



VisionLabs BestShotMobile Handbook

Contents

Introduction	3
Glossary	4
1 BestShotMobile structure overview	5
2 BestShotMobile object	6
2.1 Configuration file	6
2.2 Liveness algorithm types	6
2.3 Pushing data	7
2.4 Receiving results	7
3 IBestShotMobileObserver interface	8
3.1 BestShot callback	8
3.2 Liveness callback	9
3.3 TrackEnd callback	10
3.4 Settings the callbacks	10
4 How to use	11
5 Configuration file description	15

Introduction

BestShotMobile is a wrapper library with extended functionality, which utilizes **FaceEngine** and **TrackEngine** building blocks to produce different solutions for the next tasks:

1. Make a bestshot from the video stream.
2. Make a liveness check for this bestshot.

This short guide describes the product core concepts, represents the main BestShotMobile features and suggests usage scenarios.

It is strongly recommended to familiarize with **FaceEngine Handbook** (**FaceEngine_Handbook.pdf** in the delivery package) before reading this guide since FaceEngine core concepts are widely used all over this handbook without additional explanation.

Glossary

Term	Definition
Liveness	Face features that allow distinguishing a living person from a photo/prearranged video.
Best shot	The best detection from the video stream or several photos for the recognition algorithms

1 BestShotMobile structure overview

BestShotMobile library contains several main parts:

- **The main BestShotMobile object.** This object should be created for library use with the required liveness algorithm type. All liveness algorithms will be discussed later.
- **The user-defined BestShotMobileObserver.** BestShotMobile library works based on the asynchronous interface with returning results through callbacks. To receive those callbacks the user should implement the IBestShotMobileObserver in the user application.

2 BestShotMobile object

The **BestShotMobile object** is a root object of the entire BestShotMobile library. All functionality is based on this object.

This object should be created before beginning of the video stream processing. Call **createBestShotMobile()** function to create a BestShotMobile instance.

As a wrapper library, the BestShotMobile library requires the FaceEngine instance and the TrackEngine instance. Both pointers are expected as parameters of the **createBestShotMobile()** function.

2.1 Configuration file

For proper functioning BestShotMobile library requires the configuration file. The file is called “bestshotmobile.conf” and stored in **dataPath** directory by default. See the “FaceEngine_Handbook” for explanation of the “data” directory and its meaning.

At runtime, the configuration file data is managed by a special object that implements **ISettingsProvider** interface. The provider is instantiated using **createSettingsProvider()** function that accepts configuration file location as a parameter or uses a default one if not explicitly defined. For detail of the **ISettingsProvider** usage see the “FaceEngine_Handbook”.

2.2 Liveness algorithm types

There are several supported liveness algorithm types:

- LivenessType::None - no any liveness checks. Use this liveness type if the liveness check is not required. As a result, BestShotMobile will prepare the best shot, but this best shot will not be checked for liveness.
- LivenessType::Offline - offline on device check. This algorithm does not require a network connection. All checks are processed on the device.

This type of liveness check is supported only in the Complete SDK version.

- LivenessType::Online - online liveness check on the backend server. It is the recommended to use liveness algorithm type. It is required to fill two fields in configuration file for correct algorithm work

```
<param name="URL" type="Value::String" text="" />
<param name="Luna-Account-Id" type="Value::String" text="" />
```

- LivenessType::Offline_OSL - offline on device check using mobile edition of LivenessOneShotRGBEstimator. This algorithm does not require a network connection. All checks are processed on the device.

This type of liveness check is supported only in the Complete SDK version.

2.3 Pushing data

To catch the best shot from the video stream user should split the video stream to the frames and then push it one by one to the BestShotMobile object through the ***IBestShotMobile::pushFrame*** method.

Frames are handled in the separated thread by asynchronous scenario in the BestShotMobile library.

2.4 Receiving results

User should use callbacks to receive results from the BestShotMobile library . See the “[IBestShotMobileObserver interface](#)” chapter for details.

