CvTracker C++ library

v9.0.0

Table of contents

- Overview
- Versions
- Library files
- Key features and capabilities
- Operating principles
  - How the tracking algorithm works
  - Stop-frame function
  - Communication channel time delay compensation
- Supported pixel formats
- CvTracker class description
  - CvTracker class declaration
  - How to use the library
  - getVersion method
  - initVTracker method
  - setParam method
  - getParam method
  - getParams method
  - executeCommand method
  - processFrame method
  - getImage method
  - decodeAndExecuteCommand method
  - encodeSetParamCommand method of VTracker class
  - encodeCommand method of VTracker class
- Data structures
  - VTrackerCommand enum
  - VTrackerParam enum
- VTrackerParams class description
  - VTrackerParams class declaration

    - [Serialize video tracker params]
    - [Deserialize video tracker params]
    - [Read params from JSON file and write to JSON file]
- [Build and connect to your project]
- [Simple example]

Overview

C++ library **CvTracker** version **9.0.0** is intended for automatic object video tracking. The library is written in C++ (C++17 standard) and uses OpenCV (version >= 4.5.0) library to perform forward and backward Fourier transforms. The library is compatible **with any processors and operating systems** supporting C++ compiler (C++17 standard) and OpenCV library (version >= 4.5.0). The library provides fast calculation, compatibility with low-power processors, high accuracy and contains a lot of additional functions and modes, which allow using it in camera systems of any configuration. The library contains an advanced tracking algorithm **CSRM** developed by ConstantRobotics Ltd. The library provides tracking of low-contrast and small-sized objects against a complex background. The library contains a description of the C++ class **CvTracker**. A single instance of the class provides tracking of a single object on video. To track several objects simultaneously, several instances of the **CvTracker** class must be created. CvTracker library depends on third-party libraries: **VTracker** interface library (source code included, defines programming interface. Apache 2.0 license), **OpenCV** opensource library (linked, version >= 4.5.0, Apache 2.0 license). Additionally CvTracker demo application depends on third-party libraries: **FormatConverterOpenCv** pixel format converter library (source code included, the same license as CvTracker library), **SimpleFileDialog** file dialog library (source code included, Apache 2.0 license), **VSourceOpenCv** video capture library (source code included, the same license as CvTracker library).

Versions

Table 1 - Library versions.

Version	Release date	What's new
7.2.0	12.12.2023	Version 7.2.0.
8.0.0	25.12.2022	- The calculation speed has been increased. - Tracking stability has been improved. - Added support for YUV, YUYV, UYVY, NV12, RGB, BGR video formats for better tracking stability. - The number of configurable parameters has been reduced. - New control commands added.
8.1.0	10.06.2023	- Added new demo application for Windows and Linux. - Code optimized.

Version	Release date	What's new
9.0.0	19.12.2023	- Classes names changed. - Added new pixel format support. - Code optimized. - Tracking stability improved. - Programming interface changed. - Set/get params methods thread-safe.

Library files

The **CvTracker** library is a CMake project. Library source code files:

```
CMakeLists.txt ----------------- Main CMake file of the library.
3rdparty ----------------------- Folder with third-party libraries.
    CMakeLists.txt ------------- CMake file which includes third-party libraries.
    VTracker ------------------- Source code of the VTracker interface library.
benchmark ---------------------- Folder with CvTracker benchmark application.
    CMakeLists.txt ------------- CMake file for the benchmark application.
    main.cpp ------------------- Source code file of the benchmark application.
demo --------------------------- Folder with CvTracker demo application.
    3rdparty ------------------- Folder with third-party libraries.
        CMakeLists.txt --------- CMake file which includes third-party libraries.
        FormatConverterOpenCv -- Folder with pixel format converter library.
        SimpleFileDialog ------- Folder with file dialog library.
        VSourceOpenCv ---------- Folder with video capture library.
    CMakeLists.txt ------------- CMake file for the demo application.
    main.cpp ------------------- Source code file of the demo application.
example ------------------------ Folder with CvTracker example application.
    CMakeLists.txt ------------- CMake file for the example application.
    main.cpp ------------------- Source code file of the example application.
src ---------------------------- Folder with source code of the library.
    CMakeLists.txt ------------- CMake file of the library.
    CvTracker.cpp -------------- Source code file of the library.
    CvTracker.h ---------------- Header file which includes CvTracker class declaration.
    CvTrackerVersion.h --------- Header file which includes version of the library.
    CvTrackerVersion.h.in ------ CMake service file to generate version file.
    CsrmTracker.h -------------- Header file with CSRM video tracker implementation.
    CsrmTracker.cpp ------------ Source code file with CSRM video tracker implementation.
```

