float indian;
 float asian;
 float caucasian;

 enum Ethnicities {
  AfricanAmerican = 0,
  Indian,
  Asian,
  Caucasian,
  Count
 };

 /**
  * @brief Returns ethnicity with greatest score.
  * @see EthnicityEstimation::Ethnicities for more info.
  * */
 inline Ethnicities getPredominantEthnicity() const;

 /**
  * @brief Returns score of required ethnicity.
  * @param [in] ethnicity ethnicity.
  * @see EthnicityEstimation::Ethnicities for more info.
  * */
 inline float getEthnicityScore(Ethnicities ethnicity) const;
};
```

Implementation description:

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

Recommended thresholds:

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

Table 5: "Attribute estimator recommended thresholds"

Threshold	Recommended value
genderThreshold	0.5
adultThreshold	0.2

Accuracy:

Age:

- For cooperative (see "Appendix B. Glossary") conditions: average error depends on person age, see table below for additional details. Estimation accuracy is 2.3.

Gender:

- Estimation accuracy in cooperative mode is 99.81% with the threshold 0.5;
- Estimation accuracy in non-cooperative mode is 92.5%.

Table 6: "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

Age (years)	Average error (years)
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

> In earlier releases of Luna SDK Attribute estimator worked poorly in non-cooperative mode (only 56% gender estimation accuracy), 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.

Configurations:

See the "AttributeEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IAttributeEstimator

Plan files:

- attributes_estimation_v5_cpu.plan
- attributes_estimation_v5_cpu-avx2.plan
- attributes_estimation_v5_gpu.plan

6.4.2 Child Estimation

Name: ChildEstimator

Algorithm description:

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).

Implementation description:

The estimator (see IChildEstimator in IChildEstimator.h):

- Implements the *estimate()* function accepts **warped source image** (see chapter "Image warping" for details). 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.

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

```
struct ChildEstimation {
    float childScore = 0.0f;    //!< Numerical value in range [0, 1].
        Show the person is child or not.
    bool isChild = false        //!< Person is child (true) or not (
        false).
};
```

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `ChildEstimator ::Settings` section. By default, this threshold value is set to optimal.

Table 7: "Child estimator recommended threshold"

Threshold	Recommended value
ChildThreshold	0.8508

Configurations:

See the "ChildEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IChildEstimator

Plan files:

- childnet_estimation_flwr_cpu.plan
- childnet_estimation_flwr_cpu-avx2.plan
- childnet_estimation_flwr_gpu.plan

6.4.3 Credibility Check Estimation

Name: CredibilityCheckEstimator

Algorithm description:

This estimator estimates reliability of a person.

Implementation description:

The estimator (see ICredibilityCheckEstimator in ICredibilityCheckEstimator.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.

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

```
struct CredibilityCheckEstimation {
    float value;                            //!< estimation in [0,1] range
         .
                                            //!< The closer the score to
                                               1,
                                            //!< the more likely that
                                               person is reliable.

    CredibilityStatus credibilityStatus;   //!< estimation result
                                            //!< (@see CredibilityStatus
                                               enum).
};
```

Enumeration of possible credibility statuses:

```
enum class CredibilityStatus : uint8_t {
    Reliable = 1,                           //!< person is reliable
    NonReliable = 2                         //!< person is not reliable
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `CredibilityEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 8: "Credibility check estimator recommended threshold"

Threshold	Recommended value
reliableThreshold	0.5

Filtration parameters:

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

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

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

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

Attribute	Minimum value
blur	0.61
light	0.57

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

Attribute	Minimum value
age	18

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

Attribute	State
overlapped	false

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

Attribute	Minimum value
detection size	100

Detection size is detection width.

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

Configurations:

See the "Credibility Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ICredibilityCheckEstimator

Plan files:

- credibility_check_cpu.plan
- credibility_check_cpu-avx2.plan
- credibility_check_gpu.plan

6.5 Facial Hair Estimation

Name: FacialHairEstimator

Algorithm description:

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).

Implementation description:

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.

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
};
```

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.

Filtration parameters:

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

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

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

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

Attribute	State
result	fsdk::MedicalMask::NoMask

Attribute	Minimum value
detection size	40

> Detection size is detection width.

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

API structure name:

IFacialHairEstimator

Plan files:

- face_hair_v2_cpu.plan
- face_hair_v2_cpu-avx2.plan
- face_hair_v2_gpu.plan

6.6 Natural Light Estimation

Name: NaturalLightEstimator

Algorithm description:

This estimator aims to detect a natural light on the source face image. It can return the next results:

- Light is not natural on the face image (see `LightStatus::NonNatural` field in the LightStatus enum);
- Light is natural on the face image (see `LightStatus::Natural` field in the LightStatus enum).

Implementation description:

The estimator (see INaturalLightEstimator in INaturalLightEstimator.h):

- Implements the *estimate()* function that accepts **source warped image** in R8G8B8 format and NaturalLightEstimation 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 NaturalLightEstimation structures to return results of estimation.

The **LightStatus enumeration** contains all possible results of the NaturalLight estimation:

```
enum class LightStatus : uint8_t {
    NonNatural = 0,                 //!< light is not natural
    Natural = 1                     //!< light is natural
};
```

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

```
struct NaturalLightEstimation {
    LightStatus status;             //!< estimation result (@see
        NaturalLight enum).
    float score;                    //!< Numerical value in range [0,
        1].
};
```

There are two groups of the fields:

1. The first group contains only the result enum:

```
        LightStatus status;             //!< estimation result (@see
            LightStatus enum).
```

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

2. The second group contains scores:

```
        float score;                        //!< Numerical value in range [0,
            1].
```

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.

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `NaturalLightEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 17: "Natural light estimator recommended threshold"

Threshold	Recommended value
naturalLightThreshold	0.5

Filtration parameters:

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

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

Attribute	State
result	fsdk::MedicalMask::NoMask

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

Attribute	Minimum value
blur	0.5

Also `fsdk::GlassesEstimation` must not be equal to `fsdk::GlassesEstimation::SunGlasses`.

Configurations:

See the "Natural Light Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

INaturalLightEstimator

Plan files:

- natural_light_cpu.plan
- natural_light_cpu-avx2.plan
- natural_light_gpu.plan

6.7 Fish Eye Estimation

Name: FishEyeEstimator

Algorithm description:

This estimator aims to detect a fish eye effect on the source face image. It can return the next fish eye effect status results:

- There is no fish eye effect on the face image (see `FishEye::NoFishEyeEffect` field in the FishEye enum);
- There is fish eye effect on the face image (see `FishEye::FishEyeEffect` field in the FishEye enum).

Implementation description:

The estimator (see IFishEyeEstimator in IFishEyeEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format, face detection and FishEyeEstimation structure to return results of estimation;

- Implements the *estimate()* function that accepts `fsdk::Span` of the **source images** in R8G8B8 format, `fsdk::Span` of the face detections and `fsdk::Span` of the FishEyeEstimation structures to return results of estimation.

The **FishEye enumeration** contains all possible results of the FishEye estimation:

```
enum class FishEye {
    NoFishEyeEffect = 0,  //!< no fish eye effect
    FishEyeEffect = 1     //!< with fish eye effect
};
```

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

```
struct FishEyeEstimation {
    FishEye result;       //!< estimation result (@see FishEye enum)
    float score;          //!< fish eye effect score
};
```

There are two groups of the fields:

1. The first group contains only the result enum:

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

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

2. The second group contains scores:

```
float score;          //!< fish eye effect score
```

The scores group contains the estimation score.

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `FishEyeEstimator ::Settings` section. By default, this threshold value is set to optimal.

Table 20: "Fish Eye estimator recommended threshold"

Threshold	Recommended value
fishEyeThreshold	0.5

Recommended scenarios of algorithm usage:

Data domain: Cooperative mode only. It is means:

- High image quality;
- Frontal face looking directly at the camera.

Filtration parameters:

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

Table 21: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-8…8]
yaw	[-8…8]
roll	[-8…8]

Table 22: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

> Detection size is detection width.

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

Configurations:

See the "Fish Eye Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IFishEyeEstimator

Plan files:

- fisheye_v2_cpu.plan
- fisheye_v2_cpu-avx2.plan
- fisheye_v2_gpu.plan

6.8 Eyebrows Estimation

Name: EyeBrowEstimator

Algorithm description:

This estimator is trained to estimate eyebrow expressions. The EyeBrowEstimator returning four scores for each possible eyebrow expression. Which are - `neutral`, `raised`, `squinting`, `frowning`. Possible scores are in the range [0, 1].

If score closer to 1, it means that detected expression on image is more likely to real expression and closer to 0 otherwise.

Along with the output score value estimator also returns an enum value (EyeBrowState). The index of the maximum score determines the EyeBrow state.

Implementation description:

The estimator (see IEyeBrowEstimator in IEyeBrowEstimator.h):

- Implements the *estimate()* function accepts **warped source image**. Warped image is received from the warper (see `IWarper::warp()`); Output estimation is a structure `fsdk::EyeBrowEstimation`.

- Implements the *estimate()* function that needs the span of **warped source images** and span of structure `fsdk::EyeBrowEstimation`. Output estimation is a span of structure `fsdk::EyeBrowEstimation`.

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

```
struct EyeBrowEstimation {
    /**
     * @brief EyeBrow estimator output enum.
     * This enum contains all possible estimation results.
    **/
    enum class EyeBrowState {
        Neutral = 0,
        Raised,
        Squinting,
        Frowning
    };


    float neutralScore;       //!< 0(not neutral)..1(neutral).
    float raisedScore;        //!< 0(not raised)..1(raised).
    float squintingScore;     //!< 0(not squinting)..1(squinting).
    float frowningScore;      //!< 0(not frowning)..1(frowning).
    EyeBrowState eyeBrowState; //!< EyeBrow state
```

```
        };
```

Filtration parameters:

Table 23: "Requirements for `fsdk::EyeBrowEstimation`"

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

Table 24: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

> Detection size is detection width.

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

API structure name:

IEyeBrowEstimator

Plan files:

- eyebrow_estimation_v2_cpu.plan
- eyebrow_estimation_v2_cpu-avx2.plan
- eyebrow_estimation_v2_gpu.plan

6.9 Portrait Style Estimation

Name: PortraitStyleEstimator

Algorithm description:

This estimator is designed to estimate the position of a person's shoulders in the original image. It can return the following results:

- The shoulders are not parallel to the camera (see the `PortraitStyleStatus::NonPortrait` field in the PortraitStyleStatus enum);
- Shoulders are parallel to the camera (see the `PortraitStyleStatus::Portrait` field in the PortraitStyleStatus enum);
- Shoulders are hidden (see the `PortraitStyleStatus::HiddenShoulders` field in the PortraitStyleStatus enum);

Implementation description:

The Estimator (see IPortraitStyleEstimator in IPortraitStyleEstimator.h):

- Implements *estimate()* function that accepts R8G8B8 **source image**, detection and PortraitStyleEstimation structure to return estimation results;

- Implements an *estimate()* function that accepts `fsdk::Span` of R8G8B8 **source images**, `fsdk::Span` of detections, and `fsdk::Span` of PortraitStyleEstimation structures to return estimation results.

The **PortraitStyleStatus enumeration** contains all possible results of the PortraitStyle estimation:

```
enum class PortraitStyleStatus : uint8_t {
    NonPortrait = 0,        //!< NonPortrait
    Portrait = 1,           //!< Portrait
    HiddenShoulders = 2     //!< HiddenShoulders
};
```

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

```
struct PortraitStyleEstimation {
    PortraitStyleStatus status; //!< estimation result (@see
        PortraitStyleStatus enum).
    float nonPortraitScore;            //!< numerical value in range
        [0, 1]
    float portraitScore;               //!< numerical value in range
        [0, 1]
    float hiddenShouldersScore;        //!< numerical value in range
        [0, 1]
```

```
    };
```

There are two groups of the fields:

1. The first group contains the enum:

```
        PortraitStyleStatus status; //!< estimation result (@see
            PortraitStyleStatus enum).
```

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

2. The second group contains score:

```
        float nonPortraitScore;              //!< numerical value in range
            [0, 1]
        float portraitScore;                 //!< numerical value in range
            [0, 1]
        float hiddenShouldersScore;          //!< numerical value in range
            [0, 1]
```

The scores are defined in [0,1] range.

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `PortraitStyleEstimator` `::Settings` section. By default, this threshold value is set to optimal.

Table 25: "Portrait Style estimator recommended threshold"

Threshold	Recommended value
notPortraitStyleThreshold	0.2
portraitStyleThreshold	0.35
hiddenShouldersThreshold	0.2

Filtration parameters:

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

Type of preferable detector is FaceDetV3.

Table 26: "Requirements for `Detector`"

Attribute	Min face size
result	40

Table 27: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Maximum value
yaw	20.0
pitch	20.0
roll	20.0

Configurations:

See the "Portrait Style Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IPortraitStyleEstimator

Plan files:

- portrait_style_v3_cpu.plan
- portrait_style_v3_cpu-avx2.plan
- portrait_style_v3_gpu.plan

6.10 DynamicRange Estimation

Name: DynamicRangeEstimator

Algorithm description:

This estimator is designed to estimate dynamic range of an original image with person's face.

Implementation description:

The Estimator (see IDynamicRangeEstimator in IDynamicRangeEstimator.h):

- Implements *estimate()* function that accepts R8G8B8 **source image**, detection and DynamicRangeEstimation structure to return estimation results;

- Implements an *estimate()* function that accepts `fsdk::`Span of R8G8B8 **source images**, `fsdk::`Span of detections, and `fsdk::`Span of DynamicRangeEstimation structures to return estimation results.

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

```
struct DynamicRangeEstimation {
    float dynamicRangeScore;              //!< numerical value in range
        [0, 1]
};
```

Result estimation DynamicRangeEstimation contains the target score.

```
    float dynamicRangeScore;              //!< numerical value in range
        [0, 1]
```

The score is defined in [0,1] range.

Recommended thresholds:

Table below contains recommended user's threshold.

Table 28: "Dynamic Range estimator recommended threshold"

Threshold	Recommended value
threshold	0.5

API structure name:

IDynamicRangeEstimator

Plan files:

DynamicRangeEstimator does not use any additional models (plans, files and etc.), this is an ISO-based algorithm that is currently only implemented on CPU devices.

6.11 Headwear Estimation

Name: HeadWearEstimator

Algorithm description:

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

- There is headwear (see `HeadWearState::Yes` field in the HeadWearState enum);
- There is no headwear (see `HeadWearState::No` field in the HeadWearState enum);

And this headwear type results:

- There is no headwear on the head (see `HeadWearType::NoHeadWear` field in the HeadWearType enum);
- There is baseball cap on the head (see `HeadWearType::BaseballCap` field in the HeadWearType enum);
- There is beanie on the head (see `HeadWearType::Beanie` field in the HeadWearType enum);
- There is peaked cap on the head (see `HeadWearType::PeakedCap` field in the HeadWearType enum);
- There is shawl on the head (see `HeadWearType::Shawl` field in the HeadWearType enum);
- There is hat with ear flaps on the head (see `HeadWearType::HatWithEarFlaps` field in the HeadWearType enum);
- There is helmet on the head (see `HeadWearType::Helmet` field in the HeadWearType enum);
- There is hood on the head (see `HeadWearType::Hood` field in the HeadWearType enum);
- There is hat on the head (see `HeadWearType::Hat` field in the HeadWearType enum);
- There is something other on the head (see `HeadWearType::Other` field in the HeadWearType enum);

Implementation description:

The estimator (see IHeadWearEstimator in IHeadWearEstimator.h):

- Implements the *estimate()* function that accepts **warped image** in R8G8B8 format and HeadWearEstimation 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 HeadWearEstimation structures to return results of estimation.

The **HeadWearState enumeration** contains all possible results of the Headwear state estimation:

```
enum class HeadWearState {
    Yes = 0,          //< there is headwear
    No,               //< there is no headwear
    Count
```

```
    };
```

The **HeadWearType enumeration** contains all possible results of the Headwear type estimation:

```
enum class HeadWearType : uint8_t {
    NoHeadWear = 0,       //< there is no headwear on the head
    BaseballCap,          //< there is baseball cap on the head
    Beanie,               //< there is beanie on the head
    PeakedCap,            //< there is peaked cap on the head
    Shawl,                //< there is shawl on the head
    HatWithEarFlaps,      //< there is hat with ear flaps on the head
    Helmet,               //< there is helmet on the head
    Hood,                 //< there is hood on the head
    Hat,                  //< there is hat on the head
    Other,                //< something other is on the head
    Count
};
```

The **HeadWearStateEstimation structure** contains results of the Headwear state estimation:

```
struct HeadWearStateEstimation {
    HeadWearState result; //!< estimation result (@see HeadWearState
        enum)
    float scores[static_cast<int>(HeadWearState::Count)]; //!<
        estimation scores

    /**
     * @brief Returns score of required headwear state.
     * @param [in] state headwear state.
     * @see HeadWearState for more info.
     * */
    inline float getScore(HeadWearState state) const;
};
```

There are two groups of the fields:

1. The first group contains only the result enum:

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

2. The second group contains scores:

```
        float scores[static_cast<int>(HeadWearState::Count)]; //!<
            estimation scores
```

The **HeadWearTypeEstimation structure** contains results of the Headwear type estimation:

```
    struct HeadWearTypeEstimation {
        HeadWearType result; //!< estimation result (@see HeadWearType enum)
        float scores[static_cast<int>(HeadWearType::Count)]; //!< estimation
            scores

        /**
         * @brief Returns score of required headwear type.
         * @param [in] type headwear type.
         * @see HeadWearType for more info.
         * */
        inline float getScore(HeadWearType type) const;
    };
```

There are two groups of the fields:

1. The first group contains only the result enum:

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

2. The second group contains scores:

```
        float scores[static_cast<int>(HeadWearType::Count)]; //!< estimation
            scores
```

The **HeadWearEstimation structure** contains results of both Headwear state and type estimations:

```
    struct HeadWearEstimation {
        HeadWearStateEstimation state;  //!< headwear state estimation
                                        //!< (@see HeadWearStateEstimation)
        HeadWearTypeEstimation type;    //!< headwear type estimation
                                        //!< (@see HeadWearTypeEstimation)
    };
```

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.

Filtration parameters:

Table 29: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

Note. Detection size is detection width.

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

API structure name:

IHeadWearEstimator

Plan files:

- head_wear_v2_cpu.plan
- head_wear_v2_cpu-avx2.plan
- head_wear_v2_gpu.plan

6.12 Background Estimation

Name: BackgroundEstimator

Algorithm description:

This estimator is designed to estimate the background in the original image. It can return the following results:

- The background is non-solid (see the `BackgroundStatus::NonSolid` field in the BackgroundStatus enum);
- The background is solid (see the `BackgroundStatus::Solid` field in the BackgroundStatus enum);

Implementation description:

The estimator (see IBackgroundEstimator in IBackgroundEstimator.h):

- Implements an *estimate()* function that accepts R8G8B8 **source image**, detection and BackgroundEstimation structure to return estimation results;

- Implements an *estimate()* function that accepts `fsdk::Span` of R8G8B8 **source images**, `fsdk::Span` of detections, and `fsdk::Span` of BackgroundEstimation structures to return estimation results.

The **BackgroundStatus enumeration** contains all possible results of the Background estimation:

```
enum class BackgroundStatus : uint8_t {
    NonSolid = 0,     //!< NonSolid
    Solid = 1         //!< Solid
};
```

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

```
struct BackgroundEstimation {
    BackgroundStatus status;    //!< estimation result (@see
        BackgroundStatus enum).
    float backgroundScore;      //!< numerical value in range [0, 1],
        where 1 - is uniform background, 0 - is non uniform.
    float backgroundColorScore; //!< numerical value in range [0, 1],
        where 1 - is light background, 0 - is too dark.
};
```

There are two groups of the fields:

1. The first group contains the enum:

```
        BackgroundStatus status;    //!< estimation result (@see
            BackgroundStatus enum).
```

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

2. The second group contains scores:

```
        float backgroundScore;      //!< numerical value in range [0, 1],
            where 1 - is solid background, 0 - is non solid.
        float backgroundColorScore; //!< numerical value in range [0, 1],
            where 1 - is light background, 0 - is too dark.
```

The scores are defined in [0,1] range. If two scores are above the threshold, then the background is solid, otherwise the background is not solid.

Recommended thresholds:

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

Table 30: "Background estimator recommended thresholds"

Threshold	Recommended value
backgroundThreshold	0.5
backgroundColorThreshold	0.5

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements: The face in a frame should be large in relation to frame sizes. The face should occupy about half of the frame area.

```
  max(frameWidth, frameHeight) / max(faceWidth, faceHeight) <= 2.0
```

The type of preferable detector is FaceDetV3.

Table 31: "Requirements for `Detector`"

Attribute	Min face size
result	40

Table 32: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Maximum value
yaw	20.0
pitch	20.0
roll	20.0

Configurations:

See the "Background Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IBackgroundEstimator

Plan files:

- background_v2_cpu.plan
- background_v2_cpu-avx2.plan
- background_v2_gpu.plan

6.13 Grayscale, color or infrared Estimation

Name: BlackWhiteEstimator

Algorithm description:

BlackWhite estimator has two interfaces.

The "By full frame" interface detects if an input image is grayscale or color. It is indifferent to image content and dimensions; you can pass both face crops (including warped images) and full frames.

The "By warped frame" interface can be used only with warped images (see chapter "Image warping" for details). Checks if an image is color, grayscale or infrared.

Implementation description:

The "By full frame" interface of estimator (see ImageColorEstimation in IBlackWhiteEstimator.h):

* Implements *estimate()* function that accepts **source image** and outputs a boolean, indicating if the image is grayscale (true) or not (false).

The "By warped frame" interface of estimator (see IBlackWhiteEstimator in IBlackWhiteEstimator.h):

* Implements the *estimate()* function that accepts **warped source image**.

* Outputs ImageColorEstimation structures.

