TrackEngine Handbook

Contents

Introduction

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

> Note, that TrackEngine itself does not perform any facial recognition. It's purpose is to **prepare** required data for external systems, like VisionLabs LUNA Platform.

Glossary

- **Track** - Information on face position of a single person on a frame sequence.
- **Tracking** - Function that follows an object (face) through a frame sequence.
- **Best shot** - Image suitable for facial recognition.

Working with TrackEngine

TrackEngine is based on face detection and analysis methods provided by FaceEngine library. This document does not cover FaceEngine usage in detail, for more information please see FaceEngine_Handbook.pdf.

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 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 TrackEngine configuration
    - *vehicleEngine* - pointer to the VehicleEngine object (if with vehicle logic)
    - *launchOptions* - launch options for sdk functions
    - *return value* - pointer to ITrackEngine

It is not recommended to create multiple TrackEngine instances in one application, unless different configs are required. The reason is that batching improves performance.

The main interface to TrackEngine is Stream - an entity to which you submit video frames. To create a stream use the following TrackEngine method

- __IStream* ITrackEngine::createStream(StreamParams *params = nullptr)__

    - *params* - the pointer to stream specific parameters. It's optional parameter, if valid, then it overrides config params for the Stream. Consider `StreamParams` for details.
    - *return value* - pointer to IStream

- **IStream* ITrackEngine::createStreamWithParams(const StreamParamsOpt &params)**

    - *params* - stream specific parameters. Each parameter in the struct is optional, so if it's valid, then overrides the same config parameter for the Stream. Consider `StreamParamsOpt` for details.
    - *return value* - pointer to IStream

Note: User must own this raw pointer by fsdk::Ref, e.g. with fsdk::acquire and reset all refs to all streams before Track Engine object desctruction, otherwise memory leak or/and UB are guaranteed. This is valuable especially in languages where order of objects desctruction is not guaranteed, so users should manage objects lifetime manually (e.g. `python`). See examples.

Users can create multiple streams working concurrently (in case when need to track faces from multiple 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 either due to people leaving the visible area or due to the challenging detection conditions (poor image quality, occlusions, extreme head poses, etc).

There are two ways to work with TE. First one is async pushFrame/callbacks method (See `IStream::pushFrame`), which allows users to use simple API with async push frames per each Stream and get all tracking result events/data in another thread in callbacks. The second one is more complex, but flexible for developers. It's estimator API (See `ITrackEngine::track`), that works like SDK estimator with internal state, so users should pass batch of streams/frames to function and get ready tracking results for input streams.

Each of the methods has pros and cons. Main advantage of async method is simplicity of client side code, so users mostly don't have to deal with exceptions handling, multithreading issues and creating queues for multiple stream batches gathering and results deferred processing. All that logic is implemented in TE for `callback-mode = 1`. The common solution is to create a stream per each frame source, setup callbacks observers, and submit frames to each stream one by one basis. Frames can be pushed to each stream from different threads, depending on architecture of application, but it's recommended to use one thread per each stream and frame source. Tracking results are obtained in the callbacks from another thread (it's created in the TE), so it's the place where users should write logic of processing results. When Stream has to be finished, user should call `IStream::join` method to wait all queued frames/callbacks to be processed. After that Stream can't be used anymore and can be released. If users want to control logic of tracking maximum, they should use estimator tracking API. One of the key

advantages of estimator API is minimal memory consumption of Track Engine (so possibility to achieve better performance), because in this case it doesn't keep images in any queues (images are kept in the tracks data still, though). When users work with estimator tracking API, they don't have to deal with many of the config parameters, regulating any buffer sizes or batching settings, e.g `tracking-results -buffer-max-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 Stream serves only as a state object for tracking.

> Note: To use estimator API, users should set config parameter `callback-mode` to 0, otherwise value 1 must be set (default value is 1).

At the end of work with TE, users should call `ITrackEngine::stop` before TE 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 new frame per each stream and returns ready tracking results data for passed streams (as callbacks compatible data).

  – *streams* - streams stream identifers, must contain only unique id-s, otherwise function throws. See `IStream::getId`.
  – *frames* - frames input frames per stream. See also `IStream::pushFrame` and `Frame`.
  – *return value* - First error code and Ref to ready tracking results as callbacks compatible data. Consider `ITrackingResultBatch`.

