TrackEngine Handbook

Contents

Introduction

TrackEngine is a tool for face detection and tracking on multiple sources. It allows to pick the most suitable still images for facial recognition from a sequence of video frames.

> 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.

In the end of processing user must call ITrackEngine `stop` method.

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

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

> 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 detector in SDK, so users should pass batch of streams/frames to function and get tracking results for all 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 simple case implies one separate thread per each stream and frame source. Tracking results are obtained in the callbacks (batched or single) in another thread, so it's the place where users should write logic of processing results. If users want to control logic of tracking in 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 "frames-buffer-size", "callback-buffer-size", "min-frames-batch-size", "max-frames-batch-gather-timeout". Also, streams shouldn't be `joined` in the end. In this case Stream serves only as a state object for source 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).

Estimator API:

- **fsdk::Ref track(fsdk::Span streams, fsdk::Span frames)** Updates stream tracks by new frame per each stream and returns tracking result batch as callbacks compatible data.

    - *streams* - streams stream identifers, must contain only unique id-s, otherwise undefined behavior. See `IStream::getId`.
    - *frames* - frames input frames per stream. See also `IStream::pushFrame` and `Frame`.
    - *return value* - Ref to tracking result with callbacks arguments data. Consider `ITrackingResultBatch`.

It's not thread safe now, so concurrent calls aren't allowed, otherwise undefined behavior. 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 `validate` non-throwing function 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.

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.

Also there are some variations of this method: `pushCustomFrame`, `pushFrameWaitFor`, `pushCustomFrameWaitFo` .

TrackEngine emits various events to inform you what is happening. The events occur on a per-stream basis.

When Stream has to be finished, in `callback-mode` user must call IStream `join` method before stream destruction. Stream shouldn'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 set deprecated now remain 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)

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

Stream observer interfaces:

Per-stream observers:

- IBestShotObserver

- IVisualObserver

- IDebugObserver

Batched observers:

- IBatchBestShotObserver

- IBatchVisualObserver

- IBatchDebugObserver

By implementing one or several observer interfaces it is possible to define custom processing logic in your application.

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.

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;
```

```cpp
#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;

    //! extra detections
    // needed when detectionsFlags has combination type,
    // e.g. for EDetection_Body_Face extraDetections[EDetection_Face],
        extraDetections[EDetection_Body] are valid
    // note: for simple detection type extra detection with corresponding
        index is valid too
    fsdk::Detection extraDetections[EDetectionObject::
        EDetection_Simple_Count];

    bool hasDetectionFlag(EDetectionObject obj) {
         return (detectionsFlags & (1 << obj)) ? true : false;
    }

    void setDetectionFlag(EDetectionObject obj, bool enable) {
        if (enable) {
            detectionsFlags |= (1 << obj);
        }
        else {
            detectionsFlags &= ~(1 << obj);
        }
    }
```

```
    void setExtraDetection(EDetectionObject obj, const fsdk::Detection &
        detection) {
        extraDetections[obj] = 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

- **void trackStatusUpdate(tsdk::FrameId frameId, tsdk::TrackId trackId, tsdk::TrackStatus status)** tells that the track status updated.

    – *frameId* - id of the frame
    – *trackId* - id of the track
    – *status* - track new status

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

- **void trackReIdentificate(tsdk::FrameId frameId, tsdk::TrackId trackId, tsdk::TrackId reidTrackId)** tells that the track with id = `trackId` was matched to one of the old non-active tracks with id = `reidTrackId`. See section **ReIdentification** for details.

    – *frameId* - id of the frame
    – *trackId* - id of the track, that was matched to one of the old non-active tracks
    – *reidTrackId* - id of the non-active track, that successfully mathed to track with id = `trackId`

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

```
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,
    //! some unknown reason (shouldn't meet in normal workflow)
    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,
    //! finished by user (see `IStream::finishTracks` for details)
    USER,
    //! non-active track ends because of lifetime expired
```

```
        NONACTIVE_TIMEOUT,
        //! active track ends because of reidentification with old non-active
            track
        ACTIVE_REID,
        //! non-active track ends because of reidentification with older non-
            active track
        //  (that means, that current track couldn't been updated and was
            matched to old non-active at the same time)
        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;
    };
```

- **void trackStatusUpdate(const fsdk::Span &streamIDs, const fsdk::Span &data)** Batched
  version of the trackStatusUpdate callback.