```
struct ImageColorEstimation {

    float colorScore;       //!< 0(grayscale)..1(color);
    float infraredScore;    //!< 0(infrared)..1(not infrared);

    /**
     * @brief Enumeration of possible image color types.
     * */
    enum class ImageColorType : uint8_t {
        Color = 0,      //!< image is color.
        Grayscale,      //!< Image is grayscale.
        Infrared,       //!< Image is infrared.
    };

    ImageColorType colorType;
};
```

`ImageColorEstimation::ImageColorType` presents color image type as enum with possible values: Color, Grayscale, Infrared.

```
- For color image score `colorScore` will be close to 1.0 and the second one
    `infraredScore` - to 0.0;
- for infrared image score `colorScore` will be close to 0.0 and the second
    one `infraredScore` - to 1.0;
- for grayscale images both of scores will be near 0.0.
```

Both interfaces use different principles of color type estimation.

BlackWhite estimator is trained to work with real warped photo of faces. We do not guarantee correctness when the people in the photo are fake (not real, such as the photo in the photo).

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `BlackWhiteEstimator` `::Settings` section. By default, these threshold values are set to optimal.

Table 33: "Black and white estimator recommended thresholds"

Threshold	Recommended value
colorThreshold	0.5
irThreshold	0.5

Configurations:

See the "BlackWhite Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IBlackWhiteEstimator

Plan files:

- black_white_and_ir_v1_cpu.plan
- black_white_and_ir_v1_cpu-avx2.plan
- black_white_and_ir_v1_gpu.plan

6.14 Face features extraction functionality

6.14.1 Eyes Estimation

Name: EyeEstimator

Algorithm description:

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

For this type of estimator can be defined sensor type.

This estimator aims to determine:

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

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

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

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

Implementation description:

The estimator:

- Implements the *estimate()* function that accepts **warped source image** and warped landmarks, either of type Landmarks5 or Landmarks68. The warped image and landmarks are received from the warper (see `IWarper::warp()`);

- Classifies eyes state and detects its iris and eyelid landmarks;

- Outputs EyesEstimation structures.

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

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

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

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

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

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

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

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

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

API structure name:

IEyeEstimator

Plan files:

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

6.14.2 Red Eyes Estimation

Name: RedEyeEstimator

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details) and warped landmarks.

Red Eye estimator evaluates whether a person's eyes are red in a photo or not.

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

Implementation description:

The estimator (see IRedEyeEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts **warped source image** in R8G8B8 format and warped Landmarks5. The warped image and landmarks are received from the warper (see `IWarper::warp()`);.

- Implements the *estimate()* function that accepts `fsdk::Span` of the **source warped images** in R8G8B8 format and `fsdk::Span` of warped Landmarks.

- Outputs RedEyeEstimation structure.

RedEyeEstimation structure consists of attributes for each eye. Eye attributes consists of a score of and status. Scores is normalized float value in a range of [0..1] where 1 is red eye and 0 is not.

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

```
struct RedEyeEstimation {
    /**
     * @brief Eyes attribute structure.
     * */
    struct RedEyeAttributes {
        RedEyeStatus status;    //!< Status of an eye.
        float score;            //!< Score, numerical value in range
            [0,1].
    };

    RedEyeAttributes leftEye;  //!< Left eye attributes
    RedEyeAttributes rightEye; //!< Right eye attributes
};
```

There are two groups of the fields in RedEyeAttributes:

1. The first field is a status:

```
        RedEyeStatus status;    //!< Status of an eye.
```

2. The second field is a score, which defined in [0,1] range:

```
        float score;         //!< Score, numerical value in range [0, 1].
```

Enumeration of possible red eye statuses.

```
    enum class RedEyeStatus : uint8_t {
        NonRed,      //!< Eye is not red.
        Red,         //!< Eye is red.
    };
```

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `RedEyeEstimator`
`::Settings` section. By default, this threshold value is set to optimal.

Table 34: "Red eye estimator recommended threshold"

Threshold	Recommended value
redEyeThreshold	0.5

Filtration parameters:

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

Table 35: "Requirements for `fsdk::NaturalLight`"

Attribute	Minimum value
score	0.5

Table 36: "Requirements for `fsdk::SubjectiveQuality`"

Attribute	Minimum value
blur	0.61

Attribute	Minimum value
light	0.57
darkness	0.5
illumination	0.1
specularity	0.1

Also `fsdk::GlassesEstimation` must not be equal to `fsdk::GlassesEstimation::SunGlasses`.

Configurations:

See the "RedEyeEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IRedEyeEstimator

Plan files:

- red_eye_v1_cpu.plan
- red_eye_v1_cpu-avx2.plan
- red_eye_v1_gpu.plan

6.14.3 Gaze Estimation

Name: GazeEstimator

Algorithm description:

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.

For this type of estimator can be defined sensor type.

Estimation characteristics:

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

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

Implementation description:

The **GazeEstimation structure** contains results of the estimation. Each angle is measured in degrees and in [-180, 180] range:

```
struct GazeEstimation {
    float yaw;      //!< Eye yaw angle.
    float pitch;    //!< Eye pitch angle.
};
```

Metrics:

Table below contains gaze prediction accuracy values.

Table 37: "Gaze prediction accuracy"

	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.

API structure name:

IGazeEstimator

Plan files:

- gaze_v2_cpu.plan
- gaze_v2_cpu-avx2.plan
- gaze_v2_gpu.plan
- gaze_ir_v2_cpu.plan
- gaze_ir_v2_cpu-avx2.plan
- gaze_ir_v2_gpu.plan

6.14.4 Glasses Estimation

Name: GlassesEstimator

Algorithm description:

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" for details). Quality of estimation depends on threshold values located in faceengine configuration file (see below).

Implementation description:

Enumeration of possible glasses estimation statuses:

```
enum class GlassesEstimation: uint8_t{
    NoGlasses,      //!< Person is not wearing glasses
    EyeGlasses,     //!< Person is wearing eyeglasses
    SunGlasses,     //!< Person is wearing sunglasses
    EstimationError //!< failed to estimate
};
```

Recommended thresholds:

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

Table 38: "Glasses estimator recommended thresholds"

Threshold	Recommended value
noGlassesThreshold	0.986
eyeGlassesThreshold	0.57
sunGlassesThreshold	0.506

Configurations:

See the "GlassesEstimator settings" section in the "ConfigurationGuide.pdf" document.

Metrics:

Table below contains true positive rates corresponding to selected false positive rates.

Table 39: "Glasses estimator TPR/FPR rates"

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

API structure name:

IGlassesEstimator

Plan files:

- glasses_estimation_flwr_cpu.plan
- glasses_estimation_flwr_cpu-avx2.plan
- glasses_estimation_flwr_gpu.plan

6.14.5 Overlap Estimation

Name: OverlapEstimator

Algorithm description:

This estimator tells whether the face is overlapped by any object. It returns a structure with value of overlapping and Boolean answer. 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.

Implementation description:

The estimator (see IOverlapEstimator in IOverlapEstimator.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.

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

```
struct OverlapEstimation {
    float overlapValue; //!< Numerical value of face overlapping in
        range [0, 1].
    bool overlapped;    //!< Overlapped face (true) or not (false).
};
```

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `OverlapEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 40: "Overlap estimator recommended threshold"

Threshold	Recommended value
overlapThreshold	0.01

Configurations:

See the "OverlapEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IOverlapEstimator

Plan files:

- overlap_estimation_flwr_cpu.plan
- overlap_estimation_flwr_cpu-avx2.plan
- overlap_estimation_flwr_gpu.plan

6.15 Emotion estimation functionality

6.15.1 Emotions Estimation

Name: EmotionsEstimator

Algorithm description:

> 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.

Implementation description:

The estimator (see IEmotionsEstimator in IEmotionsEstimator.h):

- Implements the *estimate()* function that accepts **warped source image**. 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.

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

```
struct EmotionsEstimation {
    float anger;    //!< 0(not angry)..1(angry);
    float disgust;  //!< 0(not disgusted)..1(disgusted);
    float fear;     //!< 0(no fear)..1(fear);
    float happiness;//!< 0(not happy)..1(happy);
    float sadness;  //!< 0(not sad)..1(sad);
    float surprise; //!< 0(not surprised)..1(surprised);
    float neutral;  //!< 0(not neutral)..1(neutral).

    enum Emotions {
```

```
        Anger = 0,
        Disgust,
        Fear,
        Happiness,
        Sadness,
        Surprise,
        Neutral,
        Count
    };

    /**
     * @brief Returns emotion with greatest score
     * */
    inline Emotions getPredominantEmotion() const;

    /**
     * @brief Returns score of required emotion
     * @param [in] emotion emotion
     * @see Emotions for details.
     * */
    inline float getEmotionScore(Emotions emotion) const;
};
```

API structure name:

IEmotionsEstimator

Plan files:

- emotion_recognition_v2_cpu.plan
- emotion_recognition_v2_cpu-avx2.plan
- emotion_recognition_v2_gpu.plan

6.16 Mouth Estimation Functionality

Name: MouthEstimator

Algorithm description:

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

Implementation description:

Mouth Estimation

It returns the following bool flags:

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

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

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

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

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

Mouth Estimation Extended

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

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

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

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

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

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

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

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

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

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

> These estimators are trained to work with warped images (see Chapter "Image warping" for details).

Recommended thresholds:

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

Table 41: "Mouth estimator recommended thresholds"

Threshold	Recommended value
occlusionThreshold	0.3
smileThreshold	0.55
openThreshold	0.64

Configurations:

See the "Mouth Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IMouthEstimator

Plan files:

- mouth_estimation_v4_cpu.plan
- mouth_estimation_v4_cpu-avx2.plan
- mouth_estimation_v4_gpu.plan

6.17 DeepFake estimation functionality

Name: DeepFakeEstimator

Algorithm description:

This estimator is designed to predict wether the face detected in the input image synthetic or not.

Important notes

The current implementation is experimental and does not support backward compatibility. The API can be modified in upcoming versions. Tests were carried out with images generated by technologies from the list below:

- Deepfacelab (Deepfacelive)
- FaceSwap
- Face2Face
- NeuralTextures
- FSGAN
- StyleGAN* (v1, v2)
- Roop (InsightFaceSwap)

Implementation description:

`DeepFakeEstimator` returns the following structure:

```
struct DeepFakeEstimation {
    enum class State {
        Real = 0,    //!< The person in image is real
        Fake         //!< The person in image is fake (media is synthetic)
    };

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

The estimation score normalized between 0.0 and 1.0, where 1.0 equals to 100% confidence that media is not synthetic(*real*), and 0.0 equals to 0% that the media is synthetic(*fake*).