It's recommended to make each frame (from `frames`) image to be owner of data, otherwise performance overhead is possible as TE internally will clone it to keep in track data. 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 current passed frame. The reason of such delay is that, generally, tracking may require several frames to get results. See `Receiving tracking results` section for more details. It's thread safe, but blocking call. The function isn't exception safe like `pushFrame`.

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

For prevalidation of track inputs `noexcept` function `validate` is useful.

- **fsdk::Result validate(fsdk::Span streams, fsdk::Span frames, fsdk::Span<fsdk::Result> outErrors)** Validate input of multiple streams/frames in a single function call.

  – *streams* - streams stream identifers array.
  – *frames* - frames input frames per stream.
  – *outErrors* - errors output span of errors for each stream/frame.

– *return value* - Result with last error code.

When Stream has to be finished, then user should call IStream `stop` method before Stream release to get all remained tracking results. Stream can be used for tracking after call of this func.

- **fsdk::Ref stop(bool reset = false)** Finishes all current tracks and returns all remaining tracking results.
    - *reset* - reset if set true, then function resets Stream's state to initial (otherwise keep internal frame counter, statistics etc)
    - *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* - input frame image. Format must be R8G8B8 OR R8G8B8X8.
    - *frameId* - unique identifier for frames sequence.
    - *data* - is any additional data that a developer wants to receive in callbacks-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 param.
    - *return value* - true if frame was appended to the queue for processing, false otherwise - frame was skipped because of full queue.

It's recommended to make `frame` to be owner of data, otherwise performance overhead is possible as TE internally will clone it to keep in track data. Also there are some variations of this method: `pushCustomFrame`, `pushFrameWaitFor`, `pushCustomFrameWaitFor`.

When Stream has to be finished, then user should call IStream `join` method before stream release. Stream can't be used after join for processing, only "getter" functions are available.

- **void IStream::join()** Blocks current thread until all frames in this Stream will be handled and all callbacks will be executed.

Note: Ignoring this step can lead to unexpected behavior (TE writes warning log in this case).

You can set up an observer to receive and react to events. There are two types of observers: per-stream specific single observer and batched observer for all streams. Per-stream observers are deprecated now and remained only for compatibility with old versions.

Note: It's highly recommended to use new batched observers API instead of old per-stream one.

Batched observers have some advantages over per-stream observers:

- reduce and set fixed number of threads created by TrackEngine itself (see section **Threading** for details).
- eliminate performance overhead from multiple concurrently working threads used for per-stream callbacks.

- allow to easily use batched SDK API without additional aggregation of data from single callbacks. Both for GPU/CPU batched SDK API improves performance (for GPU effect is much more significant).
- give more information in output (per-stream callbacks functions signatures remain the same because of compatibility with old versions)

Stream observer interfaces:

Per-stream observers:

- IBestShotObserver

- IVisualObserver

- IDebugObserver

Batched event specific observers:

- IBatchBestShotObserver

- IBatchVisualObserver

- IBatchDebugObserver

Batched unified observer:

- ITrackingResultObserver

`ITrackingResultObserver` is the most recommended observer to work with, when `callback-mode` = 1, as it provides all ready tracking results/events in one struct/callback. This observer contains only one function `ready` which called every time, when tracking results are ready for any streams/frames. Users have assurance that all frames per each stream from `ITrackingResultBatch` were processed and can write logic based on this knowledge. `ready` function will be called once per each stream/frame except of last frame of each stream, when `trackEnd` is called for all remaining tracks (inactive too) on Stream `join`, because TE can't know which frame actually will be last from user. Note, that it's value type is the same as for `track` method (used for `callback-mode` = 0).

There is the next priority of observers use:

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.

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

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

Setting per-stream observer API example:

- **void IStream::setBestShotObserver(tsdk::IBestShotObserver* observer)** Sets a best shot observer for the Stream.

  - *observer* - pointer to the observer object, see `IBestShotObserver`. Don't set to nullptr, if you want disable it, then use `IStream::setObserverEnabled` with false.

Setting batched observer API example:

- __void ITrackEngine::setBatchBestShotObserver(tsdk::IBatchBestShotObserver *observer)__ Sets a best shot observer for all streams.

  - *observer* - pointer to the batched observer object, see `IBatchBestShotObserver`. Don't set to nullptr, if you want disable it, then use `IStream::setObserverEnabled` with false.

Setting unified batched observer API:

- __void ITrackEngine::setTrackingResultObserver(tsdk::ITrackingResultObserver *observer)__ Sets a unified tracking result observer for all streams.