    - *streamIDs* - array of streams id
    - *data* - array of callback data for each stream

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

        //! track id
        tsdk::TrackId trackId;

        //! new track status
        tsdk::TrackStatus status;
    };
```

- **void trackReIdentificate(const fsdk::Span &streamIDs, const fsdk::Span &data)** Batched

version of the `trackReIdentificate` callback. See section **ReIdentification** for details.

- *streamIDs* - array of streams id
- *data* - array of callback data for each stream

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

    //! id of track, that was matched to one of the old non-active tracks
    tsdk::TrackId trackId;

    //! id of the non-active track, that successfully mathed to track with
        id = 'trackId'
    // see human tracking algorithm section in docs for details
    tsdk::TrackId reidTrackId;

    //! similarity from matching of tracks descriptors
    float similarity;
};
```

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;
    };
```

- **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.

Tracks lifetime

All tracks live until they meet the specific conditions of tracking algorithm (e.g. out of frame bounds enough or `skip-frames` logic). Human tracking algorithm has its own rules for tracks lifetime (see section `Human tracking algorithm`). Users can finish tracks manually with IStream function `finishTracks`. `TrackEndReason` implies reason of track finishing.

Human tracking algorithm

Human tracking algorithm differs from the faces one. Tracker feature isn't used at all anymore, only detect/redetect are used. For matching tracks with new detections IOU metrics is used. 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 human tracking in that case (or under some other conditions) it moves to non-active group of tracks. `trackStatusUpdate` callback with `status = TrackStatus::NONACTIVE` is invoked to indicate about that. Tracks from that group are invisible for all observers and they don't participate in common tracking processing (detect/redetect).

Note, that the parameter `skip-frames` doesn't affect on human tracking algorithm. Human tracks are finished according to its own logic. There are some cases, when `trackEnd` is called for human track (see `TrackEndCallbackData` reasonfield):

1. non-active track is finished by timeout set by config parameter "human":"non-active-tracks-lifetime" (reason = `NONACTIVE_TIMEOUT`).

2. active track is finished because of `reIdentification` with another old track from the non-active group. Note, that the old track id becomes active again. First active track's id is just replaced with the older one and `trackStatusUpdate` is called with `status = TrackStatus::ACTIVE` for the old track id to indicate, that it's active again, `trackEnd` is called for the current active track id to indicate it doesn't exist anymore (reason = `ACTIVE_REID`). For this case config parameter "human":"reid-matching-detections-number" sets lifetime of the active track (in number of frames) needed for matching to the old non-active tracks.

3. active track is finished if it to be moved to the non-active group (e.g. detector/redetect fails), but it successfully matched (`reIdentification` called) to the old non-active one at the same time (reason = `NONACTIVE_REID`). Also in this case lifetime counter of the non-active track is reset.

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.

ReIdentification

`ReIdentification` is a feature, that improves tracking accuracy. `ReIdentification` is intended to solve problem, described in section `Human tracking algorithm`. It matches two tracks with different id-s and merges them into one track with id of the older one. `trackReIdentificate` callback signals about successfull matching and merging of the two tracks into one. The feature's behavior is regulated by config parameters "human":"reid-matching-threshold", "reid-matching-detections-number". Two tracks will be matched only if similarity between them higher then "reid-matching-threshold". If you don't want `ReIdentification` feature at all, then just set up this parameter value higher than 1.

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

Memory consumption

TrackEngine itself doesn't allocate much memory for internal calculations, but in `callback-mode = 1` it keeps frames/images in frames and callbacks queues and current tracks data. 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. The number 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.

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 section

- **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 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 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 human 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 tracking specific parameters section

- **human-landmarks-detection** - Flag to enable human 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" />
```

```xml
            <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"/>
            <param name="iou-connection-threshold" type="Value::Float1" x="0.5"
                />
            <param name="reid-matching-threshold" type="Value::Float1" x="0.85"
                />
            <param name="non-active-tracks-lifetime" type="Value::Int1" x="100"
                />
            <param name="reid-matching-detections-number" 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
#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

#define USE_GPU false
#define USE_BATCHED_OBSERVERS true // preferable way
#define USE_FACE_DETECTOR true
#define USE_BODY_DETECTOR false

#define USE_IMAGE_CACHE false // allocations optimization
#define IMAGE_CACHE_SIZE 40

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

```cpp
/**
* @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;
};

std::vector<ImageCache> streamCaches;

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 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 {
        if (USE_FACE_DETECTOR) {
            // track with id = 'trackId' finished
        }
        else if (USE_BODY_DETECTOR) {
            // track with id = 'trackId' moved to non-active tracks group or
                finished
            // we can't get actual reason from this callback (due to
                function signature compatibility with older versions)
            // users should use new batched observer api to get it (see '
                TrackEndCallbackData')
        }
    }

    void trackReIdentificate(tsdk::FrameId frameId, tsdk::TrackId trackId,
        tsdk::TrackId reidTrackId) override {
        // track with id = 'trackId' matched to one of the old non-active
            tracks with id = 'reidTrackId'
        // after this callback trackEnd will be called for track with id = '
            trackId' (like, reidTrackId replaces trackId),
        // track with id = 'reidTrackId' will be in non-active state again
            or active: that depends on whether track with id = 'trackId' was
            updated on the last frame ot not
    }

    void trackStatusUpdate(tsdk::FrameId frameId, tsdk::TrackId trackId,
        tsdk::TrackStatus status) override {
    }

    void visual(const tsdk::FrameId &frameId,
                const fsdk::Image &image,
                const tsdk::TrackInfo * trackInfo,
                const int nTrack,
```