Requirements for a detected face in the source image

- Minimum Face width is 150 pixels.
- Yaw angles should not exceed 30 degrees.
- Pitch angles should not exceed 20 degrees.

Recommended thresholds:

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

Table 42: "DeepFakeEstimator recommended settings"

Parameter	Description	Type	Default value
realThreshold	Threshold in [0..1] range.	"Value:: Float1"	0.5
defaultEstimatorType	Configuration of plan files usage.	Value::Int1	2

Possible values for `defaultEstimatorType`

Currently, available values for selecting estimation scenario are 1 and 2.

Scenario M1 means use of the first and second plan files.

Scenario M2 means use of all three plan files. At first, the estimator pre-estimates whether the detected face in the input image synthetic only with third plan file. Then, if the result is *fake*, the other two plan files will not run, and the estimator returns Estimation score - 0, and Liveness status - Fake. But, if the third plan file result is real, the estimator returns Estimation score, and Liveness status processes with the first and second plan files just like in the M1 scenario.

Other configurations of plan files usage are not provided.

Configurations:

See the "DeepFake Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IDeepFakeEstimator

API namespace:

fsdk::experimental::IDeepFakeEstimator

Plan files:

- deepfake_estimation_v1_model_1_cpu.plan
- deepfake_estimation_v1_model_1_cpu-avx2.plan
- deepfake_estimation_v1_model_1_gpu.plan
- deepfake_estimation_v1_model_2_cpu.plan
- deepfake_estimation_v1_model_2_cpu-avx2.plan
- deepfake_estimation_v1_model_2_gpu.plan
- deepfake_estimation_v1_model_3_cpu.plan

- deepfake_estimation_v1_model_3_cpu-avx2.plan

- deepfake_estimation_v1_model_3_gpu.plan

6.18 Liveness check functionality

6.18.1 HeadAndShouldersLiveness Estimation

Name: HeadAndShouldersLivenessEstimator

Algorithm description:

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.).

Implementation description:

The estimator (see IHeadAndShouldersLiveness in IHeadAndShouldersLiveness.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 "Face detection facility"). Estimates whether real person or not. Outputs float score normalized in range [0..1], 1 - is real person, 0 - is fake.

Recommended thresholds:

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

Table 43: "HeadAndShouldersLiveness estimator recommended thresholds"

Threshold	Recommended value
headWidthKoeff	1.0
headHeightKoeff	1.0
shouldersWidthKoeff	0.75
shouldersHeightKoeff	3.0

Configurations:

See the "HeadAndShouldersLivenessEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IHeadAndShouldersLivenessEstimator

Plan files:

- hs_shoulders_liveness_estimation_flwr_cpu.plan
- hs_head_liveness_estimation_flwr_cpu.plan
- hs_shoulders_liveness_estimation_flwr_cpu-avx2.plan
- hs_head_liveness_estimation_flwr_cpu-avx2.plan
- hs_shoulders_liveness_estimation_flwr_gpu.plan
- hs_head_liveness_estimation_flwr_gpu.plan

6.18.2 LivenessFlyingFaces Estimation

Name: LivenessFlyingFacesEstimator

Algorithm description:

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

Implementation description:

The estimator (see ILivenessFlyingFacesEstimator in ILivenessFlyingFacesEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Image` with valid image in R8G8B8 format and `fsdk::Detection` of corresponding **source image** (see section "Detection structure" in chapter "Face detection facility").

- Implements the *estimate()* function that needs the span of `fsdk::Image` with valid **source images** in R8G8B8 formats and span of `fsdk::Detection` of corresponding source images (see section "Detection structure" in chapter "Face detection facility").

Those methods estimate 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.

> The estimator is trained to work in combination with `fsdk::ILivenessRGBMEstimator`.

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

```
struct LivenessFlyingFacesEstimation {
    float score;    //!< Numerical value in range [0, 1].
    bool isReal;    //!< Is real face (true) or not (false).
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in LivenessFlyingFacesEstimator section. By default, these threshold values are set to optimal.

Table 44: "Mouth estimator recommended thresholds"

Threshold	Recommended value
realThreshold	0.5
aggregationCoeff	0.7

Filtration parameters:

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

Table 45: "Requirements for `fsdk::BestShotQualityEstimator::EstimationResult`"

Attribute	Acceptable values
headPose.pitch	[-30...30]
headPose.yaw	[-30...30]
headPose.roll	[-40...40]
ags	[0.5...1.0]

Table 46: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

> Detection size is detection width.

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

Configurations:

See the "LivenessFlyingFaces Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessFlyingFacesEstimator

Plan files:

- flying_faces_liveness_v4_cpu.plan
- flying_faces_liveness_v4_cpu-avx2.plan
- flying_faces_liveness_v4_gpu.plan

6.18.3 LivenessRGBM Estimation

Name: LivenessRGBMEstimator

Algorithm description:

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

Implementation description:

The estimator (see ILivenessRGBMEstimator in ILivenessRGBMEstimator.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 "Face 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.

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

```
struct LivenessRGBMEstimation {
    float score = 0.0f; //!< Estimation score
    bool isReal = false;//!< Where person is real or not
};
```

Recommended thresholds:

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

Table 47: "LivenessRGBM estimator recommended thresholds"

Threshold	Recommended value
backgroundCount	100
threshold	0.8
coeff1	0.222
coeff2	0.222

Configurations:

See the "LivenessRGBM Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessRGBMEstimator

Plan files:

- rgbm_liveness_cpu.plan
- rgbm_liveness_cpu-avx2.plan
- rgbm_liveness_gpu.plan

6.18.4 Depth Liveness Estimation

Name: LivenessDepthEstimator

Algorithm description:

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

Implementation description:

The estimator (see ILivenessDepthEstimator in ILivenessDepthEstimator.h):

- Implements the *estimate()* function that accepts **source warped image** (see chapter "Image warping" for details) in R16 format and `fsdk::DepthEstimation` structure. This method estimates whether or not depth map corresponds to the real person. Corresponding estimation output with float score which is normalized in range [0..1], where 1 - is real person, 0 - is fake.

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

```
struct DepthEstimation {
    float score; //!< confidence score in [0,1] range. The closer the
        score to 1, the more likely that person is alive.
    bool isReal; //!< boolean flag that indicates whether a person is
        real.
};
```

Recommended thresholds:

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

Table 48: "Depth estimator recommended thresholds"

Threshold	Recommended value
maxDepthThreshold	3000
minDepthThreshold	100
zeroDepthThreshold	0.66
confidenceThreshold	0.89

Filtration parameters:

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

Table 49: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-15…15]
yaw	[-15…15]
roll	[-10…10]

Table 50: "Requirements for `fsdk::Quality`"

Attribute	Minimum value
blur	0.94
light	0.90
dark	0.93

Table 51: "Requirements for `fsdk::EyesEstimation`"

Attribute	State
leftEye	Open
rightEye	Open

Also, the minimum distance between the face bounding box and the frame borders should be greater than 20 pixels.

Configurations:

See the "Depth Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessDepthEstimator

Plan files:

- depth_estimation_v2_1_cpu.plan
- depth_estimation_v2_1_cpu-avx2.plan
- depth_estimation_v2_1_gpu.plan

6.18.5 Depth and RGB OneShotLiveness estimation

Name: LivenessDepthRGBEstimator

Algorithm description:

This estimator shows whether the person's face is real or fake (photo, printed image). You can use this estimator in payment terminals (POS) and self-service cash registers (KCO) with two cameras - Depth and RGB.

The estimation is performed on the device with an Orbbec camera. The camera can be either built in a POS or KCO device or connected to it. This allows to perform the estimation at a higher speed and makes it more secure as data is not sent to the backend. Using the algorithm with Orbbec cameras lets you work with deep data. It increases system reliability and accuracy, as 3D data lets you assess facial shapes and detect fake masks more accurately.

The estimator is trained to work with warped images. For details, see chapter "Image warping".

Supported devices

The estimator works only on the following devices:

- VLS LUNA CAMERA 3D
- VLS LUNA CAMERA 3D Embedded

Different models of Orbbec cameras have different spacing between sensors. If you need to use another Orbbec Depth+RGB camera, you can change the calibration coefficients to match the device. Please, contact VisionLabs for details.

Image requirements

This estimator works based on two images:

- RGB image from the RGB camera
- Depth image (or depth map) from the depth camera

Input images must meet the following requirements:

Parameter	Requirements
Resolution	640 × 480 pixels
Compression	No
Image cropping	No
Image rotation	No
Effects overlay	No
Number of faces in the frame	1

Parameter	Requirements
Face detection bounding box size	200 pixels
Frame edges offset	10 pixels
Head pose	-20 to +20 degrees for head pitch, yaw, and roll.
Image quality	The face in the frame should not be overexposed, underexposed, or blurred. For details, see section "Image Quality Estimation".

Implementation description:

The estimator implements the following:

- The `estimate()` function that needs the depth frame as the first `fsdk::Image` object, the RGB frame as the second `fsdk::Image` object, `fsdk::Detection` and `fsdk::Landmarks5` objects (see section "Detection structure" in chapter "Face detection facility"). The estimation output is the `fsdk::DepthRGBEstimation` srtucture.
- The `estimate()` function that needs the first span of depth frames as the `fsdk::Image` objects, the second span of RGB frames as the `fsdk::Image` objects, a span of `fsdk::Detection`, and a span of `fsdk::Landmarks5` (see section "Detection structure" in chapter "Face detection facility").

  The estimation output is a span of the `fsdk::DepthRGBEstimation` structure. The second output value is the `fsdk::DepthRGBEstimation`structure.

DepthRGBEstimation

The `DepthRGBEstimation` structure contains results of the estimation:

```
struct DepthRGBEstimation {
    //!< confidence score in [0,1] range.
    //!< The closer the score to 1, the more likely that person is alive.
    float score;
    //!< boolean flag that indicates whether a person is real.
    bool isReal;
};
```

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

The value of `isReal` depends on `score` and `confidenceThreshold`. The value of the `confidenceThreshold` can be changed in configuration file *faceengine.conf* (see *ConfigurationGuide* `LivenessDepthRGBEstimator`).

API structure name:

`ILivenessDepthRGBEstimator`

See `ILivenessDepthRGBEstimator` in `ILivenessDepthRGBEstimator.h`.

Plan files:

- depth_rgb_v2_model_1_cpu.plan
- depth_rgb_v2_model_1_gpu.plan
- depth_rgb_v2_model_2_cpu.plan
- depth_rgb_v2_model_2_gpu.plan
- depth_rgb_v2_model_1_cpu-avx2.plan
- depth_rgb_v2_model_2_cpu-avx2.plan

6.18.6 LivenessOneShotRGB Estimation

Name: LivenessOneShotRGBEstimator

Algorithm description:

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

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

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

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

See image quality thresholds in the "Image Quality Estimation" section.

Implementation description:

The estimator (see ILivenessOneShotRGBEstimator in ILivenessOneShotRGBEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Image`, `fsdk::Detection` and `fsdk::Landmarks5` objects (see section "Detection structure" in chapter "Face detection facility"). Output estimation is a structure `fsdk::LivenessOneShotRGBEstimation`.

- Implements the *estimate()* function that needs the span of `fsdk::Image`, span of `fsdk::Detection` and span of `fsdk::Landmarks5` (see section "Detection structure" in chapter "Face detection facility").
  The first output estimation is a span of structure `fsdk::LivenessOneShotRGBEstimation`. The second output value (structure `fsdk::LivenessOneShotRGBEstimation`) is the result of aggregation based on span of estimations announced above. Pay attention the second output value (aggregation) is optional, i.e. `default argument`, which is `nullptr`.

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

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

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

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

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

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

The reference score is 0,5.

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

Recommended thresholds:

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

set to optimal.

Table 54: "LivenessOneShotRGB estimator recommended thresholds"

Threshold	Recommended value
realThreshold	0.5
qualityThreshold	0.5
calibrationCoeff	0.8

Configurations:

See the "LivenessOneShotRGBEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessOneShotRGBEstimator

Plan files:

- oneshot_rgb_liveness_v6_model_1_cpu.plan
- oneshot_rgb_liveness_v6_model_2_cpu.plan
- oneshot_rgb_liveness_v6_model_3_cpu.plan
- oneshot_rgb_liveness_v6_model_4_cpu.plan
- oneshot_rgb_liveness_v6_model_1_cpu-avx2.plan
- oneshot_rgb_liveness_v6_model_2_cpu-avx2.plan
- oneshot_rgb_liveness_v6_model_3_cpu-avx2.plan
- oneshot_rgb_liveness_v6_model_4_cpu-avx2.plan
- oneshot_rgb_liveness_v6_model_1_gpu.plan
- oneshot_rgb_liveness_v6_model_2_gpu.plan
- oneshot_rgb_liveness_v6_model_3_gpu.plan
- oneshot_rgb_liveness_v6_model_4_gpu.plan

6.18.6.1 Usage example

The face in the image and the image itself should meet the estimator requirements.

You can find additional information in example (`examples/example_estimation/main.cpp`) or in the code below.

```
// Minimum detection size in pixels.
constexpr int minDetSize = 200;
```

```cpp
    // Step back from the borders.
    constexpr int borderDistance = 10;

    if (std::min(detectionRect.width, detectionRect.height) < minDetSize) {
        std::cerr << "Bounding Box width and/or height is less than `minDetSize`
            - " << minDetSize << std::endl;
        return false;
    }

    if ((detectionRect.x + detectionRect.width) > (image.getWidth() -
        borderDistance) || detectionRect.x < borderDistance) {
        std::cerr << "Bounding Box width is out of border distance - " <<
            borderDistance << std::endl;
        return false;
    }

    if ((detectionRect.y + detectionRect.height) > (image.getHeight() -
        borderDistance) || detectionRect.y < borderDistance) {
        std::cerr << "Bounding Box height is out of border distance - " <<
            borderDistance << std::endl;
        return false;
    }

    // Yaw, pitch and roll.
    constexpr int principalAxes = 20;

    if (std::abs(headPose.pitch) > principalAxes ||
        std::abs(headPose.yaw) > principalAxes ||
        std::abs(headPose.roll) > principalAxes ) {

        std::cerr << "Can't estimate LivenessOneShotRGBEstimation. " <<
            "Yaw, pith or roll absolute value is larger than expected value: "
                << principalAxes << "." <<
            "\nPitch angle estimation: " << headPose.pitch <<
            "\nYaw angle estimation: " << headPose.yaw <<
            "\nRoll angle estimation: " << headPose.roll << std::endl;
        return false;
    }
```

> We recommend using `Detector type 3 (fsdk::ObjectDetectorClassType::FACE_DET_V3`
> `)`.

6.19 Personal Protection Equipment Estimation

Name: PPEEstimator

Algorithm description:

The Personal Protection Equipment (a.k.a PPE) estimator predicts wether a person is wearing one or multiple types of protection equipment such as: - Helmet; - Hood; - Vest; - Gloves.

For each one of this attributes estimator returns 3 prediction scores which indicate the possibility of person wearing that attribute, not wearing it and an "unknown" score which will be the highest of them all if the estimator wasn't able to tell wether person on the image wears/doesn't wear a particular attribute.

Implementation description:

The **Personal Protection Equipment Estimation structure** for each attribute looks as follows:

```
struct OnePPEEstimation {
    float positive = 0.0f;
    float negative = 0.0f;
    float unknown  = 0.0f;

    enum class PPEState : uint8_t {
        Positive, //!< person is wearing specific personal equipment;
        Negative, //!< person isn't wearing specific personal equipment;
        Unknown,  //!< it's hard to tell wether person wears specific
            personal equipment.
        Count     //!< state count
    };

    /**
     * @brief returns predominant personal equipment state
     * */
    inline PPEState getPredominantState();
};
```

> All three prediction scores sum up to 1.

> Estimator takes as input an image and a human bounding box of a person for which attributes shall be predicted. For more information about human detector see "Human Detection" section.

API structure name:

IPPEEstimator

Plan files:

- ppe_estimation_v1_cpu.plan

- ppe_estimation_v1_cpu-avx2.plan
- ppe_estimation_v1_gpu.plan

6.20 Medical Mask Estimation Functionality

Name: MedicalMaskEstimator

This estimator aims to detect a medical mask on the face in the source image. For the interface with MedicalMaskEstimation 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);

For the interface with MedicalMaskEstimationExtended it can return the next results:

- A medical mask is on the face (see MedicalMaskExtended::Mask field in the MedicalMaskExtended enum);
- There is no medical mask on the face (see MedicalMaskExtended::NoMask field in the MedicalMaskExtended enum);
- A medical mask is not on the right place (see MedicalMaskExtended::MaskNotInPlace field in the MedicalMaskExtended enum);
- The face is occluded with something (see MedicalMaskExtended::OccludedFace field in the MedicalMaskExtended enum);

The estimator (see IMedicalMaskEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts source warped image in R8G8B8 format and medical mask estimation structure to return results of estimation;
- Implements the *estimate()* function that accepts source image in R8G8B8 format, face detection to estimate and medical mask estimation 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 medical mask estimation 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 medical mask estimation structures to return results of the estimation.

Every method can be used with MedicalMaskEstimation and MedicalMaskEstimationExtended.

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.

6.20.1 MedicalMaskEstimator thresholds

The estimator returns several scores, one for each possible result. The final result is based on that scores and thresholds. If some score is above the corresponding threshold, that result is estimated as final. If none of the scores exceed the matching threshold, the maximum value will be taken. If some of the scores exceed their thresholds, the results will take precedence in the following order for the case with MedicalMaskEstimation:

```
Mask, NoMask, OccludedFace
```

and for the case with MedicalMaskEstimationExtended:

```
Mask, NoMask, MaskNotInPlace, OccludedFace
```

The default values for all thresholds are taken from the configuration file. See Configuration guide for details.

6.20.2 MedicalMask enumeration

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

```cpp
enum class MedicalMask {
    Mask = 0,                      //!< medical mask is on the face
    NoMask,                        //!< no medical mask on the face
    OccludedFace                   //!< face is occluded by something
};

enum class DetailedMaskType {
    CorrectMask = 0,               //!< correct mask on the face (mouth
        and nose are covered correctly)
    MouthCoveredWithMask,          //!< mask covers only a mouth
    ClearFace,                     //!< clear face - no mask on the face
    ClearFaceWithMaskUnderChin,    //!< clear face with a mask around of
        a chin, mask does not cover anything in the face region (from
        mouth to eyes)
    PartlyCoveredFace,             //!< face is covered with not a
        medical mask or a full mask
    FullMask,                      //!< face is covered with a full mask
        (such as balaclava, sky mask, etc.)
    Count
};
```

- Mask is according to `CorrectMask` or `MouthCoveredWithMask`;
- NoMask is according to `ClearFace` or `ClearFaceWithMaskUnderChin`;
- OccludedFace is according to `PartlyCoveredFace` or `FullMask`.

Note - `NoMask` means absence of medical mask or any occlusion in the face region (from mouth to eyes).

Note - `DetailedMaskType` is not supported for NPU-based platforms.

6.20.3 MedicalMaskEstimation structure

The `MedicalMaskEstimation` structure contains results of the estimation:

```
struct MedicalMaskEstimation {
    MedicalMask result;          //!< estimation result (@see
        MedicalMask enum)
    DetailedMaskType maskType;   //!< detailed type  (@see
        DetailedMaskType 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

    float scores[static_cast<int>(DetailedMaskType::Count)]{};    //!<
        detailed estimation scores

    inline float getScore(DetailedMaskType type) const;
};
```

There are two groups of the fields:

1. The first group contains the result:

```
MedicalMask result;
```

Result enum field MedicalMaskEstimation contains the target results of the estimation. Also you can see the more detailed type in MedicalMaskEstimation.

```
DetailedMaskType maskType;            //!< detailed type
```

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 score 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. More detailed scores for every type of a detailed type of face covering are

```
float scores[static_cast<int>(DetailedMaskType::Count)]{};    //!< detailed
    estimation scores
```

- maskScore is the sum of scores for CorrectMask, MouthCoveredWithMask;
- NoMask is the sum of scores for ClearFace and ClearFaceWithMaskUnderChin;
- occludedFaceScore is the sum of scores for PartlyCoveredFace and FullMask fields.

Note - DetailedMaskType, scores, getScore are not supported for NPU-based platforms. It means a user cannot use this fields and methods in code.

6.20.4 MedicalMaskExtended enumeration

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

```
enum class MedicalMaskExtended {
    Mask = 0,                   //!< medical mask is on the face
    NoMask,                     //!< no medical mask on the face
    MaskNotInPlace,             //!< mask is not on the right place
    OccludedFace                //!< face is occluded by something
};
```

6.20.5 MedicalMaskEstimationExtended structure

The MedicalMaskEstimationExtended structure contains results of the estimation:

```
struct MedicalMaskEstimationExtended {
    MedicalMaskExtended result;     //!< estimation result (@see
        MedicalMaskExtended enum)
    // scores
    float maskScore;          //!< medical mask is on the face score
    float noMaskScore;        //!< no medical mask on the face score
    float maskNotInPlace;     //!< mask is not on the right place
    float occludedFaceScore;  //!< face is occluded by something score
```

```
        };
```

There are two groups of the fields:

1. The first group contains only the result enum:

```
        MedicalMaskExtended result;
```

Result enum field MedicalMaskEstimationExtended contains 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 maskNotInPlace;   //!< mask is not on the right place
        float occludedFaceScore; //!< face is occluded by something score
```

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

6.20.6 Filtration parameters

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

Table 55: "Requirements for `fsdk::MedicalMaskEstimator::EstimationResult`"

Attribute	Acceptable values
headPose.pitch	[-40…40]
headPose.yaw	[-40…40]
headPose.roll	[-40…40]
ags	[0.5…1.0]

Configurations:

See the "Medical mask estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IMedicalMaskEstimator

Plan files:

- mask_clf_v3_cpu.plan

- mask_clf_v3_cpu-avx2.plan
- mask_clf_v3_gpu.plan

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).