  - *observer* - pointer to the observer object, see `ITrackingResultObserver`. If set to nullptr, then per event batched observers will be used.

Observers

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 '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;

#ifndef 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)** tells that the track with `trackId` has ended and no more best shots should be expected from it. Default implementation does nothing.

    – *trackId* - id of the 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 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

```cpp
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;
};
```

- *nTrack* - number of tracks

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* - frame id of foreground
  – *firstMask* - result of background subtraction operation
  – *secondMask* - result of background subtraction operation after procedures of erosion and dilation
  – *regions* - regions obtained after 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, e.g. `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 original image (or rgb image for 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 streams id
  - *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 streams id
  - *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 streams id
  - *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 streams id
    - *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 streams id
    - *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

```
/* @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;
```

```cpp
    //! 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;
};
```

```cpp
/** @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(tsdk::StreamId streamId, tsdk::
        FrameId frameId) const = 0;
};
```

- **void IStream::setObserverEnabled(tsdk::StreamObserverType type, bool enabled)** Enables
  or disables observer.

- *type* - type of observer
- *enabled* - flag to enable/disable observer

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

Track lifetime

All tracks live until they meet the specific conditions of tracking algorithm (e.g. out of frame bounds or `skip-frames` logic). Human/Body tracking algorithm has its own rules for tracks lifetime (see section `Body tracking algorithm`). TrackEndReason implies reason of track finishing.

Body tracking algorithm

Body tracking algorithm differs from the faces one. Tracker feature isn't used at all, only detect/redetect are used. For matching tracks with new detections IOU metrics is used as well. The parameter `human:iou -connection-threshold` is used for threshold. For better tracking accuracy the `ReIdentification` feature is used to merge different tracks of one human (for `ReIdentification` details see the next section).

For face tracking algorithm when detect/redetect fails, then track is updated with 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 don't participate in common tracking processing except of `ReId`.

Note, that the parameter `skip-frames` doesn't affect on body tracking algorithm. Body tracks are finished according to their own logic. Now there is only one case, when `trackEnd` is called for body track (see `TrackEndCallbackData` reasonfield): inactive track is finished by timeout set by config parameter `human:inactive-tracks-lifetime`. Value of `inactive-tracks-lifetime` should be greater than `reid-matching-detections-count`.

Some algorithm notes and parameters relation. After detect/redetect all found detections are filtered by some conditions: - Overlapped detections may be removed. For overlapping estimation IOU metric is used. If IOU is higher than threshold parameter `other:kill-intersection-value`, 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, used for removing horizontal detections.

Human tracking algorithm

Human tracking algorithm is derived from face and body tracking algorithms and based on `HumanFaceDetector`.

ReIdentification

`ReIdentification` is a optional feature, that improves tracking accuracy (config parameter `use-body-reid`). `ReIdentification` is intended to solve problem, described in section `Body tracking algorithm`. It matches two different tracks if need and merges them into one track with 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 then "reid-matching-threshold".

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

Receiving tracking results

As mentioned earlier, tracking results for specific frame, may be calculated only after several more frames submitted to TE (for both `callback-mode` types). The reason of such delay is that, generally, 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` is used to regulate max size of this buffer, so users have a guarantee, that only this count of frames are stored internally. It limits all other config parameters, which can affect on how many frames are required to calculate tracking results (e.g. `reid-matching-detections-count`). Required frames count to return tracking results for one frame: 1. 1 for Face tracking or if `ReId` feature is disabled for Body/Human tracking. 2. maximum `reid-matching-detections-count` if `ReId` feature is enabled for Body/Human tracking. This is maximum value, actually results can be ready earlier.

When `callback-mode` = 1, logic of buffered tracking results just adds small latency between frame is pushed and it's tracking results are ready from any callback. For `callback-mode` = 0, users should expect that, generally, `track` may not return any results for the Stream until required count of frames been passed to the Stream.

Memory consumption

TrackEngine itself doesn't allocate much memory for internal calculations, but it permanently holds images in the current tracks data (actually, it holds one image per each Stream) and, additionally, in frame/callback queues for `callback-mode` = 1. The main tips to reduce memory consumption is to set `frames-buffer-size`, `callback-buffer-size` and `skip-frames` low enough. To achieve high

optimized minimum memory consumption solution users should use estimator API `ITrackEngine::track` and don't keep images in any queues or minimize that in maximum.

