



VisionLabs
MACHINES CAN SEE

TrackEngine Handbook

Contents

Introduction	5
Glossary	5
Batch	5
Best shot	5
Track	5
Tracking	5
Working with TrackEngine	5
Using the asynchronous method	6
Using the estimator tracking API	6
Creating a TrackEngine instance	6
Creating the main interface	7
Concurrent streams	8
Finishing work with TrackEngine	8
Estimator API	8
Async API	9
Observers	10
Examples	11
Setting the per-stream observer API	11
Setting the batched observer API	11
Setting the unified batched observer API	11
IBestShotObserver	12
IVisualObserver	13
IDebugObserver	14
BestShotPredicate	15
VisualPredicate	15
IBatchBestShotObserver	15
IBatchVisualObserver	17
IBatchDebugObserver	17
ITrackingResultObserver	18
Track lifetime	21
Body tracking algorithm	21
Human tracking algorithm	22
Reidentification	22
Receiving tracking results	22

Memory consumption	23
Threading	23
Tracker	24
ROI	24
Settings	25
Logging section	25
mode	25
log-file-path	25
severity	25
Other parameters section	26
callback-mode	26
detector-step	26
detector-comparer	27
use-one-detection-mode	27
skip-frames	27
frg-subtractor	28
frames-buffer-size	28
callback-buffer-size	28
tracking-results-buffer-size	29
detector-scaling	29
scale-result-size	29
max-detection-count	30
minimal-track-length	30
tracker-type	30
kill-intersected-detections	31
kill-intersection-value	31
FRG section	31
frg-subtractor-type	31
use-binary-frg	32
frg-update-step	32
frg-scale-size	32
frg-regions-alignment	33
frg-regions-square-alignment	33
Vehicle section	33
best-shots-number-for-track	33
max-processing-fragments-count	34
Face tracking specific parameters section	34
face-landmarks-detection	34

Human/Body tracking specific parameters section	34
remove-overlapped-strategy	34
remove-horizontal-ratio	35
iou-connection-threshold	35
use-body-reid	35
body-reid-version	35
reid-matching-threshold	36
reid-matching-detections-count	36
inactive-tracks-lifetime	36
Detectors section	36
use-face-detector	36
use-body-detector	37
use-vehicle-detector	37
use-license-plate-detector	38
Experimental parameters section	38
detect-max-batch-size	38
redetect-max-batch-size	38
tracker-max-batch-size	38
reid-max-batch-size	39
min-frames-batch-size	39
max-frames-batch-gather-timeout	39
Config example	40
Example	42

Introduction

TrackEngine is a library for human detection and tracking on multiple sources.

TrackEngine itself does not perform any facial recognition. Its purpose is to **prepare** required data for external systems, for example VisionLabs LUNA PLATFORM.

Glossary

Batch

Group of data processed simultaneously.

Best shot

The frame of the video stream on which the face/body is fixed in the optimal angle for further processing.

Track

Information on face position of a single person on a frame sequence.

Tracking

Function that follows an object (face) through a frame sequence.

Working with TrackEngine

TrackEngine is based on face detection and analysis methods provided by the FaceEngine library.

There are two ways of working with TrackEngine:

1. Using the asynchronous pushFrame or callback method.
It allows you to use a simple API with asynchronous push frames per each stream and get all tracking result events or data in another thread in callbacks. For details, see `IStream::pushFrame` in the TrackEngine example.
2. Using the estimator tracking API.
It works like the SDK estimator with an internal state. You should pass a batch of streams or frames to the function and get ready tracking results for input streams. This way is more complex, but flexible for developers. For details, see `ITrackEngine::trackin` in the TrackEngine example.

Using the asynchronous method

The main advantage of the asynchronous method is simplicity of client side code. You will not have to deal with such tasks as:

- Exception handling
- Issue multithreading
- Creating queues for multiple stream batch gathering
- Deferred result processing

This logic is implemented in TrackEngine by `callback-mode = 1`.

The common solution is:

1. Create a stream per each frame source.
2. Set up callback observers.
3. Submit frames to each stream one by one.
Depending on your app architecture, you can push frames to each stream from different threads. But we recommend that you use one thread per each stream and frame source.

Tracking results are obtained in callbacks from another thread that is created in TrackEngine. This is the place where you should write logic of processing results. When a stream has to be finished, call the `IStream::join` method to wait all queued frames or callbacks to be processed. After that, the stream cannot be used anymore and can be released.

Using the estimator tracking API

If you want to control the logic of tracking at most, use the estimator tracking API.

Benefits of using the API:

- Minimal memory consumption of TrackEngine. It allows you to achieve better performance, because it does not keep images in any queues. Yet, images are still kept in track data.
- No need to configure parameters that regulate buffer sizes or batching settings. For example, `tracking-results-buffer-size`, `frames-buffer-size`, `callback-buffer-size`, `min-frames-batch-size`, `max-frames-batch-gather-timeout`. Also, streams should call `stop` instead of `join` on finish. In this case, a stream serves only as a state object for tracking.

Important: To use the estimator API, set the `callback-mode` parameter to 0, otherwise value 1 must be set (default value is 1).

Creating a TrackEngine instance

To create a TrackEngine instance use the following global factory functions:

- `__ITrackEngine* tsdk::createTrackEngine(fsdk::IFaceEngine* engine, const char* configPath, vsdk::IVehicleEngine* vehicleEngine = nullptr, const fsdk::LaunchOptions *launchOptions = nullptr)___`
 - `engine` - Pointer to FaceEngine instance (should be already initialized).
 - `configPath` - Path to the TrackEngine config file.
 - `vehicleEngine` - Pointer to the VehicleEngine object (if with vehicle logic).
 - `launchOptions` - Launch options for SDK functions.
 - `return value` - Pointer to ITrackEngine.
- `__ITrackEngine* tsdk::createTrackEngine(fsdk::IFaceEngine* engine, const fsdk::ISettingsProviderPtr& provider, vsdk::IVehicleEngine* vehicleEngine = nullptr, const fsdk::LaunchOptions *launchOptions = nullptr)___`
 - `engine` - Pointer to FaceEngine instance (should be already initialized).
 - `provider` - Settings provider with the TrackEngine configuration.
 - `vehicleEngine` - Pointer to the VehicleEngine object (if with vehicle logic).
 - `launchOptions` - Launch options for SDK functions.
 - `return value` - Pointer to ITrackEngine.

Important: At most one TrackEngine instance per process may exist simultaneously. Please see FaceEngine Handbook/SDK Workflow for a detailed list of lifetime and threading constraints.

Creating the main interface

The main interface to TrackEngine is stream - an entity to which you submit video frames.

To create a stream use the following TrackEngine methods:

- `__IStream* ITrackEngine::createStream(StreamParams *params = nullptr)___`
 - `params` - Pointer to stream specific parameters. It is an optional parameter. If valid, then it overrides configuration parameters for a stream. For details, see StreamParams in the TrackEngine example.
 - `return value` - Pointer to ITrackEngine.
- **`IStream* ITrackEngine::createStreamWithParams(const StreamParamsOpt ¶ms)`**
 - `params` - Stream specific parameters. Each parameter in the struct is optional. If it is valid, then overrides the same config parameter for the stream. For details, see StreamParamsOpt in the TrackEngine example.
 - `return value` - Pointer to ITrackEngine.

Important: You must own this raw pointer by `fsdk::Ref`, for example, with `fsdk::acquire` and reset all references to all streams before TrackEngine object destruction. Otherwise memory

leak or/and UB are guaranteed. This is valuable especially in languages where order of objects destruction is not guaranteed, so you should manage objects lifetime manually (for example, python).

Concurrent streams

You can create multiple streams working concurrently, for example, if you need to track faces from several cameras. In each stream, the engine detects faces and builds their tracks. Each face track has its own unique identifier. It is therefore possible to group face images belonging to the same person with their track IDs. Please note, tracks may break from time to time on either of the reasons:

- Due to people leaving the visible area.
- Due to the challenging detection conditions. For example, poor image quality, occlusions, extreme head poses, and so on.

Finishing work with TrackEngine

At the end of work with TrackEngine, call `ITrackEngine::stop` before the TrackEngine object can be released.

- **`void ITrackEngine::stop()`** - Stops processing.

Estimator API

- **`fsdk::ResultValue<fsdk::FSDKError, ITrackingResultBatchPtr> track(fsdk::Span streams, fsdk::Span frames)`** - Updates stream tracks by a new frame per each stream and returns ready tracking results data for passed streams (as callbacks compatible data).
 - `streams` - Streams stream identifiers, must contain only unique IDs, otherwise function throws. For details, see `IStream::getId`.
 - `frames` - Frames input frames per a stream. For details, see `IStream::pushFrame` and `Frame`.
 - `return value` - First error code and reference to ready tracking results as callbacks compatible data. For details, see `ITrackingResultBatch`.

We recommend that you make each frame (from `frames`) image to be the data owner. Otherwise, performance overhead is possible as TrackEngine internally will clone it to keep in track data. The function returns only ready tracking results per each stream, so it can return tracking results for stream previously passed frames, as well as not return results for the current passed frame. The reason of such delay is that tracking may require several frames to get results. For details, see the `Receiving tracking results` section. It is thread safe, but call blocking. The function is not exception safe like `pushFrame`.

Important: Regulating batch size for `track` is the critical step to achieve high performance of tracking. Higher values lead to better utilization or throughput (especially on GPU), but increase latency of system.

For pre-validation of track inputs, `noexcept function validate` is useful.

- **`fsdk::Result validate(fsdk::Span streams, fsdk::Span frames, fsdk::Span<fsdk::Result> outErrors)`** - Validates input of multiple streams or frames in a single function call.
 - `streams` - Streams an array of stream identifiers array.
 - `frames` - Frames input frames per a stream.
 - `outErrors` - Error output span of errors for each stream or frame.
 - `return value` - Result with the last error code.