3 IBestShotMobileObserver interface

As was explained in the “BestShotMobile object” chapter all frames are handled into the BestShotMobile library with an asynchronous scheme.

To receive results of the processing use should define the callbacks by implementing the IBestShotMobileObserver interface. This interface has several virtual methods that should be overridden.

3.1 BestShot callback

All frames from the **pushFrame** method are handled with the next steps:

1. Face detection to find any face on the frame
2. Checking quality of the frame and head pose with BestShotQualityEstimator. See “FaceEngine_Handbook” for details.

User code will receive the **bestShot** callback with a **BestShotInfo** structure for every frame with detected face. The **BestShotInfo** structure contains next fields:

```
struct BestShotInfo {
    //!< State of this frame
    BestShotState state;

    //!< Source image
    fsdk::Image image;
    //!< Detection with face
    fsdk::Detection detection;
    //!< Face landmarks
    fsdk::Landmarks5 landmarks;

    //!< Estimation of the head position.
    //!< This parameter could help to show notification to the user in
    case of bad angles.
    fsdk::HeadPoseEstimation headPoseEstimation;
    //!< AGS estimation result.
    //!< This parameter could help to show notification to the user in
    case of bad quality.
    float agsEstimation;

    //!< Index of the frame
    tsdk::FrameId frameIndex;
    //!< Index of the track
    tsdk::TrackId trackId;
};
```


If all checks are successfully processed and the results are good enough (head angles are less than thresholds, quality of the frame is good) the current frame is estimated as the best shot. The **BestShotInfo::state** field will be set to **BestShotState::Ok** in this case.

If some parameters of the frame are not good or some error happened during the processing the **BestShotInfo::state** will be set to one of the next values: **BestShotState::BadQuality** - in case of bad frame quality, **BestShotState::BadHeadPose** - in case of bad head angles, **BestShotState::Error** - in case of some error during processing.

It can be a good solution to show some notification to the user about **BestShotState::BadQuality** and **BestShotState::BadHeadPose** statuses of the bestshot.

If the best shot without liveness check is not required for any business logic and notification to the user is not required this user-defined callback could be just empty. In case of some additional logic (for example, some user interaction) this callback could be useful.

3.2 Liveness callback

All the frames that were estimated as the best shot will be checked for the liveness state at the next step. The result of this check will be sent through the separated **liveness** callback with the **BestShotInfo** structure and the **LivenessState** enum.

The **BestShotInfo** structure was described in the previous section.

The **LivenessState** enum can contain the next states:

```
enum class LivenessState {
    Alive,                //!< Face was estimated as alive.
    Fake,                 //!< Face was estimated as not alive.
    None,                 //!< No liveness check at all.
    NotReady,             //!< Liveness check not ready for now.
        Need more frames to handle.
    BadHeadPose,          //!< Head angles bigger than thresholds.
    BadQuality,           //!< Image quality is less than threshold.
    FaceNotFound,         //!< Face was not found.
    FaceTooClose,         //!< Face is too close to the camera.
    FaceCloseToBorder,    //!< Face is too close to the frame border
    .
    FaceTooSmall,         //!< Face is too small.
    TooManyFaces,         //!< There is more than one face in the
        frame.
    Timeout,              //!< Liveness timeout.
    CriticalError          //!< Critical during liveness processing.
};
```

Note It can be a good solution to show some notification to the user about **LivenessState::BadQuality**, **LivenessState::BadHeadPose**, **LivenessState::FaceTooClose**, **LivenessState::FaceTooSmall** and **LivenessState::TooManyFaces** statuses of the liveness.

3.3 TrackEnd callback

The BestShotMobile library has a face tracking functionality. When the current face track has ended no more best shots should be expected from it. To handle this situation user-defined **trackEnd** callback could be used.

If such situation should not be handled by the business logic this callback could be just empty.

3.4 Settings the callbacks

To use all callbacks user should implement the **IBestShotMobileObserver** interface. Then the user should create this implementation class and pass the pointer to this class to the **setBestShotMobileObserver** method.