Threading

TrackEngine is multi-threaded. Count of threads is configurable and depends on the currently bound `FaceEngine` settings and type of observers been used (batched or single). TrackEngine calls Observers functions in separate threads. If batched observers are used, then only one additional thread will be created and used for all batched callbacks and all streams. If per-stream single observers are used, then for each stream it's own separate callback thread will be created and used for it's callbacks invocations. In this case all callbacks are invoked from the one thread per-stream. Whatever callback type is used, it is recommended to avoid long-time running tasks in these functions, because pushing to callback buffer blocks main processing thread, so main processing thread always waits until there is free slot in that buffer to push a callback (buffer's size is set by parameter **callback-buffer-size**, see below). The `checkBestShot` and `needRGBImage` functions are called in the main frame processing thread. It is also recommended to avoid expensive computations in these functions. Perfectly, these predicates should take zero performance cost.

Threads count guarantees (excluding calculating threads of SDK): - If batched observers are used, then users have guarantee, that TrackEngine uses only 2-3 threads itself. - If per-stream single observers are used, then users have guarantee, that TrackEngine uses only 1-2 + *number of created streams* threads itself.

Tracker

TrackEngine uses tracker to update the current detections in the case of detect/redetect fail. TrackEngine supports several trackers (see `tracker-type` parameter in the config, section `Settings`). Some platforms don't support all trackers. `vlTracker` is the tracker based on neural networks. It's the only tracker, that can be used for GPU/NPU processing (other trackers, except of `none`, don't support GPU/NPU) and for processing concurrently running multiple streams (it has batching implementation, so provides better CPU utilization). `KCF/opencv` trackers are simple CPU trackers, that should be used only in case of few tracks in total for all streams at the moment. `None` tracker choosen disables tracking feature at all, so it leads to better performance, but degradation of tracking quality.

ROI

TrackEngine supports ROI of tracking: it can be set only with per Stream parameter `humanRelativeROI` from `tsdk::StreamParams/tsdk::StreamParamsOpt`. Note, that it's relative ROI, so it sets rect as relative of frame size. If ROI is set, then detector finds faces/humans only in that area, while tracker can move tracks outside of ROI for several frames (maximum `detector-step` frames). If `detector-scaling` is 1 then TE extracts ROI and makes scaling as one operation (firstly extracting ROI and

then scaling), so there isn't any overhead on extracting ROI. There's common rule to achieve better performance: find frame size ratio (width / height) for most Streams without ROI and set ROI-s for different streams with equal ratio width / height. The most optimal case implies all Streams within one application instance have equal ratio of universal ROI (`humanRelativeROI` or original frame size if ROI isn't set) and `detector-scaling` is 1. Consider TE example of code to work with ROI at the end of doc.

> Note: It's recommended to use this feature with `detector-scaling` set 1 instead of extracting ROI of original frame before `pushFrame*`.

Settings

TrackEngine config format is similar to FaceEngine's. See FaceEngine_Handbook.pdf for format details.

Logging section

- **mode** - logging mode. possible values:
  - *l2c* - log to console only
  - *l2f* - log to file
  - *l2b* - log to console and file. This is the default.
- **severity** - logging severity level. 0 - write all information .. 2 - errors only. `1` by default.

Other sections

- **use-one-detection-mode** - if value is equal to 1, then only one "best" track will be tracked. `0` by default.
- **callback-mode** - If value set to 1, then async push/callback mode should be used, estimator tracking API should be used otherwise. `1` by default.
- **detector-comparer** - the parameter goes with **use-one-detection-mode** and if that is equal to 1, then this parameter sets strategy to find best track on the frame. See config for more details. `1` by default.
- **detector-step** - Number of frames between full face detections. The lower the number is, the more likely TrackEngine is to detect a new face as soon as it appears. The higher the number, the higher the overall performance. It is used to balance between computation performance and face detection recall. `7` by default.
- **skip-frames** - If track wasn't updated by detect/redetect for this number of frames, then track is finished. very high values may lead to performance degradation. Parameter doesn't affect on body/human tracking. `36` by default.
- **frg-subtractor** - Whether to enable foreground subtractor or not. This feature can drastically improve performance, especially, on sources with low level activity, but at the same time this may reduce face detection recall in rare cases. `1` by default.