When a stream has to be finished, call the `IStream stop` method before a stream release to get all remaining tracking results. You can use the stream for tracking after calling this function.

- **`fsdk::Ref stop(bool reset = false)`** - Finishes all current tracks and returns all remaining tracking results.
 - `reset` - If set to `true`, resets a stream state to initial. Otherwise, keeps internal frame counter, statistics, and so on.
 - `return value` - Remaining tracking results for the stream.

Async API

- **`__bool IStream::pushFrame(const fsdk::Image &frame, uint32_t frameId, tsdk::AdditionalFrameData *data = nullptr)__`** - Pushes a single frame to the stream buffer.
 - `frame` - Frame image input. The format must be R8G8B8 OR R8G8B8X8.
 - `frameId` - Unique identifier for a frame sequence.
 - `data` - Additional data that a developer wants to receive in a callback realization. It must be allocated only with `new` or be equal to `nullptr`. Do not use the delete-operator. The garbage collector is implemented inside `TrackEngine` for this parameter.
 - `return value` - Returns `true`, if a frame was appended to the queue for processing. Returns `false` otherwise - a frame was skipped because of the full queue.

We recommend that you make `frame` to be owner of data, otherwise performance overhead is possible as `TrackEngine` internally will clone it to keep in track data. Also there are some variations of this method: `pushCustomFrame`, `pushFrameWaitFor`, `pushCustomFrameWaitFor`.

When a stream has to be finished, call the `IStream join` method before a stream release. You cannot use the stream after joining for processing, only “getter” functions are available.

- **`void IStream::join()`** - Blocks the current thread until all frames in this stream are handled and all callbacks are executed.

Important: Ignoring this step may lead to unexpected behavior. TrackEngine writes a warning log in this case.

Observers

You can set up an observer to receive and react on events. There are two types of observers:

- Per-stream specific single observer
 - > Per-stream observers are deprecated now and remained only for compatibility with old versions.
- Batched observer for all streams
 - > We recommend that you use the new batched observers API instead of old per-stream one.

Batched observers have some advantages over per-stream observers. Batched observers:

- Reduce and set a fixed number of threads created by TrackEngine. For details, see the Threading section.
- Eliminate performance overhead from multiple concurrently working threads used for per-stream callbacks.
- Allow to easily use the batched SDK API without additional aggregation of data from single callbacks. Both for GPU/CPU, the batched SDK API improves performance. For GPU effect is much more significant.
- Give more information in output. Per-stream callback function signatures remain the same because of compatibility with old versions.

Stream observer interfaces:

Per-stream observers	Batched event specific observers	Batched unified observer
IBestShotObserver	IBatchBestShotObserver	ITrackingResultObserver
IVisualObserver	IBatchVisualObserver	
IDebugObserver	IBatchDebugObserver	

ITrackingResultObserver is the most recommended observer to work with when `callback-mode = 1`. It provides all ready tracking results or events in one structure or callback. This observer contains only one function ready. It is called every time when tracking results are ready for any streams or frames.

You can be sure that all frames per each stream from ITrackingResultBatch were processed and can write logic based on this knowledge.

The ready function will be called once per each stream or frame. The exception is the last stream frame when `trackEnd` is called for all remaining tracks (inactive, too) on Stream join. The reason is that TrackEngine cannot define which frame actually will be the last one.

Note, that its value type is the same as for the track method (used for callback-mode = 0).

The priority of observer use is as follows:

1. ITrackingResultObserver
2. Batched event specific observers
3. Per-stream observers

It means, that if 1 is set up, then it will be used. Otherwise, if 2 is set up, then it will be used, otherwise - 3.

Important: You have to setup either single per-stream or batched observers for all streams, but not both at the same time.

The IBestShotPredicate type defines recognition suitability criteria for face detections. By implementing a custom predicate, you can alter the best shot selection logic and, therefore, specify which images will make it to the recognition phase.

Important: Placing images of different resolutions in one stream is not recommended.

Examples

Setting the per-stream observer API

- **void IStream::setBestShotObserver(tsd::IBestShotObserver* observer)** - Sets a best shot observer for a stream.
 - *observer* - Pointer to the observer object. For details, see IBestShotObserver. Do not set to nullptr, if you want disable it. Instead, set IStream::setObserverEnabled to false.

Setting the batched observer API

- **__void ITrackEngine::setBatchBestShotObserver(tsd::IBatchBestShotObserver *observer)__** - Sets a best shot observer for all streams.
 - *observer* - Pointer to the batched observer object. For details, see IBatchBestShotObserver. Do not set to nullptr, if you want disable it. Instead, set IStream::setObserverEnabled to false.

Setting the unified batched observer API

- **__void ITrackEngine::setTrackingResultObserver(tsd::ITrackingResultObserver*observer)__** Sets a unified tracking result observer for all streams.
 - *observer* - Pointer to the observer object. For details, see ITrackingResultObserver. If set to nullptr, then batched observers will be used per event.

IBestShotObserver

- **void bestShot(const tsdk::DetectionDescr& descr)** - Called for each emerged best shot. It provides information on a best shot, including frame number, detection coordinates, cropped still image, and other data. See the DetectionDescr structure definition below for details. Default implementation does nothing.
 - descr - Best shot detection description.

```
struct TRACK_ENGINE_API DetectionDescr {
    //! Index of the frame
    tsdk::FrameId frameIndex;

    //! Index of the track
    tsdk::TrackId trackId;

    //! Source image
    fsdk::Image image;

    fsdk::Ref<ICustomFrame> customFrame;

    //! Face landmarks
    fsdk::Landmarks5 landmarks;

#ifdef MOBILE_BUILD
    //! Human landmarks
    fsdk::HumanLandmarks17 humanLandmarks;

    //! NOTE: only for internal usage, don't use this field, it isn't valid
    ptr
    fsdk::IDescriptorPtr descriptor;
#endif

    //! Is it full detection or redetect step
    bool isFullDetect;

    //! Detections flags
    // needed to determine what detections are valid in extraDetections
    // see EDetectionFlags
    uint32_t detectionsFlags;

    //! Detection
    // always is valid, even when detectionsFlags is combination type
    // useful for one detector case
    // see detectionObject
```

```
fsdk::Detection detection;
};
```

- **void trackEnd(const tsdk::TrackId& trackId)** - Indicates that a track with trackId has ended and no more best shots should be expected from it. Default implementation does nothing.
 - trackId - ID of a track.

```
/** @brief Track status enum. (see human tracking algorithm section in docs
    for details)
 */
enum class TrackStatus : uint8_t {
    ACTIVE = 0,
    NONACTIVE
};
```

IVisualObserver

- **void visual(const tsdk::FrameId &frameId, const fsdk::Image &image, const tsdk::TrackInfo * trackInfo, const int nTrack)** - Allows to visualize the current stream state. It is intended mainly for debugging purposes. The function must be overloaded.
 - frameId - Current frame ID.
 - image - Frame image.
 - trackInfo - Array of currently active tracks.
 - nTrack - Number of tracks.

```
struct TRACK_ENGINE_API TrackInfo {
    //! Face landmarks
    fsdk::Landmarks5 landmarks;

    #if !TE_MOBILE_BUILD
        //! Human landmarks
        fsdk::HumanLandmarks17 humanLandmarks;
    #endif

    //! Last detection for track
    fsdk::Rect rect;

    //! Id of track
    TrackId trackId;

    //! Score for last detection in track
```

```

float lastDetectionScore;

//! Detector id
TDetectorID m_sourceDetectorId;

//! number of detections for track (count of frames when track was
    updated with detect/redetect)
size_t detectionsCount;

//! id of frame, when track was created
tsdk::FrameId firstFrameId;

//! Is it (re)detected or tracked bounding box
bool isDetector;
};

```

IDebugObserver

- **void debugDetection(const tsdk::DetectionDebugInfo& descr)** - Detector debug callback. Default implementation does nothing.
 - descr - Detection debugging description.

```

struct DetectionDebugInfo {
    //! Detection description
    DetectionDescr descr;

    //! Is it detected or tracked bounding box
    bool isDetector;

    //! Filtered by user bestShotPredicate or not.
    bool isFiltered;

    //! Best detection for current moment or not
    bool isBestDetection;
};

```

- **void debugForegroundSubtraction(const tsdk::FrameId& frameId, const fsdk::Image& firstMask, const fsdk::Image& secondMask, fsdk::Rect * regions, int nRegions)** - Background subtraction debug callback. Default implementation does nothing.
 - frameId - Foreground frame ID.
 - firstMask - Result of the background subtraction operation.

- secondMask - Result of the background subtraction operation after procedures of erosion and dilation.
- regions - Regions obtained after the background subtraction operation.
- nRegions - Number of returned regions.

BestShotPredicate

- **bool checkBestShot(const tsdk::DetectionDescr& descr)** - Predicate for best shot detection. This is the place to perform any required quality checks (by means of, for example, FaceEngines Estimators). This function must be overloaded.
 - descr - Detection description.
 - return value - true, if descr has passed the check, false otherwise.

VisualPredicate

- **__bool needRGBImage(const tsdk::FrameId frameId, const tsdk::AdditionalFrameData *data)** - Predicate for visual callback. It serves to decide whether to output original image in visual callback or not. This function can be overloaded. Default implementation returns true.
 - frameId - ID of the frame.
 - data - Frame additional data passed by user.
 - return value - true, if the original image or an RGB image for a custom frame needed in output in visual callback, false otherwise.