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 across cameras network;
- tracking of visitors across cameras network;
- search for a person across the cameras network in case when face was not captured (e.g. across CCTV cameras in the city);
- 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.

The human descriptor is based on to the following criteria:

- clothes (type and color);
- shoes;
- accessories;
- hairstyle;
- body type;
- anthropometric parameters of the body.

Note. The human reidentification algorithm is trained to work with input data that meets the following requirements:

- input images should be in R8G8B8 format (will work worse in night mode);
- the smaller side of input crop should be greater than 60 px;
- inside of same crop, one person should occupy more than 80% (sometimes several persons fit into the same frame).

7.2 Descriptor

Descriptor object stores a compact set of packed properties as well as some helper parameters that were used to extract these properties from the source image. Together these parameters determine descriptor compatibility. Not all descriptors are compatible with each other. It is impossible to batch and match incompatible descriptors, so you should pay attention to what settings do you use when extracting them. Refer to section "Descriptor extraction" for more information on descriptor extraction.

7.2.1 Descriptor Versions

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

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

See chapter "Appendix A. Specifications" for details about performance and precision of different descriptor versions.

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

Descriptor version may be specified in the configuration file (see section "Configuration data" in chapter "Core facility").

7.2.1.1 Face descriptor

Currently next versions are available: 58, 59, 60, 62. Descriptors have **backend** and **mobilenet** implementations. Versions 58, 62 supports only **backend** implementation. Backend versions more precise, but mobilenet faster and have smaller model files. See Appendix A.1 and A.2 for details about performance and precision of different descriptor versions.

7.2.1.2 Human descriptor

Versions of human descriptors are available: 102, 103, 104, 105, 106, 107, 108, 109, 110, 112, 113

> Versions 102, 103, 104 are deprecated.

To create a human descriptor, human batch, human descriptor extractor, human descriptor matcher you must pass the human descriptor version

- DV_MIN_HUMAN_DESCRIPTOR_VERSION = 102 or
- HDV_TRACKER_HUMAN_DESCRIPTOR_VERSION = 102, //!< Deprecated. human descriptor for tracking of people on one camera, light and fast version

- HDV_PRECISE_HUMAN_DESCRIPTOR_VERSION = 103, //!< Deprecated. precise human descriptor, heavy and slow
- HDV_REGULAR_HUMAN_DESCRIPTOR_VERSION = 104, //!< Deprecated. regular human descriptor, use it by default for multi-cameras tracking
- HDV_TRACKER_V2 = 105, //!< human descriptor for tracking of people, light and fast version.
- HDV_PRECISE_V2 = 106, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V2 = 107, //!< regular human descriptor.
- HDV_TRACKER_V3 = 108, //!< human descriptor for tracking of people, light and fast version.
- HDV_PRECISE_V3 = 109, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V3 = 110, //!< regular human descriptor.
- HDV_PRECISE_V4 = 112, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V4 = 113 //!< regular human descriptor.

7.2.2 Descriptor Batch

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

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

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

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

The following notes should be kept in mind:

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

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

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

Memory deallocation takes place when a batch is released.

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

7.2.3 Descriptor Extraction

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

- a face detection area inside the image (see chapter "Detection facility")
- a pre-allocated descriptor (see section "Descriptor")
- a pre-computed landmarks (see chapter "Image warping")

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

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

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

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

Descriptor extraction implementation supports execution on GPUs.

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

Also *IDescriptorExtractor* returns *descriptor score* for each extracted descriptor. Descriptor score is

normalized value in range [0,1], where 1 - face in the warp, 0 - no face in the warp. This value allows you filter descriptors extracted from false positive detections.

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

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

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

7.2.4 Descriptor Matching

It is possible to match a pair (or more) previously extracted descriptors to find out their similarity. With this information, it is possible to implement face search and other analysis applications.
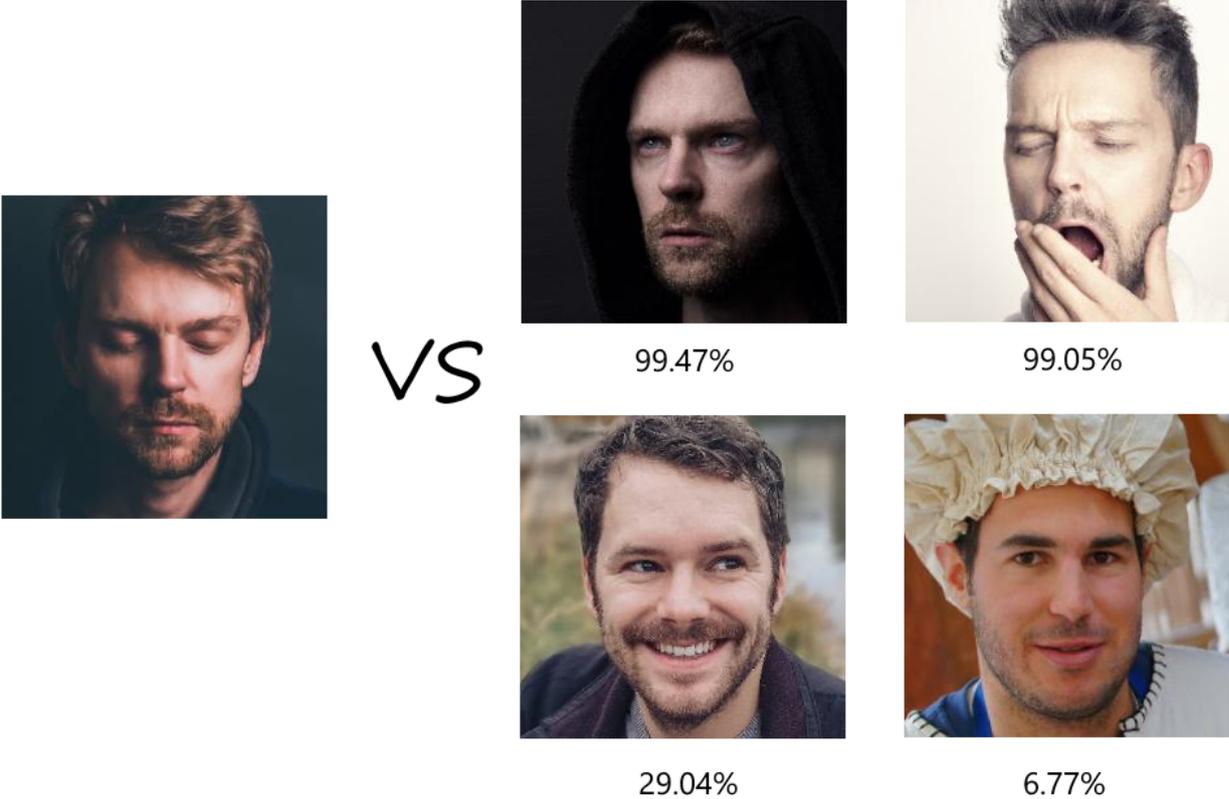
Figure 16: Matching

By means of *match* function defined by the *IDescriptorMatcher* interface it is possible to match a pair of

descriptors with each other or a single descriptor with a descriptor batch (see section "Descriptor batch" for details on batches).

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

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

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

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

8 System Requirements

8.1 Android installations

FaceEngine requires:

- Android version 4.4.4 or newer.

For development:

- Android Gradle Plugin Version - 3.2.1;
- Gradle Version - 4.6;
- Java 8.

x86

- Android NDK = android-16
- Android SDK = 16
- Android STL = c++_shared
- Compiler = Clang 12.0.5

x86_64

- Android NDK = android-21
- Android API = 21
- Android STL = c++_shared
- Compiler = Clang 12.0.5

armeabi-v7a

- Android NDK = android-16
- Android API = 16
- Android STL = c++_shared
- Compiler = Clang 12.0.5

arm64-v8a

- Android NDK = android-21
- Android API = 21
- Android STL = c++_shared
- Compiler = Clang 12.0.5

> 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 Hardware requirements

9.1 Embedded installations

9.1.1 CPU requirements

Supported CPU architectures:

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

9.2 Android for embedded

One more step to online activation process, in addition to information about LUNA SDK licensing, described in **VisionLabs LUNA SDK Licensing**, paragraph **License activation**.

Besides the common steps for online-activation, described in document **VisionLabs LUNA SDK Licensing**, for **Android for embedded** systems, execute a native licensed binary for **Android for embedded** with **root permissions** at least once.

10 Migration guide

10.1 Overview

Here you can find information about important changes in the next releases of LUNA SDK.

10.2 v.5.17.0

10.2.1 IHeadAndShouldersLivenessEstimator

Since v.5.17.0 the estimator `IHeadAndShouldersLivenessEstimator` has been deprecated (See `IHeadAndShouldersLivenessEstimator.h`). If you need this estimator, please, contact VisionLabs for details.

10.2.2 IChildEstimator

Since v.5.17.0 the estimator `IChildEstimator` has been deprecated (See `IChildEstimator.h`). Use the `IAttributeEstimator` (See `IAttributeEstimator.h`) with `IAttributeEstimator::EstimationRequest::estimateAge` instead.