4 How to use

Follow the next steps to use the BestShotMobile library :

1. Implement the **IBestShotMobileObserver** interface. All virtual methods should be implemented.
2. Create FaceEngine and TrackEngine instances. See the “FaceEngine_Handbook” and the “TrackEngine_Handbook” for details.
3. Create BestShotMobile instance.
4. Create an instance of the callbacks implementation class.
5. Pass a pointer to the callbacks implementation class to the **setBestShotMobileObserver** method of the **BestShotMobile** instance.
6. Push all needed frames to the **pushFrame** method of the **BestShotMobile** instance.
7. Receive and handle all needed callbacks.
8. Wait until the end of processing by calling the blocking method **join** of the **BestShotMobile** instance.

See the next code example:

```
#include <bsmobile/IBestShotMobile.h>

#include <iostream>

struct BestShotMobileObserver : mobile::IBestShotMobileObserver {

    void bestShot(const mobile::BestShotInfo& bestShotInfo) override {
        std::cout << "This is just an example of the bestShot callback
            implementation!" << std::endl;
    }

    void liveness(const mobile::LivenessState livenessSate,
        const mobile::BestShotInfo& bestShotInfo) override {
        std::cout << "This is just an example of the liveness callback
            implementation!" << std::endl;
    }

    void trackEnd(const tsdk::TrackId& trackId) override {
        std::cout << "This is just an example of the trackEnd callback
            implementation!" << std::endl;
    }
};
```

```

fsdk::Image takeNextFrame() {
    // some implementation of the frame capturing here
    return fsdk::Image{};
}

int main() {
    // Create the FaceEngine instance based on the configuration file
    // (./data/faceengine.conf by default)
    auto resFaceEngine = fsdk::createFaceEngineMobile("./data");
    if (!resFaceEngine) {
        std::cout << "Failed to create FaceEngineMobile instance! " <<
            resFaceEngine.what() << std::endl;
        return -1;
    }
    fsdk::Ref<fsdk::FaceEngineType> faceEngine = resFaceEngine.getValue();

    // Take license object
    fsdk::ILicense* license = faceEngine->getLicense();
    if (!license) return -1;
    // Make activation with license configuration file (./data/license.conf
    // by default)
    auto resActivate = fsdk::activateLicense(license, "./data/license.conf")
        ;
    if (!resActivate) {
        std::cout << "Failed to activate license! " << resActivate.what() <<
            std::endl;
        return -1;
    }

    // Create the TrackEngine settings instance and read parameters
    // from the config (./data/trackengine.conf by default)
    auto resTrackEngineSettings = fsdk::createSettingsProvider("./data/
        trackengine.conf");
    if (!resTrackEngineSettings) {
        std::cout << "Failed to parse trackengine settings! " <<
            resTrackEngineSettings.what() << std::endl;
        return -1;
    }
    fsdk::Ref<fsdk::ISettingsProvider> trackEngineSettings =
        resTrackEngineSettings.getValue();

    // Create the TrackEngine instance.
    auto resTrackEngine = tsdk::createTrackEngine(
        faceEngine,
        trackEngineSettings

```

```

    );
    if (!resTrackEngine) {
        std::cout << "Failed to build TrackEngine instance! " <<
            resTrackEngine.what() << std::endl;
        return -1;
    }
    fsdk::Ref<tsdk::ITrackEngine> trackEngine = resTrackEngine.getValue();

    // Create the BestShotMobile settings instance and read parameters
    // from the config (./data/bestshotmobile.conf by default)
    auto resBestShotMobileSettings = fsdk::createSettingsProvider("./data/
        bestshotmobile.conf");
    if (!resBestShotMobileSettings) {
        std::cout << "Failed to parse bestshotmobile settings! " <<
            resBestShotMobileSettings.what() << std::endl;
        return -1;
    }

    fsdk::Ref<fsdk::ISettingsProvider> bestShotMobileSettings =
        resBestShotMobileSettings.getValue();
    // Check the liveness type. Online in this example.
    bestShotMobileSettings->setValue(
        "BestShotMobile::Settings",
        "LivenessType",
        1
    );
    // Online liveness use a backend server.
    // So, need to set the URL to the Liveness API
    bestShotMobileSettings->setValue(
        "LivenessOnline::Settings",
        "URL",
        "http://example.com:12345/5/liveness"
    );
    // And need to set the Account-Id
    bestShotMobileSettings->setValue(
        "LivenessOnline::Settings",
        "Luna-Account-Id",
        "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
    );
    // Create the BestShotMobile instance
    std::unique_ptr<mobile::IBestShotMobile> bsmobile {
        mobile::createBestShotMobile(
            faceEngine,
            trackEngine,
            bestShotMobileSettings