IBatchBestShotObserver

- **void bestShot(const fsdk::Span &streamIDs, const fsdk::Span &data)** - Batched version of the bestShot callback.
 - streamIDs - Array of stream IDs.
 - data - Array of callback data for each stream.

```
struct TRACK_ENGINE_API BestShotCallbackData {
    //! detection description. see 'DetectionDescr' for details
    tsdk::DetectionDescr descr;

    //! additional frame data, passed by user in 'pushFrame'. see '
    AdditionalFrameData' for details
    tsdk::AdditionalFrameData *frameData;
};
```

- **void trackEnd(const fsdk::Span &streamIDs, const fsdk::Span &data)** - Batched version of the trackEnd callback.
 - streamIDs - Array of stream IDs.
 - data - Array of callback data for each stream.

```
/**
 * @brief Track end reason. See 'TrackEndCallbackData' for details.
 */
enum class TrackEndReason : uint8_t {
    //! not used anymore, deprecated value (may be removed in future
    releases)
    DEFAULT,
    //! note: deprecated field, not used anymore
    UNKNOWN,
    //! intersection with another track (see "kill-intersected-detections")
    INTERSECTION,
    //! tracker is disabled or failed to update track
    TRACKER_FAIL,
    //! track's gone out of frame
    OUT_OF_FRAME,
    //! `skip-frames` parameter logic (see docs or config comments for
    details)
    SKIP_FRAMES,
    //! note: deprecated field, not used anymore
    USER,
    //! note: deprecated field, not used anymore
    NONACTIVE_TIMEOUT,
    //! note: deprecated field, not used anymore
    ACTIVE_REID,
    //! note: deprecated field, not used anymore
    NONACTIVE_REID,
    //! all stream tracks end on stream finishing (IStream::join called)
    STREAM_END
};

struct TRACK_ENGINE_API TrackEndCallbackData {
    //! frame id
    tsdk::FrameId frameId;

    //! track id
    tsdk::TrackId trackId;

    //! parameter implies reason of track ending
};
```



```
// NOTE: now it's using only for human tracking, don't use this for
// other detectors
TrackEndReason reason;
};
```

IBatchVisualObserver

- **void visual(const fsdk::Span &streamIDs, const fsdk::Span &data)** - Batched version of the visual callback.
 - streamIDs - Array of stream IDs.
 - data - Array of callback data for each stream.

```
struct TRACK_ENGINE_API VisualCallbackData {
    //! frame id
    tsdk::FrameId frameId;

    //! this is either original image (if 'pushFrame' used) or RGB image got
    from custom frame convert (is 'pushCustomFrame' used)
    fsdk::Image image;

    //! tracks array raw ptr
    tsdk::TrackInfo *trackInfo;

    //! number of tracks
    int nTrack;

    //! additional frame data, passed by user in 'pushFrame'. See '
    AdditionalFrameData' for details.
    tsdk::AdditionalFrameData *frameData;
};
```

IBatchDebugObserver

- **void debugForegroundSubtraction(const fsdk::Span &streamIDs, const fsdk::Span &data)** - Batched version of the debugForegroundSubtraction callback.
 - streamIDs - Array of stream IDs.
 - data - Array of callback data for each stream.
- **void debugDetection(const fsdk::Span &streamIDs, const fsdk::Span &data)** - Batched version of the debugDetection callback.
 - streamIDs - Array of stream IDs.

- data - Array of callback data for each stream.

```
struct TRACK_ENGINE_API DebugForegroundSubtractionCallbackData {
    //! frame id
    tsdk::FrameId frameId;

    //! first mask of the foreground subtraction
    fsdk::Image firstMask;

    //! second mask of the foreground subtraction
    fsdk::Image secondMask;

    //! regions array raw ptr
    fsdk::Rect *regions;

    //! number of regions
    int nRegions;
};

/** @brief Detection data for debug callback.
 */
struct TRACK_ENGINE_API DebugDetectionCallbackData {
    //! Detection description
    DetectionDescr descr;

    //! Is it detected or tracked bounding box
    bool isDetector;

    //! Filtered by user bestShotPredicate or not.
    bool isFiltered;

    //! Best detection for current moment or not
    bool isBestDetection;
};
```

ITrackingResultObserver

- **void IStream::setObserverEnabled(tsd::StreamObserverType type, bool enabled)** - Enables or disables the observer.
 - type - Observer type.
 - enabled - Flag to enable or disable the observer

For full Stream API, see class IStream from IStream.h header file.

```

/* @brief Ready tracking result observer interface.
*/
struct TRACK_ENGINE_API ITrackingResultObserver {
    /**
     * @brief Ready tracking result notification
     * @param value tracking result. See 'ITrackingResultBatch' for details
     * @note pure virtual method
     */
    virtual void ready(fsdk::Ref<ITrackingResultBatch> result) = 0;

    virtual ~ITrackingResultObserver() = default;
};

/** @brief Tracking results per stream/frame.
     It involves different tracking data/events per stream/frame
*/
struct TrackingResult {
    //! stream Id
    tsdk::StreamId streamId;

    //! frame Id
    tsdk::FrameId frameId;

    //! Source image
    fsdk::Image image;

    //! Source custom frame
    fsdk::Ref<ICustomFrame> customFrame;

    //! data passed by user
    tsdk::AdditionalFrameData *userData;

    //! array of trackStart (new tracks) events
    fsdk::Span<TrackStartCallbackData> trackStart;

    //! array of trackEnd (finished tracks) events
    fsdk::Span<TrackEndCallbackData> trackEnd;

    //! array of current tracks data (each element of array matches to
    tracks of specific `EDetectionObject`)
    fsdk::Span<VisualCallbackData> tracks;

    //! array of detections, so this implies only detections, received from
    Detector (bestshots in case of custom `checkBestShot` func)

```

```

    // note: `detections` can be deduced from `tracks` actually (see fields
    // `isDetector`, `isFullDetect`)
    // but can be useful when `checkBestShot` is used or in simple cases,
    // when only detections from Detector are needed
    fsdk::Span<BestShotCallbackData> detections;

    //! array of debug detections data (it, mostly, copies data from `tracks`
    // with some extra debug params like `isFiltered`, `isBestDetection`)
    // note: deprecated field, use `tracks` and `detections` instead for
    // better experience
    fsdk::Span<DebugDetectionCallbackData> debugDetections;

    //! debug foreground subtractions data (mostly, used for debug purposes,
    // but can be used to get FRG info)
    fsdk::Optional<DebugForegroundSubtractionCallbackData>
        debugForegroundSubtraction;

    //! human tracks, including both face and body (new API version of `
    // tracks`)
    // NOTE: visual observer must be enabled to get this, as it's derived
    // from `tracks`
    fsdk::Span<HumanTrackInfo> humanTracks;
};

/** @brief Tracking results batch as 2D vector of stream/frame.
    It contains tracking results for several frames per each stream
    */
struct ITrackingResultBatch : public fsdk::IRefCounted {
    /**
        * @brief Get array of stream identifiers, tracking results are ready
        * for.
        * @return span of Stream ids.
        * */
    virtual fsdk::Span<tsdk::StreamId> getStreamIds() const = 0;

    /**
        * @brief Get array of frame identifiers for given Stream, tracking
        * results are ready for.
        * @param streamId id of the Stream.
        * @note streamId can be any value (not only from `getStreamIds`), so
        * func returns empty span, if Stream has no ready tracking results
        * yet.
        * @return span of frame ids in tracking result for specific Stream.
        * */
    virtual fsdk::Span<tsdk::FrameId> getStreamFrameIds(tsdk::StreamId

```

```

        streamId) const = 0;

    /**
     * @brief Get tracking result by stream Id and frame Id
     * @param streamId id of the Stream, tracking result requested for.
     *         It should be one of the `getStreamIds` array elements.
     * @param frameId id of the Frame, tracking result requested for. It
     *         should be one of the `getStreamFrameIds` array elements for `
     *         streamId`.
     * @return Tracking result for Stream/Frame. See `TrackingResult`
     *         struct for details.
     * @note Return data is valid while the parent ITrackingResultBatch
     *         is alive.
     * @throws std::invalid_argument if streamId is not one of the `
     *         getStreamIds` array elements.
     * @throws std::invalid_argument if frameId is not one of the `
     *         getStreamFrameIds` array elements for passed `streamId`.
     * */
    virtual TrackingResult getTrackingResult(tsd::StreamId streamId, tsdk::
        FrameId frameId) const = 0;
};

```

Track lifetime

All tracks live until they meet specific conditions of a tracking algorithm, for example, out of frame bounds or the skip-frames logic. The human or body tracking algorithm has its own rules for track lifetimes. For details, see section Body tracking algorithm. TrackEndReason implies a reason of track finishing.

Body tracking algorithm

Body tracking algorithm differs from the face one. It does not use the tracker feature, only detect or redetect are used. It also used the intersection over union (IOU) metrics for matching tracks with new detections. The human:iou-connection-threshold parameter is used as a threshold. For better tracking accuracy the ReIdentification feature is used to merge different tracks of one human. For more information, see ReIdentification.

For the face tracking algorithm, when detect or redetect fails, the track is updated with a tracker. But for body tracking in that case or under some other conditions, it moves to the group of inactive tracks. Tracks from this group are invisible for all observers, and they do not participate in common tracking processing except of ReId.

Note, that the skip-frames parameter does not affect the body tracking algorithm. Body tracks are

finished according to their own logic. Now, there is only one case, when `trackEnd` is called for a body track (see the `TrackEndCallbackData` reason field). An inactive track is finished by a timeout specified in the `human:inactive-tracks-lifetime` config parameter. A value of `inactive-tracks-lifetime` should be greater than `reid-matching-detections-count`.

Consider the below algorithm notes and parameters relation.

After detect/redetect, all found detections are filtered by some conditions:

- Overlapped detections may be removed. For overlapping estimation, the IOU metric is used. If the IOU is higher than the `other:kill-intersection-value` threshold parameter, then no one, both or detection with lower detection score, is removed from further processing, depending on parameter `remove-overlapped-strategy`.
- Detections, considered to be horizontal, are removed. `remove-horizontal-ratio` sets detection width to height ratio threshold. It is used for removing horizontal detections.