Example of code (before version v.5.17.0):

```
// Create child estimator.
auto resChildEstimator = faceEngine->createChildEstimator();
if(!resChildEstimator) {
    std::cerr << "Failed to create child estimator instance. Reason: "
        << resChildEstimator.what();
    std::cerr << std::endl;
    return -1;
}
fsdk::IChildEstimatorPtr childEstimator = resChildEstimator.getValue();

// Get child estimation.
fsdk::ChildEstimation childEstimation;
fsdk::Result<fsdk::FSDKError> childEstimationResult = childEstimator->
    estimate(warp, childEstimation);
if(childEstimationResult.isOk()) {
    std::cout << "\nChild estimate:";
    std::cout << "\nchildScore: " << childEstimation.childScore << " (
        range [0, 1])";
    std::cout << "\nis child: " << childEstimation.isChild << " (1 - is
        child, 0 - is adult)";
    std::cout << std::endl;
} else {
```

```
        std::cerr << "Failed child estimation. Reason: " <<
            childEstimationResult.what() << std::endl;
    }
```

Example of code (from version v.5.17.0):

```
    // Create attribute estimator.
    auto resAttributeEstimator = faceEngine->createAttributeEstimator();
    if(!resAttributeEstimator) {
        std::cerr << "Failed to create attribute estimator instance. Reason:
            " << resAttributeEstimator.what();
        std::cerr << std::endl;
        return -1;
    }
    fsdk::IAttributeEstimatorPtr attributeEstimator = resAttributeEstimator.
        getValue();

    // Get attribute estimation.
    using AttrsRequest = fsdk::IAttributeEstimator::EstimationRequest;
    AttrsRequest attributesRequest = AttrsRequest::estimateAge;
    fsdk::IAttributeEstimator::EstimationResult attributeEstimation;

    fsdk::Result<fsdk::FSDKError> attributeEstimatorResult =
        attributeEstimator->estimate(warp, attributesRequest,
            attributeEstimation);

    if(attributeEstimatorResult.isOk()) {
        std::cout << "\nAttribute estimate:";
        std::cout << "\nage: " << attributeEstimation.age.value() << " (in
            years)" << std::endl;
        std::cout << std::endl;
    } else {
        std::cerr << "Failed to make attribute estimation. Reason: " <<
            attributeEstimatorResult.what();
        std::cerr << std::endl;
    }
```

10.2.3 Index

Since v.5.17.0 IDynamicIndex can be saved as a file to hard disc after removing of descriptors.

Example of code (before version v.5.17.0):

```
    // index with removed descriptors could not be saved
    for (std::size_t i = 0; i < nRemoved; ++i) {
        auto resRemove = index->removeDescriptor(i);
        if (resRemove.isError()) {
            // process error
            ...
        }
    }
```

Example of code (from version v.5.17.0):

```
    // remove descriptors
    for (std::size_t i = 0; i < nRemoved; ++i) {
        auto resRemove = index->removeDescriptor(i);
        if (resRemove.isError()) {
            // process error
            ...
        }
    }

    // erase descriptors
    auto resEraseRemovedDescs = index->eraseRemovedDescriptors();
    if(resEraseRemovedDescs.isError()) {
        // process error
        ...
    }

    // get map of new descriptors
    auto map = resEraseRemovedDescs.getValue();
    for (std::size_t i = 0; i < count; ++i) {
        // if the old ID is not found, the error InvalidDescriptorId will be
            returned
        auto resMapFind = map->getId(i);
            if (resMapFind.isError()) {
                // process error or skip not found id
                ...
            }
            // we can get new id by old id
            auto newId = resMapFind.getValue();

    }
```

```
    // now we can save index
    auto resSave = index->saveToDynamicIndex("your_index_name.bin");
    if (resSave.isError()) {
        // process error
    }
    // get status of index for search DIS_VALID, DIS_INVALID and etc.
    auto status = resSave.getValue();
```

10.2.4 FishEyeEstimator

Since v.5.13.0 method `estimate` of `IFishEyeEstimator` by crop and detection has been deprecated (See `IFishEyeEstimator.h`). Use `estimate` by warped image instead.

Example of code (before version 5.13.0):

```
    fsdk::FishEyeEstimation estimation;
    fsdk::Result<fsdk::FSDKError> res = fishEyeEstimator->estimate(image,
        detection, estimation);
```

Example of code (from version 5.13.0):

```
    fsdk::FishEyeEstimation estimation;
    fsdk::Result<fsdk::FSDKError> res = fishEyeEstimator->estimate(warp,
        estimation);
```

10.3 v.5.6.0

10.3.1 Vector2

Since v.5.6.0, the member `array` in `fsdk::Vector2` has been removed. You should use the `x` and `y` members instead of the removed `array` one.

Example of code (before version 5.6.0):

```
    fsdk::Vector2<int> vector2;
    vector2.x = 10;
    vector2.y = 20;
    // or
    vector2.array[0] = 10;
    vector2.array[1] = 20;
```

Example of code (from version 5.6.0):

```
        fsdk::Vector2<int> vector2;
        vector2.x = 10;
        vector2.y = 20;
```

10.3.2 BlackWhiteEstimator

Since v.5.6.0 method `estimate` of `IBlackWhiteEstimator` by full image has been deprecated (See `IBlackWhiteEstimator.h`). Use `estimate` by warped image instead.

Example of code (before version 5.6.0):

```
        bool isGray = false;
        Result<FSDKError> res = BlackWhiteEstimator->estimate(fullImage, isGray)
            ;
```

Example of code (from version 5.6.0):

```
        fsdk::ImageColorEstimation estimation;
        Result<FSDKError> res = BlackWhiteEstimator->estimate(warp, estimation);
```

10.4 v.5.5.0

From v.5.5.0 the default value of `numThreads` (runtime.conf) was replaced by `-1`. Which means that will be taken the maximum number of available threads. This number of threads is equal to according number of available processor cores.

Example of setting (before version 5.5.0):

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

Example of setting (from version 5.5.0):

```
  <param name="numThreads" type="Value::Int1" x="-1" />
```

From v.5.5.0 the method `loadFromFile(const char* path)` (See `ILicense.h`) is deprecated. The use is allowed, but can be useless. Please use the method `loadFromFile(const char* path, const fsdk::ISettingsProvider* settings)` instead.

10.4.0.1 Examples of code

Example of code (before version 5.5.0):

```
const bool isLicenseFileLoadedSuccessfully = license->loadFromFile(path)
    .isOk());
```

Example of code (from version 5.5.0):

```
auto resSettings = fsdk::createSettingsProvider("License Config Path");
if (!resSettings.isOk()) {
    return -1;
}

fsdk::ISettingsProviderPtr settings = resSettings.getValue();

// Create new license from file
const bool isLicenseFileLoadedSuccessfully = license->loadFromFile(path,
    settings).isOk());
```

10.5 v.5.2.0

From v.5.2.0 the 101 version of human descriptor is not supported, it was changed by 104. Currently, three versions are available: 102 (tracker), 103 (precise), 104 (regular). It means that all instances (such as IDescriptorExtractor, IDescriptorMatcher and etc.) cannot be created with the version 101.

10.6 v.5.1.0

From version v.5.1.0 IHeadPoseEstimatorPtr and IAGSEstimatorPtr are deprecated. Use IBestShotQualityEstimatorPtr instead.

Note. AGS score thresholds are different for IAGSEstimatorPtr and IBestShotQualityEstimatorPtr. Read more on the BestShotQuality estimation page.

10.7 v.5.0.0

10.7.1 Objects creation

The fsdk::acquire(...) method for the pointer acquiring for IFaceEngine objects is not allowed for usage starting from version 5.0.0. In addition, the types of values returned from the create methods of IFaceEngine were changed.

Most of the `create` methods now return the following structure - `fsdk::ResultValue<fsdk::FSDKError, ObjectClassPtr>` Thus it is easy to check the correctness of the result (using one of the following methods `result.isOk()` or `result.isError()`) and get an error (using the `result.getError()` method). The `result.what()` method can be used to get the text description of the error.

10.7.1.1 Examples of code

Example of code (before version 5.0.0):

```
fsdk::IAttributeEstimatorPtr estimator = fsdk::acquire(faceEngine->
    createAttributeEstimator());
if (estimator.isNull())  {
    std::cout << "Object pointer is nullptr" << std::endl;
    ... // process error
}
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::IAttributeEstimatorPtr>
    resEstimator = faceEngine->createAttributeEstimator();
if (resEstimator.isError())  {
    std::cout << "Error: " << resEstimator.what() << std::endl;
    ... // process error
}

fsdk::IAttributeEstimatorPtr estimator = resEstimator.getValue();
```

10.7.2 Interface of ILicense

From version v.5.0.0 we changed the interface of `ILicense`. Now all methods of this class return `fsdk::Result<fsdk::FSDKError>`, `fsdk::ResultValue<fsdk::FSDKError, bool>` or `fsdk::ResultValue<fsdk::FSDKError, uint32_t>` instead of `bool`.

10.7.2.1 Examples of code

Example of code (before version 5.0.0):

```
    const bool res = license->isActivated();
    if (!res) {
        /* error case code */
    }
```

Example of code (from version 5.0.0):

```
    const fsdk::ResultValue<fsdk::FSDKError, bool> result = license->
        isActivated();
    if (result.isError()) {
        /* error case code */
    }

    const bool value = result.getValue();
    if (!value) {
        /* false case code */
    }
```

From version v.5.0.0 we changed the arguments of methods getExpirationDate and checkFeatureId in class ILicense. Now the input arguments of getExpirationDate and checkFeatureId is fsdk::LicenseFeature instead of uint32_t. And the second argument of getExpirationDate was removed. The return value of getExpirationDate is fsdk::ResultValue<fsdk::FSDKError , uint32_t>.

Example of code (before version 5.0.0):

```
    long long expDate = 0;
    const bool result =
        license->getExpirationDate(static_cast<uint32_t>(fsdk::
            LicenseFeature::Detection), expDate);

    if (result == false) {
        /* error case code */
    }
```

Example of code (from version 5.0.0):

```
    const fsdk::ResultValue<fsdk::FSDKError, uint32_t> result =
        license->getExpirationDate(fsdk::LicenseFeature::Detection);

    if (result.isError()) {
        /* error case code */
    }

    const uint32_t expDate = result.getValue();
```

Example of code (before version 5.0.0):

```
        const bool res = license->checkFeatureId(static_cast<uint32_t>(fsdk::
            LicenseFeature::Detection));
        if (!res) {
            /* error case code */
        }
```

Example of code (from version 5.0.0):

```
        const fsdk::ResultValue<fsdk::FSDKError, bool> result = license->
            checkFeatureId(fsdk::LicenseFeature::Detection);
        if (result.isError()) {
            /* error case code */
        }

        const bool value = result.getValue();
        if (!value) {
            /* false case code */
        }
```

10.7.3 Interface of HumanLandmark

From version v.5.0.0 we changed the interface of HumanLandmark. Now member point doesn't store zero coordinates in the case when it is not visible. For this purposes we added member visible which stores true if point is visible.

Example of code (before version 5.0.0):

```
    if (humanLandmark.point.x == 0 && humanLandmark.point.y == 0) {
        // point is not visible case code
    }
    else {
        // point is visible case code
    }
```

Example of code (from version 5.0.0):

```
    if (humanLandmark.visible == false) {
        // point is not visible case code
    }
    else {
        // point is visible case code
    }
```

10.7.4 Interface of IDescriptorBatch

From version v.5.0.0 we renamed method `IDescriptorBatch::getDescriptorSize()` to `IDescriptorBatch::getDescriptorLength()`.

Example of code (before version 5.0.0):

```
uint32_t descriptorLength = descriptorBatch->getDescriptorSize();
```

Example of code (from version 5.0.0):

```
uint32_t descriptorLength = descriptorBatch->getDescriptorLength();
```

10.7.5 Interface of Detection

From version v.5.0.0 we changed the interface of the `Detection` structure. Now all members of this structure are private and could be available through the public methods.