```

```

    )
};
if (!bsmobile) {
    std::cout << "Failed to create BestShotMobile!" << std::endl;
    return -1;
}

// Create the user defined BestShotMobileObserver class instance.
BestShotMobileObserver bestShotMobileObserver;
// Set this instance as an observer into the BestShotMobile instance.
bsmobile->setBestShotMobileObserver(&bestShotMobileObserver);

// Just for example
size_t maxFramesCount = 100;
for (size_t frameIndex = 0; frameIndex < maxFramesCount; /*empty*/) {
    fsdk::Image nextFrame = takeNextFrame();

    // If the internal queue of the BestShotMobile has a free place, the
    // frame
    // will be taken into the processing and the ``pushFrame`` will
    // return true.
    if (bsmobile->pushFrame(nextFrame, frameIndex))
        ++frameIndex;
}

// Wait all processing finish.
bsmobile->join();

return 0;
}

```

5 Configuration file description

The configuration file has a module-based structure. Each liveness algorithm has its own parameters.

There is also a common section with the BestShotMobile parameters.

Table 2: BestShotMobile parameters

Parameter	Description	Default value
LivenessType	Default liveness algorithm.	0

|| Possible values:

0 - LivenessNone

1 - LivenessOnline

2 - LivenessOffline

--	--	--	--	--

| Type: Value::Int1 ||

Table 3: LivenessNone parameters

Parameter	Description	Default value
AGSThreshold	Threshold for the quality (AGS) check. Type: Value::Float1	0.2
HeadPoseThreshold	Thresholds for the head pose check. Pitch, yaw, roll angles. Type: Value::Float3	x="20.0"y="20.0"z="30.0"

Table 4: LivenessOnline parameters

Parameter	Description	Default value
AGSThreshold	Threshold for the quality (AGS) check. Type: Value::Float1	0.2

Parameter	Description	Default value
HeadPoseThreshold	Thresholds for the head pose check. Pitch, yaw, roll angles. Type: Value::Float3	x="20.0"y="20.0"z="30.0"
MinFaceSize	Minimum face size to check. Type: Value: Int1	220
MinFrameSize	Minimum frame size to check. Type: Value: Int1	480
MinBorderPaddings	Minimum distance between face rect and image borders. Type: Value: Int1	25
URL	Backend API URL. Type: Value::String	"" (empty)
Luna-Account-Id	Account ID to work with backend API. Type: Value::String	"" (empty)

Table 5: LivenessOffline parameters

Parameter	Description	Default value
AGSThreshold	Threshold for the quality (AGS) check. Type: Value::Float1	0.2
HeadPoseThreshold	Thresholds for the head pose check. Pitch, yaw, roll angles. Type: Value::Float3	x="20.0"y="20.0"z="30.0"
MinCheckCount	Minimum best shots count to check. Type: Value::Int	3

Table 6: LivenessOfflineOSL parameters

Parameter	Description	Default value
AGSThreshold	Threshold for the quality (AGS) check. Type: Value::Float1	0.2
HeadPoseThreshold	Thresholds for the head pose check. Pitch, yaw, roll angles. Type: Value::Float3	x="25.0"y="25.0"z="35.0"
MinFaceSize	Minimum face size to check. Type: Value: Int1	200

Parameter	Description	Default value
MinBorderPaddings	Minimum distance between face rect and image borders. Type: Value: Int1	10