Human tracking algorithm

The human tracking algorithm is derived from the face and body tracking algorithms and based on `HumanFaceDetector`.

ReIdentification

`ReIdentification` is an optional feature, that improves tracking accuracy (config parameter `use-body-reid`). `ReIdentification` is intended to solve the problem, described in section `Body tracking algorithm`. It matches two different tracks, if needed, and merges them into one track with the ID of the older one. The feature's behavior is regulated by config parameters `reid-matching-threshold`, `reid-matching-detections-count`. Two tracks will be matched only if similarity between them higher than `reid-matching-threshold`.

Note: The current version of `TrackEngine` supports the `ReIdentification` feature only for human/body tracking.

Receiving tracking results

Tracking results for a specific frame can be calculated only after several more frames are submitted to `TrackEngine` for both `callback-mode` types. The reason for such a delay is that tracking may require several frames to get results.

Such logic implies internal buffer of tracking results for several frames. Config parameter `tracking-results-buffer-max-size` regulates the maximum size of this buffer and guarantees that only this

count of frames is stored internally. It limits all other config parameters, such as `reid-matching-detections-count`, which can define how many frames are required to calculate tracking results.

Required frame counts to return tracking results for one frame:

- 1 for face tracking or if ReId feature is disabled for Body/Human tracking.
- Maximum `reid-matching-detections-count` if ReId feature is enabled for Body/Human tracking. This is the maximum value, actually results can be ready earlier.

If `callback-mode = 1`, logic of buffered tracking results adds small latency between frame is pushed. Its tracking results are ready from any callback.

If `callback-mode = 0`, you should expect that `track` may not return any results for the Stream until the required count of frames has been passed to the Stream.

Memory consumption

TrackEngine does not allocate much memory for internal calculations. Yet, it holds images in the current track data (one image per stream) and in frame or callback queues for `callback-mode = 1`.

Tip: To reduce memory consumption, set `frames-buffer-size`, `callback-buffer-size`, and `skip-frames` low enough.

To achieve high optimized minimum memory consumption solution:

- Use estimator API `ITrackEngine::track`.
- Do not keep images in any queues or minimize them in maximum.

Threading

TrackEngine is multi-threaded. Count of threads is configurable. It also depends on the currently bound FaceEngine settings and observer type (batched or single) being used.

TrackEngine calls the Observers functions in separate threads. If you use batched observers, TrackEngine will create only one additional thread and use it for all batched callbacks and streams.

If you use per-stream single observers, TrackEngine creates a separate callback thread for each stream and uses it for its callbacks invocations. In this case, all callbacks are invoked from one thread per-stream.

Note: Regardless of the callback type you use, we recommend that you avoid long-time running tasks in these functions. The reason is that pushing to a callback buffer blocks the main processing thread. The main processing thread always waits until there is a free slot in that buffer to push a callback. You can set a buffer size in the `callback-buffer-size` parameter.

The `checkBestShot` and `needRGBImage` functions are called in the main frame processing thread. We also recommend that you avoid expensive computations in these functions.

These predicates should take zero performance cost.

Excluding calculating SDK threads, thread count guarantees the following:

- Using batched observers guarantees that TrackEngine uses only 2-3 threads itself.
- Using per-stream single observers guarantees that TrackEngine uses only 1-2 + number of created streams threads itself.

Tracker

TrackEngine uses a tracker to update current detections in the case of detect or redetect fails. TrackEngine supports the following trackers:

- `vlTracker` - Neural network based tracker. You can use the tracker for:
 - GPU/NPU processing.
 - *Processing concurrently running multiple streams. It has a batching implementation, so provides better CPU utilization.
- `kcf` and `opencv` - Simple CPU trackers. You should use them only in case of few tracks in total for all streams at the moment.
- `none` - Disables the tracking feature. It leads to better performance, but also to degradation of tracking quality. Supports GPU/NPU.

Some platforms don't support all trackers.

To specify the tacker to be used, set the `tracker-type` parameter.

ROI

TrackEngine supports region of interest (ROI) of tracking. You can set it only with the `humanRelativeROI` per stream parameter from `tsdk::StreamParams/tsdk::StreamParamsOpt`.

Note, that it is a relative ROI, so it sets a rectangle as relative to frame size. If ROI is set, then detector finds faces or humans only in that area, while tracker can move tracks outside of ROI for several frames. You can specify the maximum number of frames in `detector-step`.

If `detector-scaling` is 1, TrackEngine extracts ROI and makes scaling as one operation. Firstly, it extracts ROI, and then scaling, so there is no any overhead on extracting ROI.

A common rule to achieve better performance is:

1. Find a frame size ratio (width / height) for most streams without ROI.
2. Set ROI-s for different streams with equal width or height ratio.

The most optimal case implies that all streams within one application instance have equal ratio of universal ROI (`humanRelativeROI` or original frame size if ROI is not set) and `detector-scaling` is 1.

Consider the TrackEngine example of code to work with ROI.

Note: We recommend that you to use this feature with `detector-scaling` set 1 instead of extracting ROI of original frame before `pushFrame*`.

Settings

This section describes parameters of the `trackengine.conf` file. You can use these parameters to configure TrackEngine.

Logging section

mode

Specifies a logging mode.

Possible values:

Value	Description
<code>l2c</code>	Writes logs to a console only.
<code>l2f</code>	Writes logs to a file.
<code>l2b</code>	Default. Writes logs to a console and file.

```
<param name="mode" type="Value::String" text="l2b" />
```

log-file-path

Specifies a path to the log file.

The default value is `log.txt`.

```
<param name="log-file-path" type="Value::String" text="log.txt" />
```

severity

Specifies a logging severity level.

Possible values:

Value	Description
0	Debug level.
1	Info level.
2	Default. Warning level.
3	Error only level.

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

Other parameters section

callback-mode

Specifies a mode TrackEngine works in.

Possible values:

Value	Description
0	Default. The estimator mode.
1	The asynchronous push or callback mode.

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

detector-step

Specifies a number of frames between frames with full detection.

Possible values are in range 1-30. The lower the number is, the more likely TrackEngine is to detect a new face as soon as it appears. The higher the number is, the higher the performance is. You can use to balance between computation performance and face detection recall.

The default value is 7.

```
<param name="detector-step" type="Value::Int1" x="7" />
```

detector-comparer

Specifies a detector comparer to get the best face detection in a frame based on the following metrics:

Value	Metric	Description
0	DctConfidence	Gets the best face detection by a score.
1	DctCenter	Default. Gets the best face detection if it is located closer to the frame center.
2	DctCenterAndConfidence	Gets the best face detection if it matches the DctConfidence and DctCenter metrics.
3	DctSize	Gets the face detection of a face that has the biggest size.

The parameter works with the use-one-detection-mode parameter.

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

use-one-detection-mode

Specifies a detection mode.

Possible values:

Value	Description
0	Default. Tracks all detections in an image.
1	Tracks one detection in an image.

```
<param name="use-one-detection-mode" type="Value::Int1" x="0" />
```

skip-frames

Specifies a number of frames by which a track must be updated by a detector or re-detector. Otherwise, the track is finished.

Note that very high values may lead to performance degradation.

The parameter applies to face tracking only.

The default value is 18.

```
<param name="skip-frames" type="Value::Int1" x="18" />
```

frg-subtractor

Specifies whether to enable the foreground subtractor. This parameter can improve performance, especially on sources with low activity. Yet, it may reduce face detection recall in rare cases.

Possible values:

Value	Description
0	Disables the foreground subtractor.
1	Default. Enables the foreground subtractor.

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

frames-buffer-size

Specifies a size of the internal storage buffer for input frames.

A larger buffer preserves more frames and reduces skipping. If detection performance is not high enough to keep up with the frame submission rate, the buffer prevents frame loss. Yet, increasing this value also increases RAM/VRAM consumption. You can use the parameter to balance between resource utilization and face detection recall.

The minimal value is 10 because of internal algorithm requirements, for example, batching.

The default value is 20.

The parameter is applied **per stream** and does not affect the estimator API (`callback-mode = 0`).

```
<param name="frames-buffer-size" type="Value::Int1" x="20" />
```

callback-buffer-size

Specifies a size of the internal storage buffer for all callbacks.

A larger buffer ensures higher performance. Yet, it may use more memory.

The parameter does not affect the estimator API (`callback-mode = 0`).

The default value is 20.

```
<param name="callback-buffer-size" type="Value::Int1" x="20" />
```

tracking-results-buffer-size

Specifies the maximum buffer size for stored tracking results.

The parameter can limit some other parameters, for example, `reid-matching-detections-count`.

```
<param name="tracking-results-buffer-size" type="Value::Int1" x="20" />
```

detector-scaling

Specifies whether frame scaling before detection or foreground subtraction is enabled for performance reasons.

Possible values:

Value	Description
0	Disables frame scaling.
1	Default. Enables frame scaling.

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

scale-result-size

Specifies a size of a frame to be scaled, in pixels, for a detection step, if scaling is enabled.

The default value is 640.

Upper scaling is impossible.

```
<param name="scale-result-size" type="Value::Int1" x="640" />
```

max-detection-count

Specifies the maximum detection count that can be found by one detector call.

The parameter limits performance load. We recommend that you set up a very high value, if you do not want any limits.

The default value is 128.

```
<param name="max-detection-count" type="Value::Int1" x="128" />
```

minimal-track-length

Specifies the minimum detection or re-detection count for a track (see `TrackInfo::detectionsCount`). This count must be met for a track to be returned in tracking results.

The parameter does not apply to human tracking.

The default value is 1. It allows you to get all track data, but there can be short tracks because of detector faults. In this case, you should implement your own logic to filter such tracks.

```
<param name="minimal-track-length" type="Value::Int1" x="1" />
```

tracker-type

Specifies the tracker type to be used.

