VisionLabs
MACHINES CAN SEE

TrackEngine Handbook

VisionLabs B.V.

Keizersgracht 311, 1016 EE, Amsterdam, the Netherlands

+31 20 369 04 93

info@visionlabs.ai

www.visionlabs.ai

Contents

Introduction 3

Glossary 3

Working with TrackEngine 3

BestShotObserver 4

VisualObserver 5

DebugObserver 5

BestShotPredicate 6

Threading 6

Settings 7

Logging section 7

Other section 7

Config example 7

Example 8

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)**
  - *engine* - pointer to FaceEngine instance (should be already initialized)
  - *configPath* - path to TrackEngine config file
  - *return value* - pointer to ITrackEngine
- **ITrackEngine* tsdk::createTrackEngine(fsdk::IFaceEngine* engine, const fsdk::ISettingsProviderPtr& provider)**
  - *engine* - pointer to FaceEngine instance (should be already initialized)
  - *provider* - settings provider with TrackEngine configuration
  - *return value* - pointer to ITrackEngine

It is not recommended to create multiple TrackEngine instances in one application.

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()**
  - *return value* - pointer to IStream

You can create multiple streams at once if required (in cases when you would like 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).

The frames are submitted on a one by one basis and each frame has its own unique id.

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

You can set up an observer to receive and react to events.

Stream observer interfaces:

- BestShotObserver
- VisualObserver
- DebugObserver

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

BestShotPredicate 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.

BestShotObserver

- **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 {
      //! Source image
      fsdk::Image image;

      //! Face landmarks
      fsdk::Landmarks5 landmarks;

      //! Detection
      fsdk::Detection detection;

      //! Index of the frame
      tsdk::FrameId frameIndex;
```

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

- **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

VisualObserver

- **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;

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

    //! Id of track
    TrackId trackId;

    //! Score for last detection in track
    float lastDetectionScore;

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

- *nTrack* - number of tracks

DebugObserver

- **void debugDetection(const tsdk::DetectionDebugInfo& descr)** detector debug callback. Default implementation does nothing.

- *descr* - detection debugging description

```
struct TRACK_ENGINE_API 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
    //! value that checkBestShot returned
    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

Threading

TrackEngine is multi-threaded. The number of threads is configurable and depends on the currently bound `FaceEngine` settings.

TrackEngine calls Observer functions in separate threads. The `checkBestShot` function is called in the main frame processing thread. It is recommended to avoid expensive computations in `checkBestShot`.

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

- **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 there is no detection in estimated area, TrackEngine will wait this number of frames before considering the track lost and finishing it. `36` by default.
- **frg-subtractor** - Whether to enable foreground subtractor or not. Foreground subtractor reduces the detection area by cutting out static background parts. While improving performance, this may reduce face detection recall in some cases. `1` by default.
- **frame-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 consumption dramatically. It is used to balance between resource utilization and face detection recall. `10` 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. Otherwise, if the buffer becomes smaller, the behaviour becomes more like realtime appearance.

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="skip-frames" type="Value::Int1" x="36" />
        <param name="frg-subtractor" type="Value::Int1" x="1" />
        <param name="frame-buffer-size" type="Value:Int1" x="10" />
        <param name="callback-buffer-size" type="Value::Int1" x="300" />
    </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.

```
#include "tsdk/ITrackEngine.h"
#include <opencv/cv.hpp>
#include <iostream>
#include <map>

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

struct Observer :
        tsdk::IBestShotObserver,
        tsdk::IVisualObserver,
        tsdk::IBestShotPredicate {

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

    }

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

        void visual(const tsdk::FrameId &frameId,
                    const fsdk::Image &image,
                    const tsdk::TrackInfo * trackInfo,
                    const int nTrack,
                    const tsdk::AdditionalFrameData* data) override {
            // 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::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));
            }
        }

        void trackEnd(const tsdk::TrackId& trackId) override {
            // nothing to do here
        }

        bool checkBestShot(const tsdk::DetectionDescr& descr, const tsdk::
            AdditionalFrameData* data) override {
            // the bigger the better (example of best shot logic)
            const fsdk::Rect rect = descr.detection.getRect();
            if (rect.getArea() > m_bestAreas[descr.trackId]) {
                m_bestAreas[descr.trackId] = rect.getArea();
                return true;
            }
            return false;
        }
    };

    int main() {
        cv::Mat frame; //current frame
        cv::VideoCapture capture; //create the capture object
        int keyboard;
        capture.open(0);
        Observer visualObserver{0};

        const std::string fsdkDataPath = "path_to_faceEngine_data";
        const std::string licenseConfPath = fsdkDataPath + "/license.conf";
        // TrackEngine needs initialized faceEngine
        auto faceEngine = fsdk::createFaceEngine(fsdkDataPath.c_str()).getValue
            ();
        fsdk::ISettingsProviderPtr config = fsdk::createSettingsProvider((
            fsdkDataPath + "/faceengine.conf").c_str()).getValue();
        faceEngine->setSettingsProvider(config);

        // Make license activation
        // Required only for mobile platform for now.
        fsdk::ILicense* licensePtr = faceEngine->getLicense();
        if (!licensePtr) {
            std::cout << "Failed to get FaceEngine license." << std::endl;
            return -1;
        }

        if (!fsdk::activateLicense(licensePtr, licenseConfPath.c_str())) {
            std::cout << "Failed to activate FaceEngine license." << std::endl;
            return -1;
        }

        // get TrackEngine
        fsdk::Ref<tsdk::ITrackEngine> trackEngine =
                fsdk::acquire_as<tsdk::ITrackEngine>(tsdk::createTrackEngine(
                    faceEngine,
                     "path_to_trackEngine_data/trackengine.conf"));
        // create stream
        fsdk::Ref<tsdk::IStream> stream = fsdk::acquire(trackEngine->
            createStream());
        // set callbacks for stream
        stream->setVisualObserver(&visualObserver);
        stream->setBestShotPredicate(&visualObserver);
        stream->setBestShotObserver(&visualObserver);

        if (!capture.isOpened()) {
            //error in opening the video input
            std::cout << "video not opened"<< std::endl;
            exit(EXIT_FAILURE);
        }
        int counter = 0;

        while( (char)keyboard != 'q' && (char)keyboard != 27 ) {
            //read the current frame
            if(!capture.read(frame)) {
                std::cerr << "Unable to read next frame." << std::endl;
                exit(EXIT_FAILURE);
            }

            // convert opencv matrix to fsdk::Image
            fsdk::Image im = fsdk::Image(frame.cols, frame.rows, fsdk::Format::
               R8G8B8, frame.data);
            fsdk::Image newim = im.clone();

            // push frame to our stream
            if (!stream->pushFrame(newim, counter++, nullptr)) {
                std::cout << "pushFrame error " << std::endl;
                capture.release();
                return 0;
            }

            // draw windows for all streams
            for (auto& im : frameImages) {
                cv::imshow(cv::format("frame %d",im.first), im.second);
            }

            // draw windows for all best shots
            for (auto& im : bestShotImages) {
                cv::imshow(cv::format("bestShot %d",im.first), im.second);
            }

            keyboard = cv::waitKey( 30 );
        }
        //delete capture object
        capture.release();
    }
```