- **frames-buffer-size** - Size of the internal storage buffer for the input frames. Applied **per stream**. The bigger the buffer is, the more frames are preserved and less likely to be skipped, if detection performance is not high enough to keep up with the frame submission rate. However, increasing this value also increases RAM/VRAM consumption dramatically. It is used to balance between resource utilization and face detection recall. `20` by default.
- **callback-buffer-size** - The size of the internal storage buffer for all callbacks. The larger the buffer is, the higher performance is ensured, but memory consumption may be higher. `20` by default.
- **max-detection-count** - Maximum detections count could be found by one detector call. Parameter limits performance load. If you don't want any limits, just set up very high value. `20` by default.
- **minimal-track-length** - Minimum detections (detect/redetect) count for track (see `TrackInfo::detectionsCount`) to return it in tracking results (parameter is ignored for body/human tracking). Default value 1 allows user to get all tracks data, but there can be short tracks, because of detector faults, so users should implement their own logic to filter such tracks, `1` by default.
- **detector-scaling** - Do scaling frame before detection for performance reasons. `1` by default.
- **scale-result-size** - If scaling is enabled, frame will be scaled to this size in pixels (by the max dimension - width or height). `640` by default.
- **tracker-type** - Type of tracker to use (not used for body tracking), `kcf` by default.

Face tracking specific parameters section

- **face-landmarks-detection** - Flag to enable face landmarks detection. Disabling it improves performance. `1` by default.

Human/Body tracking specific parameters section

- **human-landmarks-detection** - Flag to enable body landmarks detection. Disabling it improves performance. `1` by default.
- **remove-overlapped-strategy** - strategy, used for removing overlapped detections after (re)detect ["none", "both", "score"]. "score" by default.
- **remove-horizontal-ratio** - width to height ratio threshold, used for removing horizontal detections. "1.6" by default.
- **iou-connection-threshold** - IOU value threshold, used for matching tracks and detections. `0.5` by default.
- **reid-matching-threshold** - reID value threshold (similarity), used for matching tracks to each other. `0.85` by default.

Detectors section

- **use-face-detector** - Flag to use or not face detection. `1` by default.
- **use-body-detector** - Flag to use or not body detection. `0` by default.

- **use-vehicle-detector** - Flag to use or not vehicle detection. `0` by default.
- **use-license-plate-detector** - Flag to use or not license plate detection. `0` by default.

For full parameters set with descriptions see trackengine.conf file in the `data` directory.

Config example

```xml
<?xml version="1.0"?>
<settings>
    <section name="logging">
        <param name="mode" type="Value::String" text="l2b" />
        <param name="severity" type="Value::Int1" x="1" />
    </section>

    <section name="other">
        <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="36" />
        <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="min-frames-batch-size" type="Value::Int1" x="0" />
        <param name="max-frames-batch-gather-timeout" type="Value::Int1" x="
            0" />
        <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="20" />
        <param name="minimal-track-length" type="Value::Int1" x="1" />
        <param name="tracker-type" type="Value::String" text="vlTracker" />
        <param name="kill-intersected-detections" type="Value::Int1" x="1"
            />
        <param name="kill-intersection-value" type="Value::Float1" x="0.55"
            />
    </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="1" />
        <param name="remove-overlapped-strategy" type="Value::String" text="
            score" />
        <param name="remove-horizontal-ratio" type="Value::Float1" x="1.6"/>
```

```xml
            <param name="iou-connection-threshold" type="Value::Float1" x="0.5"
                />
            <param name="reid-matching-threshold" type="Value::Float1" x="0.85"
                />
            <param name="inactive-tracks-lifetime" type="Value::Int1" x="100" />
            <param name="reid-matching-detections-count" type="Value::Int1" x="7
                " />
        </section>

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

```cpp
// 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>
```

```cpp
#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 slighlty
                                    increases memory consumption
```

```cpp
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 peformance
    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;
    };

    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;
    }

    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,
```

```cpp
    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
```

```cpp
        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 (size_t 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 {
    };
```

```cpp
    };

    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,
```

```cpp
                            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 resuls 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()) {
```

```cpp
                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);
    }

#ifndef 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);
    const std::chrono::high_resolution_clock::time_point start = std::chrono
        ::high_resolution_clock::now();
    bool usbCam = false;

    if (argc > 1) {
        for (int i = 1; i < argc; i++) {
```

```cpp
        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 estmator 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);
```

```cpp
    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;
```

```cpp
            for (int 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) :
```

```cpp
                        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
}
```

```cpp
        // 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();
}
```