The parameter does not apply to body tracking.

Possible values:

- kcf
- opencv
- carkalman
- vlTracker
- none

The default value is vlTracker.

```
<param name="tracker-type" type="Value::String" text="" />
```

kill-intersected-detections

Specifies whether to remove intersected detections, if the intersection detection area of two tracks in relation to their total area (IOU, intersection over union) is greater than the value of `kill-intersection-value`.

The parameter applies to face tracking only.

For body and body+face tracking, use the `remove-overlapped-strategy` parameter. Note that setting `remove-overlapped-strategy` to a value other than `none` causes removing detections after a detector. It happens, if the intersection detection area of two tracks in relation to their total area is greater than the value of `kill-intersection-value`. In this case, the tracks are not affected.

Possible values:

Value	Description
0	Disables the parameter.
1	Default. Enables the parameter.

```
<param name="kill-intersected-detections" type="Value::Int1" x="1" />
```

kill-intersection-value

Specifies a number to be compared with the intersection over union size. If the value of `kill-intersection-value` is less than the intersection detection area of two tracks in relation to their total area (IOU), the intersected detections will be removed.

The default value is 0.55.

```
<param name="kill-intersection-value" type="Value::Float1" x="0.55"/>
```

FRG section

frg-subtractor-type

Specifies a type of the foreground subtractor algorithm to be used.

The default value is `MOG`.

```
<param name="frg-subtractor-type" type="Value::String" text="MOG" />
```

use-binary-frg

Specifies whether to use a binary foreground subtractor on CPU.

Possible values:

Value	Description
0	Disables the parameter. Regions are used for detection. The value improves performance on CPU, but degrades tracking accuracy and recall. For GPU, the binary option is always used because it provides better performance.
1	Default. Enables the parameter. If any region is found by the foreground subtractor, a full frame is used for detection.

```
<param name="use-binary-frg" type="Value::Int1" x="1" />
```

frg-update-step

Specifies an update step of the foreground mask measured in frame count. The value should be greater or equal to the detector-step parameter value.

The default value is 20.

```
<param name="frg-update-step" type="Value::Int1" x="20" />
```

frg-scale-size

Specifies the size of the calculated foreground mask and is used to find regions. A higher value increases accuracy, but decreases performance. The value should be lower than the scale-result-size parameter value.

Do not use this parameter for the **MOG** foreground subtractor type. For **MOG** scaling, the value of `scale-result-size` should be 0.25, that is the original frame size, if `detector-scaling` is 0.

The default value is 160.

```
<param name="frg-scale-size" type="Value::Int1" x="160" />
```


frg-regions-alignment

Specifies a value by which regions (rectangles) calculated from the foreground are aligned. The parameter is useful for better tracking quality and recall if use-binary-frg is 0.

The parameter works only for CPU.

The parameter is measured in absolute values. It should be in the range 0..scale-result-size if detector-scaling is 1, and [0..frame_max_side_size] if detector-scaling is 0. frame_max_side_size is the maximum side size of the original frame.

You can find an optimal value empirically. High values provide better recall and tracking accuracy, but can degrade performance. The highest possible value equals to binary-frg. Lower values can degrade accuracy but increase performance.

The default value is 360.

```
<param name="frg-regions-alignment" type="Value::Int1" x="360" />
```

frg-regions-square-alignment

Specifies whether to align foreground regions to a rectangle with equal sides. The rectangle's maximum side is chosen. See also frg-regions-alignment.

Possible values:

Value	Description
0	Disables the foreground region alignment to a square.
1	Default. Enables foreground region alignment to a square.

```
<param name="frg-regions-square-alignment" type="Value::Int1" x="1" />
```

Vehicle section

best-shots-number-for-track

Specifies a number of best shots for tracking.

The default value is 2.

```
<param name="best-shots-number-for-track" type="Value::Int1" x="2" />
```

max-processing-fragments-count

Specifies the maximum number of fragments for a stream being processed in time. Zero value means no limit.

The default value is 1.

```
<param name="max-processing-fragments-count" type="Value::Int1" x="1" />
```

Face tracking specific parameters section

face-landmarks-detection

Specifies whether to enable face landmark detection. Disabling the parameter improves performance.

Possible values:

Value	Description
0	Disables face landmark detection.
1	Default. Enables face landmark detection.

```
<param name="face-landmarks-detection" type="Value::Int1" x="1" />
```

Human/Body tracking specific parameters section

remove-overlapped-strategy

Specifies the strategy to be used for removing after detection or re-detection.

Possible values:

- none
- both
- score

The default value is score.

```
<param name="remove-overlapped-strategy" type="Value::String" text="score" />
```

remove-horizontal-ratio

Specifies a width to height ratio threshold. This threshold is used for removing horizontal detections. The default value is 1.6.

```
<param name="remove-horizontal-ratio" type="Value::Float1" x="1.6"/>
```

iou-connection-threshold

Specifies the intersection over union threshold value. This value is used for matching tracks and detections. The default value is 0.5.

```
<param name="iou-connection-threshold" type="Value::Float1" x="0.5"/>
```

use-body-reid

Specifies whether to use re-identification of body tracks. Enabling the parameter improves accuracy of body tracking, but reduces performance.

Possible values:

Value	Description
0	Disables using of re-identification of body tracks.
1	Default. Enables using of re-identification of body tracks.

```
<param name="use-body-reid" type="Value::Int1" x="1"/>
```

body-reid-version

Specifies a version of a re-identification neural network used for body tracking. The default value is 108.

```
<param name="body-reid-version" type="Value::Int1" x="108"/>
```

reid-matching-threshold

Specifies a re-identification similarity threshold. The threshold is used for matching tracks to each other. The default value is 0.85.

```
<param name="reid-matching-threshold" type="Value::Float1" x="0.85"/>
```

reid-matching-detections-count

Specifies a number of detections that a track must have to be matched by re-identification. The minimum value is 1. The default value is 2.

```
<param name="reid-matching-detections-count" type="Value::Int1" x="2" />
```

inactive-tracks-lifetime

Specifies a lifetime of inactive body tracks, which are used for re-identification. It is measured in number of frames and used for matching tracks to each other.

Higher values lead to better quality of re-identification, but reduce performance.

The default value is 100.

```
<param name="inactive-tracks-lifetime" type="Value::Int1" x="100" />
```

Detectors section

use-face-detector

Specifies whether to use face detection. Possible values:

Value	Description
0	Disables using of face detection.
1	Default. Enables using of face detection.

```
<param name="use-face-detector" type="Value::Int1" x="1" />
```

use-body-detector

Specifies whether to use body detection.

Possible values:

Value	Description
0	Default. Disables using of body detection.
1	Enables using of body detection.

```
<param name="use-body-detector" type="Value::Int1" x="0" />
```

use-vehicle-detector

Specifies whether to use vehicle detection.

Possible values:

Value	Description
0	Default. Disables using of vehicle detection.
1	Enables using of vehicle detection.

```
<param name="use-vehicle-detector" type="Value::Int1" x="0" />
```

use-license-plate-detector

Specifies whether to use license plate detection.

Possible values:

Value	Description
0	Default. Disables using of license plate detection.
1	Enables using of license plate detection.

```
<param name="use-license-plate-detector" type="Value::Int1" x="0" />
```

Experimental parameters section

Parameter described in this section can be removed, renamed, or moved to other sections in future updates.

detect-max-batch-size

Specifies the maximum batch size for detection to limit memory consumption.

The default value is 0 and means there is no limit.

```
<param name="detect-max-batch-size" type="Value::Int1" x="0" />
```

redetect-max-batch-size

Specifies the maximum batch size for re-detection to limit memory consumption.

The default value is 0 and means there is no limit.

```
<param name="redetect-max-batch-size" type="Value::Int1" x="0" />
```

tracker-max-batch-size

Specifies the maximum batch size for a tracker to limit memory consumption.

The default value is 0 and means there is no limit.

```
<param name="tracker-max-batch-size" type="Value::Int1" x="0" />
```

reid-max-batch-size

Specifies the maximum batch size for reID to limit memory consumption.

The default value is 0 and means there is no limit.

```
<param name="reid-max-batch-size" type="Value::Int1" x="0" />
```

min-frames-batch-size

Specifies a number of frames per stream that need to be processed.

Possible value are in range [0, 5].

Value 0 means auto fitting of the parameter. Auto fitting depends on TrackEngine internal implementation, possible value is detector-step.

The parameter is associated with the max-frames-batch-gather-timeout parameter. The two parameters regulate processing latency versus throughput and device utilization.

Higher values lead to higher processing latency. Yet, they can increase throughput and device utilization.

The default value is 1 and means that any available stream frames will be processed.

```
<param name="min-frames-batch-size" type="Value::Int1" x="1" />
```

max-frames-batch-gather-timeout

Specifies the maximum available timeout per stream, in milliseconds, to gather the next frame batch. This timeout starts from the last processing begin time point.

Value 0 means no timeout and equals to min-frames-batch-size = [0-1].

This parameter is associated with min-frames-batch-size. If min-frames-batch-size is not 1, this parameter regulates timeout of batch waiting.

Higher values lead to higher processing latency. Yet, they can increase throughput and device utilization.