Example of code (before version 5.0.0):

```
fsdk::Detection detection = ...; // Somehow initialized detection object
fsdk::Rect rect = detection.rect; // Get the detection rect
float score = detection.score; // Get the detection score
```

Example of code (from version 5.0.0):

```
fsdk::Detection detection = ...; // Somehow initialized detection object
fsdk::Rect rect = detection.getRect(); // Get the detection rect
float score = detection.getScore(); // Get the detection score
```

10.7.6 Interface of IDetector

From version v.5.0.0 we changed the interface of `IDetector` structure. Now method `detect` returns `ResultValue<FSDKError, Ref<IFaceDetectionBatch>>` instead of `ResultValue<FSDKError, Ref<IResultBatch<Face>>>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IResultBatch<fsdk::Face
    >>> detectorResult = faceDetector->detect(
        fsdk::Span<const fsdk::Image>(&image, 1),
```

```
        fsdk::Span<const fsdk::Rect>(&imageRect, 1),
        detectionsCount,
        fsdk::DT_ALL);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IFaceDetectionBatch>>
    detectorResult = faceDetector->detect(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Rect>(&imageRect, 1),
        detectionsCount,
        fsdk::DT_ALL);
```

Also we changed input and output parameters of the method `redetectOne`. Now it takes `Image` and `Detection` instead of `Face`. And returns `ResultValue<FSDKError, Face>` instead of `ResultValue<FSDKError, bool>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, bool> redetectResult = faceDetector->
    redetectOne(face);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Face> redetectResult = faceDetector
    ->redetectOne(image, detection);
```

10.7.7 IFaceDetectionBatch

We added `IFaceDetectionBatch` structure to replace `IResultBatch<Face>`.

Example of code (before version 5.0.0):

```
fsdk::Ref<IResultBatch<Face>> resultBatch = ...; // Somehow get the
    IResultBatch<Face>
for (std::size_t i = 0; i < resultBatch->getSize(); ++i) {
    fsdk::Span<fsdk::Face> faces = resultBatch->getResults(i);
    for (auto& face : faces) {
        const fsdk::Rect& rect = face.detection.rect;
        const float score = face.detection.score;
        const fsdk::Landmarks5& lm5 = face.landmarks5.value();
        const fsdk::Landmarks68& lm68 = face.landmarks68.value();
```

```
        // Some code which uses received objects
    }
  }
```

Example of code (from version 5.0.0):

```
  fsdk::Ref<fsdk::IFaceDetectionBatch> faceDetectionBatch = ...; // Somehow
      get the IFaceDetectionBatch
  for (std::size_t i = 0; i < faceDetectionBatch->getSize(); ++i) {
      fsdk::Span<const fsdk::Detection> detections = faceDetectionBatch->
          getDetections(i);
      fsdk::Span<const fsdk::Landmarks5> landmarks5 = faceDetectionBatch->
          getLandmarks5(i);
      fsdk::Span<const fsdk::Landmarks68> landmarks68 = faceDetectionBatch->
          getLandmarks68(i);
      for (std::size_t j = 0; j < detections.size(); ++j) {
          const fsdk::Rect& rect = detections[j].getRect();
          const float score = detections[j].getScore();
          const fsdk::Landmarks5& lm5 = landmarks5[j];
          const fsdk::Landmarks68& lm68 = landmarks68[j];
          // Some code which uses received objects
      }
  }
```

10.7.8 Interface of IHumanDetector

From version v.5.0.0 we changed the interface of `IHumanDetector` structure. Now method `detect` returns `ResultValue<FSDKError, Ref<IHumanDetectionBatch>>` instead of `ResultValue< FSDKError, Ref<IResultBatch<Human>>>`.

Example of code (before version 5.0.0):

```
  fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IResultBatch<fsdk::Human
      >>> detectResult = humanDetector->detect(
          fsdk::Span<const fsdk::Image>(&image, 1),
          fsdk::Span<const fsdk::Rect>(&imageRect, 1),
          detectionsCount,
          fsdk::DCT_ALL);
```

Example of code (from version 5.0.0):

```
  fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IHumanDetectionBatch>>
      detectResult = humanDetector->detect(
```

```
            fsdk::Span<const fsdk::Image>(&image, 1),
            fsdk::Span<const fsdk::Rect>(&imageRect, 1),
            detectionsCount,
            fsdk::HDT_ALL);
```

Also we changed input and output parameters of the method `redetectOne`. Now it takes `Image` and `Detection` instead of `Human`. And returns `ResultValue<FSDKError, Human>` instead of `ResultValue<FSDKError, bool>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, bool> redetectResult = humanDetector->
    redetectOne(human);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Human> redetectResult =
    humanDetector->redetectOne(image, detection);
```

10.7.9 IHumanDetectionBatch

We added `IHumanDetectionBatch` structure to replace `IResultBatch<Human>`.

Example of code (before version 5.0.0):

```
fsdk::Ref<IResultBatch<Human>> resultBatch = ...; // Somehow get the
    IResultBatch<Human>
for (std::size_t i = 0; i < resultBatch->getSize(); ++i) {
    fsdk::Span<fsdk::Human> humans = resultBatch->getResults(i);
    for (auto& human : humans) {
        const fsdk::Rect& rect = human.detection.rect;
        const float score = human.detection.score;
        const fsdk::Landmarks17& lm17 = face.landmarks5.value();
        // Some code which uses received objects
    }
}
```

Example of code (from version 5.0.0):

```
const fsdk::Ref<fsdk::IHumanDetectionBatch> humanDetectionBatch = ...; //
    Somehow get the IHumanDetectionBatch
for (std::size_t i = 0; i < humanDetectionBatch->getSize(); ++i) {
```

```
        fsdk::Span<const fsdk::Detection> detections = humanDetectionBatch->
            getDetections(i);
        fsdk::Span<const fsdk::HumanLandmarks17> landmarks = humanDetectionBatch
            ->getLandmarks17(i);
        for (std::size_t j = 0; j < detections.size(); ++j) {
            const fsdk::Rect rect = detections[j].getRect();
            const float score = detections[j].getScore();
            const fsdk::HumanLandmarks17 lm17 = landmarks[j];
            // Some code which uses received objects
        }
    }
```

10.7.10 Interface of ILivenessFlyingFaces

From version v.5.0.0 we changed the interface of `ILivenessFlyingFaces` structure. Now both methods `estimate` take `Image` and `Detection` instead of `Face`.

Example of code (before version 5.0.0):

```
  fsdk::LivenessFlyingFacesEstimation flyingFacesEstimation;
  Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
      estimate(face, flyingFacesEstimation);
```

Example of code (from version 5.0.0):

```
  fsdk::LivenessFlyingFacesEstimation flyingFacesEstimation;
  Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
      estimate(
          image,
          detection,
          flyingFacesEstimation);
```

Example of code (before version 5.0.0):

```
  Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
      estimate(
          fsdk::Span<const fsdk::Face>(&face, 1),
          fsdk::Span<fsdk::LivenessFlyingFacesEstimation>(&estimation, 1));
```

Example of code (from version 5.0.0):

```
fsdk::LivenessFlyingFacesEstimation flyingFacesEstimation;
Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
    estimate(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Detection>(&detection, 1),
        fsdk::Span<fsdk::LivenessFlyingFacesEstimation>(&
            flyingFacesEstimation, 1));
```

10.8 v.3.10.1

10.8.1 Detector FaceDetV3 changes

From version 3.10.1 we changed the logic for image resizing in FaceDetV3 detector. Now you can change the minimum and maximum sizes of the faces that will be searched in the photo from the `faceengine.conf` configuration. To get new parameter which will be identical to old setting you need to set `minFaceSize`:

The old recommended `imageSize=640` will be identical to new meaning of setting `minFaceSize=20`

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

and `imageSize=320` will be identical to new meaning of setting `minFaceSize=40`

```
config->setValue("FaceDetV3::Settings", "minFaceSize", 40);
```

10.8.2 Detector FaceDetV1, FaceDetV2 changes

From version 3.10.1 we changed the name of parameter `minSize` to `minFaceSize` in `faceengine.conf` for FaceDetV1, FaceDetV2 detector types. The logic and default value for image resizing left unchanged.

11 Best practices

11.1 Overview

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

11.1.1 Creation and deletion order

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

- at first FaceEngine object should be created (using createFaceEngine method)

- after that all child objects, such as detectors, estimators etc, can be created

- at the end of the work all these child objects should be deleted in the first place

- and only after that the main FaceEngine object can be deleted

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

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

int main() {
    // application code here

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

11.1.2 Multithread scenario

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

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

11.1.3 Thread pools

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

11.1.4 Estimators. Creation and Inference

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

11.1.5 Forking process

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

12 Device-specific constraints

12.1 Image constraints

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

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

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

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

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

- construction (both with Image-owned memory and as a wrapper for a user-defined memory) and assignment (including deep copy);

- destruction;

- set() family of functions (functionally the same as construction/assignment);

- convert() function, but only in transfer mode; This means that both source and destination formats must match, only memory residency may differ. This function supports only synchronous memory transfers in the following directions:

    - host <-> GPU
    - GPU <-> GPU
    - host <-> NPU
    - NPU <-> NPU.

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

The following operations are **NOT** supported:

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

13 Appendix A. Specifications

13.1 Classification performance

Classification performance was measured on a two datasets:

- Cooperative dataset (containing 20K images from various sources obtained at several banks);
- Non cooperative dataset (containing 20K).

The two tables below contain true positive rates corresponding to select false positive rates.

Table 56: "Classification performance @ low FPR on cooperative dataset"

FPR	TPR CNN 58	TPR CNN 59	TPR CNN 59m	TPR CNN 60	TPR CNN 60m	TPR CNN 62
10^{-7}	0.9910	0.9911	0.9876	0.9917	0.9660	0.9909
10^{-6}	0.9916	0.9915	0.9904	0.9917	0.9824	0.9950
10^{-5}	0.9918	0.9919	0.9915	0.9919	0.9889	0.9976
10^{-4}	0.9919	0.9921	0.9919	0.9921	0.9909	0.9988

Table 57: "Classification performance @ low FPR on non cooperative dataset"

FPR	TPR CNN 58	TPR CNN 59	TPR CNN 59m	TPR CNN 60	TPR CNN 60m	TPR CNN 62
10^{-7}	0.9767	0.9832	0.9377	0.9893	0.8797	0.9916
10^{-6}	0.9839	0.9880	0.9629	0.9914	0.9246	0.9917
10^{-5}	0.9880	0.9908	0.9794	0.9914	0.9595	0.9918
10^{-4}	0.9909	0.9924	0.9880	0.9925	0.9821	0.9920

13.2 Descriptor size

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

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

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

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

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

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

14 Appendix B. Glossary

Table 59: Glossary

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

14.1 Descriptor

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

14.2 Cooperative Photoshooting and Recognition

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

Typical highlights:

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

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

14.3 Matching

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

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

15 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".