```cpp
                   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 (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 {
    // the bigger the better (example of best shot logic)
    /*if (m_bestAreas.find(descr.trackId) == m_bestAreas.end())
        m_bestAreas[descr.trackId] = 0;

    if (descr.detection.rect.getArea() > m_bestAreas[descr.trackId]) {
        m_bestAreas[descr.trackId] = descr.detection.rect.getArea();
        return true;
    }*/
    return true;
}

bool needRGBImage(const tsdk::FrameId frameId, const tsdk::
```

```cpp
        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;

    // realization like per-stream observers (see `SuperObserver`)
    void bestShot(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk::
        Span<tsdk::BestShotCallbackData> &data) override {
    }

    void trackEnd(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk::
        Span<tsdk::TrackEndCallbackData> &data) override {
    }

    void trackStatusUpdate(const fsdk::Span<tsdk::StreamId> &streamIDs,
        const fsdk::Span<tsdk::TrackStatusUpdateCallbackData> &data) override
         {
    }

    void trackReIdentificate(const fsdk::Span<tsdk::StreamId> &streamIDs,
        const fsdk::Span<tsdk::TrackReIdentificateCallbackData> &data)
        override {
    }

    void visual(const fsdk::Span<tsdk::StreamId> &streamIDs, const fsdk::
        Span<tsdk::VisualCallbackData> &data) override {
    }
```

```cpp
        void debugForegroundSubtraction(const fsdk::Span<tsdk::StreamId> &
            streamIDs,
                                    const fsdk::Span<tsdk::
                                        DebugForegroundSubtractionCallbackData> &
                                        data) override {
        }

        void debugDetection(const fsdk::Span<tsdk::StreamId> &streamIDs,
                        const fsdk::Span<tsdk::DebugDetectionCallbackData> &data
                            ) override {
        }
};

int main(int argc, char** argv) {
    if (USE_FACE_DETECTOR && USE_BODY_DETECTOR) {
        std::cout << "Both face and body detectors are't supported yet" <<
            std::endl;
        exit(EXIT_FAILURE);
    }

    int keyboard;
    int streamCount = 1;
    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++) {
            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));
```

```cpp
        }
    } 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();


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


    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) {
        configTE->setValue("other", "tracker-type", "vlTracker");
    }
#ifdef WITH_GPU
    // WARN! gpu supports only 'vlTracker' or 'none' tracker
    if (USE_GPU) {
        configTE->setValue("other", "tracker-type", "vlTracker");
    }
#endif


    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;
```

```
        BatchedSuperObserver batchedSuperObserver;

        threads.reserve(streamCount);
        std::atomic<bool> stop{false};

        streamCaches.resize(streamCount, IMAGE_CACHE_SIZE);

        auto threadFunc = [&](int captureIndex){
            uint32_t index = 0;
            auto& capture = captures[captureIndex];
            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[captureIndex], 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);
                          }
                        }
```

```cpp
                    else
#endif
                    {
                         image = cvImageCPUWrapper.clone();
                    }
                    std::cout << "Image:" << image.getWidth() << "x" <<
                         image.getHeight() << " residence: " << static_cast<
                         int>(image.getMemoryResidence()) << std::endl;
                    streamsList[captureIndex]->pushFrameWaitFor(image, index
                         , nullptr, std::numeric_limits<uint32_t>::max());
                }

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

    if (USE_BATCHED_OBSERVERS) {
        // set batched callbacks
        trackEngine->setBatchBestShotObserver(&batchedSuperObserver);
        trackEngine->setBatchVisualObserver(&batchedSuperObserver);
        trackEngine->setBatchDebugObserver(&batchedSuperObserver);
    }

    int observerIndex = 0;
    for (int i = 0; i < streamCount; i++) {
        // create stream
```

```cpp
        fsdk::Ref<tsdk::IStream> stream = fsdk::acquire(trackEngine->
            createStream());
        observers[observerIndex].m_streamId = observerIndex;

        if (!USE_BATCHED_OBSERVERS) {
            // 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);

        threads.emplace_back(std::async(std::launch::async, threadFunc, i));
        std::cout << "stream " << i << " started" << std::endl;

        observerIndex++;
    }

    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;
    }
```

```cpp
    // it's recommended to join each stream manually
    for (auto &stream : streamsList) {
        stream->join();
    }

    // this internally calls join for all streams (that wasn't joined yet)
        and stops processing
    trackEngine->stop();

    const std::chrono::high_resolution_clock::time_point now = std::chrono::
        high_resolution_clock::now();
    const std::chrono::milliseconds duration =
            std::chrono::duration_cast<std::chrono::milliseconds>(now -
                start);
    std::cout << "TOTAL DURATION: " << duration.count() << std::endl;
}
```