The default value is 100.

```
<param name="max-frames-batch-gather-timeout" type="Value::Int1" x="100" />
```

Config example

```
<?xml version="1.0"?>

<settings>
  <section name="logging">
    <param name="mode" type="Value::String" text="l2b" />
    <param name="log-file-path" type="Value::String" text="log.txt" />
    <param name="severity" type="Value::Int1" x="1" />
  </section>

  <section name="other">
    <param name="callback-mode" type="Value::Int1" x="1" />
    <param name="detector-step" type="Value::Int1" x="7" />
    <param name="detector-comparer" type="Value::Int1" x="1" />
    <param name="use-one-detection-mode" type="Value::Int1" x="0" />
    <param name="skip-frames" type="Value::Int1" x="18" />
    <param name="frg-subtractor" type="Value::Int1" x="1" />
    <param name="frames-buffer-size" type="Value::Int1" x="20" />
    <param name="callback-buffer-size" type="Value::Int1" x="20" />
    <param name="tracking-results-buffer-size" type="Value::Int1" x="20" />
    <param name="detector-scaling" type="Value::Int1" x="1" />
    <param name="scale-result-size" type="Value::Int1" x="640" />
    <param name="max-detection-count" type="Value::Int1" x="128" />
    <param name="minimal-track-length" type="Value::Int1" x="1" />
    <param name="tracker-type" type="Value::String" text="" />
    <param name="kill-intersected-detections" type="Value::Int1" x="1" />
    <param name="kill-intersection-value" type="Value::Float1" x="0.55" />
  </section>

  <section name="FRG">
    <param name="frg-subtractor-type" type="Value::String" text="MOG" />
    <param name="use-binary-frg" type="Value::Int1" x="1" />
    <param name="frg-update-step" type="Value::Int1" x="20" />
    <param name="frg-scale-size" type="Value::Int1" x="160" />
    <param name="frg-regions-alignment" type="Value::Int1" x="360" />
    <param name="frg-regions-square-alignment" type="Value::Int1" x="1" />
  </section>

  <section name="vehicle">
```



```

    <param name="best-shots-number-for-track" type="Value::Int1" x="2"
    />
    <param name="max-processing-fragments-count" type="Value::Int1" x="1
    " />
</section>

<section name="face">
    <param name="face-landmarks-detection" type="Value::Int1" x="1" />
</section>

<section name="human">
    <param name="human-landmarks-detection" type="Value::Int1" x="0" />
    <param name="remove-overlapped-strategy" type="Value::String" text="
    score" />
    <param name="remove-horizontal-ratio" type="Value::Float1" x="1.6"/>
    <param name="iou-connection-threshold" type="Value::Float1" x="0.5"
    />
    <param name="use-body-reid" type="Value::Int1" x="1"/>
    <param name="body-reid-version" type="Value::Int1" x="108"/>
    <param name="reid-matching-threshold" type="Value::Float1" x="0.85"/>
    <param name="reid-matching-detections-count" type="Value::Int1" x="2"
    />
    <param name="inactive-tracks-lifetime" type="Value::Int1" x="100" />
</section>

<section name="detectors">
    <param name="use-face-detector" type="Value::Int1" x="1" />
    <param name="use-body-detector" type="Value::Int1" x="0" />
    <param name="use-vehicle-detector" type="Value::Int1" x="0" />
    <param name="use-license-plate-detector" type="Value::Int1" x="0" />
</section>

<section name="experimental">
    <param name="detect-max-batch-size" type="Value::Int1" x="0" />
    <param name="redetect-max-batch-size" type="Value::Int1" x="0" />
    <param name="tracker-max-batch-size" type="Value::Int1" x="0" />
    <param name="reid-max-batch-size" type="Value::Int1" x="0" />
    <param name="min-frames-batch-size" type="Value::Int1" x="1" />
    <param name="max-frames-batch-gather-timeout" type="Value::Int1" x="
    100" />
</section>

<section name="debug">
    <param name="save-debug-info" type="Value::Int1" x="0" />
    <param name="show-profiling-data" type="Value::Int1" x="0" />

```

```

    <param name="save-buffer-log" type="Value::Int1" x="0" />
    <param name="batched-processing" type="Value::Int1" x="1" />
</section>

```

```
</settings>
```

Example

Minimal TrackEngine example.

The example is based on OpenCV library as the easiest and well-known mean of capturing frames from a camera and drawing.

```

// Simple example of TE with opencv cv::VideoCapture used as media player

#include "../inc/tsdk/ITrackEngine.h"
#include <opencv2/highgui.hpp>
#include <opencv2/videoio.hpp>
#include <opencv2/video.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>
#include <map>
#include <thread>
#include <future>

#ifdef WITH_GPU // to build with GPU support or not
#include "cuda_runtime.h"
#endif

// different options
#define USE_ROI false // use ROI of processing

#define IS_REALTIME false // is realtime app (for realtime apps frames
    should be skipped if overload and TE frames-buffer is full)
#define USE_GPU false // use GPU processing

// settings of callbacks, only one of them can be set at time
// if all these options set `false`, then simple per-stream observers are
    used (they're deprecated now, so choose one of these settings)
#define USE_ESTIMATOR_API false // use estimator API, the most flexible way
    for developers to use TE, but requires more code

```

```

        // frame/tracking_result queues, gathering
        batches, etc.
#define USE_BATCHED_OBSERVERS false // use async API and batched observers (
    one observer for all streams and per event type)
#define USE_UNIFIED_OBSERVER true // use async API and one unified observer
    for all tracking events and streams (preferable way)

// used detectors
// if both are enabled then `human` tracking works
#define USE_FACE_DETECTOR true // enable face detector
#define USE_BODY_DETECTOR true // enable body detector

// Optimization tips
// for GPU image cache is preferable way in order to avoid overhead of
    memory allocations
#define USE_IMAGE_CACHE true // reuse images from cache
#define IMAGE_CACHE_SIZE 40 // cache size

#define FLOWER_CACHE_SIZE 1024 // increase flower cache for better
    performance

        // NOTE: on CPU higher cache size slightly
        increases memory consumption

std::map<int,cv::Mat> frameImages;
std::map<int,cv::Mat> bestShotImages;

namespace {
    template<typename Type>
    static fsdk::Span<Type> vectorToSpan(const std::vector<Type> &vec) {
        return fsdk::Span<Type>(const_cast<Type*>(vec.data()), vec.size());
    }
}

/**
 * @brief Image wrapper. needed only for public access to protected method
 * fsdk::Image::getRefCount
 */
class ImageWrapper : public fsdk::Image {
public:
    ImageWrapper() {};

    int getRefCount() const {
        return fsdk::Image::getRefCount();
    }
};

```

```

/**
 * @brief Simple image cache to avoid allocations on GPU for performance
 * reasons
 */
class ImageCache {
public:
    ImageCache(uint32_t size)
        : m_images(size) {

    }

    fsdk::Image get(int width, int height, fsdk::Image::MemoryResidence
memoryResidence) {
        auto it = m_images.begin();

        // find empty or free (ref count == 1) slot
        for (; it != m_images.end(); ++it) {
            if (!it->isValid() ||
                (it->getRefCount() == 1 && width == it->getWidth() &&
                 height == it->getHeight() && it->getMemoryResidence() ==
                 memoryResidence)) {
                break;
            }
        }

        if (it == m_images.end()) {
            return fsdk::Image();
        }

        // if empty, then create new one
        if (!it->isValid()) {
            it->create(width, height, fsdk::Format::R8G8B8, false,
memoryResidence);
        }

        return static_cast<fsdk::Image*>(*it);
    }

private:
    std::vector<ImageWrapper> m_images;
};

#ifdef WITH_GPU
static fsdk::Image getCachedImage(ImageCache &cache, int width, int height,

```

```

fsdk::Image::MemoryResidence memoryResidence) {
    fsdk::Image result = cache.get(width, height, memoryResidence);

    if (!result.isValid()) {
        result.create(width, height, fsdk::Format::R8G8B8, false,
            memoryResidence);
    }

    return result;
}
#endif

struct FrameAdditionalData : tsdk::AdditionalFrameData {
    tsdk::StreamId streamId;
    tsdk::FrameId frameId;

    FrameAdditionalData(tsdk::StreamId streamId, tsdk::FrameId frameId)
        : streamId(streamId)
        , frameId(frameId) {
    }
};

struct SuperObserver :
    tsdk::IBestShotObserver,
    tsdk::IVisualObserver,
    tsdk::IDebugObserver,
    tsdk::IBestShotPredicate,
    tsdk::IVisualPredicate {

    int m_streamId;
    std::map<int, int> m_bestAreas;
    SuperObserver(int streamId) : m_streamId{ streamId } {

    }

    SuperObserver() : m_streamId{} {}

    ~SuperObserver() override = default;

    void bestShot(const tsdk::DetectionDescr& detection, const tsdk::
        AdditionalFrameData* data) override {
        if (detection.image.getMemoryResidence() == fsdk::Image::
            MemoryResidence::MemoryGPU) // for gpu transfer to cpu or use cv
            :GpuMat
            return;

```

```

        // save best shot crop to map
        const cv::Mat cvFrame(detection.image.getHeight(), detection.image.
            getWidth(), CV_8UC3, const_cast<void*>(detection.image.getData())
        );
        const auto rect = detection.detection.getRect();
        bestShotImages[detection.trackId] = cvFrame(cv::Rect(rect.x, rect.y,
            rect.width, rect.height)).clone();
    }

    void trackEnd(const tsdk::TrackId& trackId) override {
        // track with id = 'trackId' finished
    }

    void visual(const tsdk::FrameId &frameId,
        const fsdk::Image &image,
        const tsdk::TrackInfo * trackInfo,
        const int nTrack,
        const tsdk::AdditionalFrameData* data) override {
        if (image.getMemoryResidence() == fsdk::Image::MemoryResidence::
            MemoryGPU) // for gpu transfer to cpu or use cv::GpuMat
            return;

        // convert fsdk::Image to cv::Mat
        const cv::Mat cvFrame(image.getHeight(), image.getWidth(), CV_8UC3,
            const_cast<void*>(image.getData()));
        // save frame to the map
        frameImages[m_streamId] = cvFrame.clone();
        for (int i = 0; i < nTrack; i++) {
            // draw detection rectangle on frame
            cv::putText(frameImages[m_streamId],
                std::to_string(trackInfo[i].trackId),
                cv::Point(trackInfo[i].rect.x + trackInfo[i].rect.
                    width / 2, trackInfo[i].rect.y + trackInfo[i].
                    rect.height / 2),
                cv::FONT_HERSHEY_SIMPLEX,
                1,
                cv::Scalar(10, 200, 10),
                2);
            cv::rectangle(frameImages[m_streamId],
                cv::Rect(trackInfo[i].rect.x,
                    trackInfo[i].rect.y,
                    trackInfo[i].rect.width,
                    trackInfo[i].rect.height),
                trackInfo[i].isDetector ? cv::Scalar(150, 10, 10)

```

```

        : cv::Scalar(10, 10, 150), 2);
    }
}

bool checkBestShot(const tsdk::DetectionDescr& descr, const tsdk::
    AdditionalFrameData* data) override {
    // here can be code of best shot predicate if need
    return true;
}

bool needRGBImage(const tsdk::FrameId frameId, const tsdk::
    AdditionalFrameData*) override {
    return true;
}

// callbacks, mostly, for debug purposes
void debugForegroundSubtraction(const tsdk::FrameId& frameId, const fsdk
    ::Image& firstMask,
    const fsdk::Image& secondMask, fsdk::Rect * regions, int nRegions)
    override {
};

void debugDetection(const tsdk::DetectionDebugInfo& descr) override {
};
};

struct BatchedSuperObserver :
    tsdk::IBatchBestShotObserver,
    tsdk::IBatchVisualObserver,
    tsdk::IBatchDebugObserver {

    BatchedSuperObserver() = default;
    ~BatchedSuperObserver() override = default;

    // here simple realization via per-stream observers (see `SuperObserver
    `) just for demonstration
    void bestShot(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk::
        Span<tsdk::BestShotCallbackData> &data) override {
        for (size_t i = 0; i != streamIDs.size(); ++i) {
            SuperObserver(streamIDs[i]).bestShot(data[i].descr, data[i].
                frameData);
        }
    }
}

void trackStart(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk

```

```

        ::Span<tsdk::TrackStartCallbackData> &data) override {
            for (size_t i = 0; i != streamIDs.size(); ++i) {
                SuperObserver(streamIDs[i]).trackStart(data[i].frameId, data[i].
                    trackId);
            }
        }

void trackEnd(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk::
    Span<tsdk::TrackEndCallbackData> &data) override {
    for (size_t i = 0; i != streamIDs.size(); ++i) {
        SuperObserver(streamIDs[i]).trackEnd(data[i].trackId);
    }
}

void visual(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk::
    Span<tsdk::VisualCallbackData> &data) override {
    for (size_t i = 0; i != streamIDs.size(); ++i) {
        SuperObserver(streamIDs[i]).visual(data[i].frameId, data[i].
            image, data[i].trackInfo, data[i].nTrack, nullptr);
    }
}

void debugForegroundSubtraction(const fsdk::Span<tsdk::StreamId> &
    streamIDs,
                                const fsdk::Span<tsdk::
                                    DebugForegroundSubtractionCallbackData> &
                                    data) override {
    for (size_t i = 0; i != streamIDs.size(); ++i) {
        SuperObserver(streamIDs[i]).debugForegroundSubtraction(data[i].
            frameId, data[i].firstMask, data[i].secondMask, data[i].
            regions, data[i].nRegions);
    }
}

void debugDetection(const fsdk::Span<tsdk::StreamId> &streamIDs,
    const fsdk::Span<tsdk::DebugDetectionCallbackData> &data
        ) override {
    for (size_t i = 0; i != streamIDs.size(); ++i) {
        tsdk::DetectionDebugInfo dbgInfo;
        dbgInfo.descr = data[i].descr;
        dbgInfo.isBestDetection = data[i].isBestDetection;
        dbgInfo.isDetector = data[i].isDetector;
        dbgInfo.isFiltered = data[i].isFiltered;

        SuperObserver(streamIDs[i]).debugDetection(dbgInfo);
    }
}

```



```

    }
}
};

struct TrackingResultObserver : tsdk::ITrackingResultObserver {

    void ready(fsdk::Ref<tsdk::ITrackingResultBatch> result) override {
        // any postprocessing tracking results code here
        if (!result) {
            return;
        }

        // we reuse code of BatchedSuperObserver
        auto streamsCount = result->getStreamIds().size();
        auto streamIds = result->getStreamIds();

        for (size_t streamInd = 0; streamInd != streamsCount; ++streamInd) {
            auto streamFrames = result->getStreamFrameIds(streamIds[
                streamInd]);

            for (auto frameId : streamFrames) {
                auto streamFrameResults = result->getTrackingResult(
                    streamIds[streamInd], frameId);

                auto trackStart = streamFrameResults.trackStart;
                auto trackEnd = streamFrameResults.trackEnd;
                auto tracks = streamFrameResults.tracks;
                auto debugData = streamFrameResults.debugDetections;
                auto debugForegroundSubtractions = streamFrameResults.
                    debugForegroundSubtractions;
                auto detections = streamFrameResults.detections;

                // input and output arrays of ids should be equal
                assert(streamFrameResults.streamId == streamIds[streamInd]);

                if (!debugForegroundSubtractions.empty()) {
                    std::vector<tsdk::StreamId> _streamIds;
                    _streamIds.resize(debugForegroundSubtractions.size(),
                        streamFrameResults.streamId);
                    BatchedSuperObserver().debugForegroundSubtraction(
                        vectorToSpan(_streamIds), debugForegroundSubtractions
                    );
                }

                if (!debugData.empty()) {

```

```

        std::vector<tsdk::StreamId> _streamIds;
        _streamIds.resize(debugData.size(), streamFrameResults.
            streamId);
        BatchedSuperObserver().debugDetection(vectorToSpan(
            _streamIds), debugData);
    }

    if (!detections.empty()) {
        std::vector<tsdk::StreamId> _streamIds;
        _streamIds.resize(detections.size(), streamFrameResults.
            streamId);
        BatchedSuperObserver().bestShot(vectorToSpan(_streamIds)
            , detections);
    }

    if (!tracks.empty()) {
        std::vector<tsdk::StreamId> _streamIds;
        _streamIds.resize(tracks.size(), streamFrameResults.
            streamId);
        BatchedSuperObserver().visual(vectorToSpan(_streamIds),
            tracks);
    }

    if (!trackStart.empty()) {
        std::vector<tsdk::StreamId> _streamIds;
        _streamIds.resize(trackStart.size(), streamFrameResults.
            streamId);
        BatchedSuperObserver().trackStart(vectorToSpan(
            _streamIds), trackStart);
    }

    if (!trackEnd.empty()) {
        std::vector<tsdk::StreamId> _streamIds;
        _streamIds.resize(trackEnd.size(), streamFrameResults.
            streamId);
        BatchedSuperObserver().trackEnd(vectorToSpan(_streamIds)
            , trackEnd);
    }
}

}

}

~TrackingResultObserver() override {};
};

```

```

int main(int argc, char** argv) {
    if (!USE_FACE_DETECTOR && !USE_BODY_DETECTOR) {
        std::cerr << "Both face and body detectors are disabled" << std::
            endl;
        exit(EXIT_FAILURE);
    }

#ifdef WITH_GPU
    if (USE_GPU) {
        std::cerr << "GPU build is off, GPU can't be used." << std::endl;
        exit(EXIT_FAILURE);
    }
#endif

    int streamCount = 1;
    std::vector<ImageCache> streamCaches;
    std::vector<cv::VideoCapture> captures;
    captures.reserve(argc);
    bool usbCam = false;

    if (argc > 1) {
        for (int i = 1; i < argc; i++) {
            cv::VideoCapture capture;
            capture.open(argv[i]);

            if (!capture.isOpened()) {
                //error in opening the video input
                std::cout << "video" << argv[i] << " not opened"<< std::endl
                    ;
                exit(EXIT_FAILURE);
            } else {
                double frameCount = capture.get(cv::CAP_PROP_FRAME_COUNT);
                std::cout << argv[i] << " opened." << frameCount << "frames
                    total" << std::endl;
            }
            captures.emplace_back(std::move(capture));
        }
    } else {
        cv::VideoCapture capture;
        capture.open(0);
        if (!capture.isOpened()) {
            //error in opening the video input
            std::cout << "video from webcam not opened"<< std::endl;
            exit(EXIT_FAILURE);
        }
    }
}

```

```

        usbCam = true;
        captures.emplace_back(std::move(capture));
    }

    streamCount = captures.size();

    if (USE_ESTIMATOR_API) {
        if (streamCount > 1) {
            std::cout << "Estimator API allows to process multiple sources,
                but TE example supports only one source for estimator API now,
                " <<
                " so one Source will be used for tracking of " <<
                streamCount << " Streams." << std::endl;
        }
    }

    // create FaceEngine and then TrackEngine objects
    fsdk::ISettingsProviderPtr config = fsdk::createSettingsProvider("./data/
        /faceengine.conf").getValue();
    auto faceEngine = fsdk::createFaceEngine("./data/").getValue();
    faceEngine->setSettingsProvider(config);

    if(!fsdk::activateLicense(faceEngine->getLicense(), "./data/license.conf
        ")) {
        return 1;
    }

    auto runtimeSettings = faceEngine->getRuntimeSettingsProvider();

    fsdk::ISettingsProviderPtr configTE = fsdk::createSettingsProvider("./
        data/trackengine.conf").getValue();
    configTE->setValue("detectors", "use-face-detector", USE_FACE_DETECTOR);
    configTE->setValue("detectors", "use-body-detector", USE_BODY_DETECTOR);

    // enable vlTracker, if there are many streams, because it's intended
    for multiple streams processing
    if (streamCount > 1 || USE_GPU) { // WARN! gpu supports only 'vlTracker'
        or 'none' tracker
        configTE->setValue("other", "tracker-type", "vlTracker");
    }

    // set binary FRG for GPU and disable for CPU for max perf
    configTE->setValue("FRG", "use-binary-frg", USE_GPU ? 1 : 0);

    if (USE_ESTIMATOR_API) {

```

```

        configTE->setValue("other", "callback-mode", 0); // must set 0 for
        estimator API
    }

    if (USE_GPU) {
        // NOTE: for GPU also valid parameters from runtime config must be
        // set:
        // "Runtime":"defaultGpuDevice" to actual used GPU number, "Runtime"
        //:"deviceClass" to "gpu".
        // it can be changed in the config file or here from runtime
        // settings provider
        if (runtimeSettings->getValue("Runtime", "defaultGpuDevice").asInt
            (-1) == -1) {
            runtimeSettings->setValue("Runtime", "defaultGpuDevice", 0);
        }

        runtimeSettings->setValue("Runtime", "deviceClass", "gpu");
    }

    runtimeSettings->setValue("Runtime", "programCacheSize",
        FLOWER_CACHE_SIZE);

    auto trackEngine = tsdk::createTrackEngine(faceEngine, configTE).
        getValue();

    std::vector<fsdk::Ref<tsdk::IStream>> streamsList;
    std::vector<SuperObserver> observers(streamCount);
    std::vector<std::future<void>> threads;

    TrackingResultObserver trackingResultObserver;
    BatchedSuperObserver batchedSuperObserver;

    if (USE_ESTIMATOR_API) {
        // for estimator API callback isn't used
    }
    else { // else set callback(s)
        if (USE_UNIFIED_OBSERVER) {
            trackEngine->setTrackingResultObserver(&trackingResultObserver);
        }
        else if (USE_BATCHED_OBSERVERS) {
            // set batched callbacks
            trackEngine->setBatchBestShotObserver(&batchedSuperObserver);
            trackEngine->setBatchVisualObserver(&batchedSuperObserver);
            trackEngine->setBatchDebugObserver(&batchedSuperObserver);
        }
    }

```

```

}

streamCaches.resize(streamCount, IMAGE_CACHE_SIZE);
std::atomic<bool> stop{ false };

auto threadFunc = [&trackEngine, &captures, &streamsList, &streamCaches,
    &stop, usbCam](int streamInd) {
    uint32_t index = 0;
    auto& capture = captures[streamInd];
    cv::Mat frame; //current frame

    if (capture.isOpened()) {
        while (!stop && capture.read(frame)) {
            if (!usbCam)
                index = static_cast<int>(capture.get(cv::
                    CAP_PROP_POS_FRAMES));
            else
                index++;

            if (!frame.empty()) {
                const fsdk::Image cvImageCPUWrapper(frame.cols, frame.
                    rows, fsdk::Format::R8G8B8, frame.data, false); // no
                    copy, just wrapper

                fsdk::Image image;

#ifdef WITH_GPU
                if (USE_GPU) {
                    if (USE_IMAGE_CACHE) {
                        fsdk::Image cachedImage = getCachedImage(
                            streamCaches[streamInd], frame.cols, frame.
                                rows, fsdk::Image::MemoryResidence::MemoryGPU
                            );

                        cudaMemcpy(const_cast<void*>(cachedImage.getData
                            ()), const_cast<void*>(cvImageCPUWrapper.
                                getData()),
                            cvImageCPUWrapper.getDataSize(),
                                cudaMemcpyHostToDevice);
                    }
                    else {
                        image.create(cvImageCPUWrapper, fsdk::Image::
                            MemoryResidence::MemoryGPU);
                    }
                }
            }
            else

```

```

#endif

    {
        image = cvImageCPUWrapper.clone(); // clone because
        TE internally keeps last frame image for tracks
        data

        // performance
        // overhead is
        // possible
        // otherwise
    }

    if (USE_ESTIMATOR_API) {
        // here we track the same stream/image in batch of
        // size `streamCount`
        // just to demonstrate estimator API using, in real
        // case ofc different streams can be processed
        // in order to do that, some code should be written
        // for gathering batch of frames from different
        // streams
        // and calling `track` with that batch in another
        // thread
        // the best approach is to use one thread loop with
        // `track` per each TE object created
        // NOTE: `track` is thread safe (blocking call)
        std::vector<tsdk::StreamId> streamIds;
        std::vector<tsdk::Frame> frames;

        for (size_t i = 0; i != streamsList.size(); ++i) {
            streamIds.emplace_back(streamsList[i]->getId());
            frames.emplace_back();
            frames.back().image = image;
            frames.back().frameId = index;
            frames.back().userData = new FrameAdditionalData
                (streamsList[i]->getId(), index);
        }

        const auto validateRes = trackEngine->validate(fsdk
            ::Span<tsdk::StreamId>(streamIds), fsdk::Span<
            tsdk::Frame>(frames));

        if (!validateRes) {
            std::cerr << "Wrong input for `track`" << std::
                endl;
        }
        else {

```

```

        try {
            auto result = trackEngine->track(fsdk::Span<
                tsdk::StreamId>(streamIds), fsdk::Span<
                tsdk::Frame>(frames));
            if (result)
                TrackingResultObserver().ready(result.
                    getValue());
        }
        catch (const std::exception &e) {
            std::cerr << "`Track` exception: " << std::
                string(e.what()) << std::endl;
        }
    }
}
else {
    tsdk::Frame frame;
    frame.image = image;
    frame.frameId = index;
    frame.userData = nullptr;

    if (!streamsList[streamInd]->validateFrame(frame))
        std::cerr << "Wrong input frame " << index << "
            for `pushFrame*`, Stream with index: " <<
                streamInd << std::endl;
    else {
        const bool pushFrameRes = IS_REALTIME ?
            streamsList[streamInd]->pushFrame(frame) :
            streamsList[streamInd]->pushFrameWaitFor(
                frame, std::numeric_limits<uint32_t>::max
                    ());

        if (!pushFrameRes) {
            std::cerr << "Failed to push frame: " <<
                index << " for Stream with index: " <<
                streamInd << std::endl;
        }
    }
}
}

if (index % 1000 == 0) {
    if (!usbCam) {
        const double frameCount = capture.get(cv::
            CAP_PROP_FRAME_COUNT);
        const double framePos = capture.get(cv::

```



```

        CAP_PROP_POS_FRAMES);
        std::cout << "Stream " << streamInd << " progress:"
            << (framePos / frameCount) * 100.0 << "%"
            << std::endl;
    } else {
        std::cout << "Stream " << streamInd << " progress:"
            << index << " frames" << std::endl;
    }
}
std::cout << "Stream " << streamInd << " ended" << std::endl;
capture.release();
}
else {
    std::cout << "Stream " << streamInd << " is not opened" << std::
        endl;
}
};

// ROI feature
tsdk::StreamParamsOpt streamParamsOpt;
if (USE_ROI) {
    // detect tracks only on bottom half of the frame
    streamParamsOpt.humanRelativeROI = fsdk::FloatRect(0.0f, 0.5f, 1.0f,
        0.5f); // x, y, width, height
}

// create streams
int observerIndex = 0;
for (int i = 0; i < streamCount; i++) {
    observers[observerIndex].m_streamId = observerIndex;
    fsdk::Ref<tsdk::IStream> stream = USE_ROI ?
        fsdk::acquire(trackEngine->createStreamWithParams(
            streamParamsOpt)) : fsdk::acquire(trackEngine->createStream(
        ));

    if (!USE_BATCHED_OBSERVERS && !USE_UNIFIED_OBSERVER && !
        USE_ESTIMATOR_API) {
        // set per-stream callbacks
        stream->setBestShotObserver(&observers[observerIndex]);
        stream->setVisualObserver(&observers[observerIndex]);
        stream->setDebugObserver(&observers[observerIndex]);
    }

    // always per-stream predicates

```

```

        // NOTE: here we use "super" observers just to simplify code,
        // actually, separate vector of predicates should be created
        stream->setBestShotPredicate(&observers[observerIndex]);
        stream->setVisualPredicate(&observers[observerIndex]);

        // by default all observers are enabled, this is just demonstration
        // of API using
        stream->setObserverEnabled(tsdk::StreamObserverType::SOT_BEST_SHOT,
            true);
        stream->setObserverEnabled(tsdk::StreamObserverType::SOT_VISUAL,
            true);
        stream->setObserverEnabled(tsdk::StreamObserverType::SOT_DEBUG, true
        );

        streamsList.emplace_back(stream);

        if (!USE_ESTIMATOR_API)
            threads.emplace_back(std::async(std::launch::async, threadFunc,
                i));

        std::cout << "Stream " << i << " started" << std::endl;
        observerIndex++;
    }

    // for estimator API create only one source
    if (USE_ESTIMATOR_API)
        threads.emplace_back(std::async(std::launch::async, threadFunc, 0));

    while (true) {
        bool notFinished = false;

        for (auto &thread : threads) {
            if (thread.wait_for(std::chrono::milliseconds(10)) == std::
                future_status::timeout)
                notFinished = true;
        }
        if (!notFinished)
            break;
    }

    // it's recommended to `join` each stream before and to stop TE before
    // TE object release
    for (auto &stream : streamsList) {
        if (USE_ESTIMATOR_API) {
            auto remainingResults = stream->stop(); // returns all remaining

```

```

        tracking events

        // for face tracking it's
        // only `trackEnd` for
        // remaining tracks
        // for body/human also
        // events for remaining
        // frames
        // for body we may get
        // tracking results with
        // delay, so TE may
        // still keep several
        // last frames and their
        // events
        // note `Receiving
        // tracking results`
        // section in docs.

        if (remainingResults)
            TrackingResultObserver().ready(remainingResults);
    } else
        stream->join(); // wait all queued frames/callbacks to be
                        // processed
    }

    trackEngine->stop();
}

```