When the library is delivered in compiled form, the user gets the following files:

```
CMakeLists.txt ----------------- Main CMake file of the library.
3rdparty ----------------------- Folder with third-party libraries.
    CMakeLists.txt ------------- CMake file which includes third-party libraries.
    VTracker ------------------- Source code of the VTracker interface library.
benchmark ---------------------- Folder with CvTracker benchmark application.
    CMakeLists.txt ------------- CMake file for the benchmark application.
    main.cpp ------------------- Source code file of the benchmark application.
demo --------------------------- Folder with CvTracker demo application.
```

```
    3rdparty ------------------- Folder with third-party libraries.
        CMakeLists.txt --------- CMake file which includes third-party libraries.
        FormatConverterOpenCv -- Folder with pixel format converter library.
        SimpleFileDialog ------- Folder with file dialog library.
        VSourceOpenCv ---------- Folder with video capture library.
    CMakeLists.txt ------------- CMake file for the demo application.
    main.cpp ------------------- Source code file of the demo application.
example ------------------------ Folder with CvTracker example application.
    CMakeLists.txt ------------- CMake file for the example application.
    main.cpp ------------------- Source code file of the example application.
compiled ----------------------- Folder with CvTracker files.
    libCvTracker.a ------------- Static library file for Linux OS.
    or libCvTracker.lib -------- Static library file for Windows OS.
    CvTracker.h ---------------- Header file which includes VTracker class declaration.
    CvTrackerVersion.h --------- Header file which includes version of the library.
```

Key features and capabilities

Table 2 - Key features and capabilities.

Parameter	Value and description
Programming language	C++ (standard C++17) using the OpenCV library (version >= 4.5.0) to perform Fast Fourier transforms.
Compatibility with different operating systems	Compatible with any operating system that supports the C++ compiler (standard C++17) and the OpenCV library (version >= 4.5.0).
Maximum size of the tracking rectangle	**128x128** pixels. It is possible to track part of an object if it does not fit into the tracking rectangle. The shape of the tracking rectangle can be any within the minimum and maximum allowable limits.
Minimum size of the tracking rectangle	**16x16** pixels. The object can be **2x2** pixels size for normal tracking.
Minimum object size	**2x2** pixels.
The minimum object contrast	**5%.** Contrast refers to the ratio of the difference between the average brightness of pixels belonging to the object and the average brightness of pixels belonging to the background. The demo application is designed to evaluate this parameter.
Maximum object offset per one frame	The library provides tracking of objects as they change their position (change the position of the object center) per one video frame of up to **110 pixels** in any direction. The maximum allowable object displacement per video frame is determined by the search window size, which is set by the user.
Discreteness of object coordinates	**1 pixel** when estimating the center of an object (center of a tracking rectangle).

Parameter	Value and description
Object size estimation	The library estimates the size and position of an object within the tracking rectangle to enable automatic adjustment of its position and size at the operator's command.
Auto adjustment of the tracking rectangle position	The library can automatically adjust the position of the tracking rectangle while tracking an object. This allows to reduce the probability of tracking failure in the case of tracking dynamic maneuvering objects. User can enable/disable this function.
Auto adjustment of the tracking rectangle size	The library can automatically adjust the size of the tracking rectangle while tracking an object. This allows to reduce the probability of tracking failure in the case of tracking dynamic maneuvering objects. User can enable/disable this function.
Object speed estimation	The library calculates the horizontal and vertical components of object speed in video frames (pixels per frame).
Changing the parameters	The library allows user to change parameters of the tracking algorithm even while tracking. Excepts input pixel format and number of color channels for processing.
Supported pixel formats	Supported pixel formats of input video frames: **GRAY**, **BGR**, **RGB**, **YUV**, **YUYV**, **UYVY**, **NV12**, **NV21**, **YU12** and **YV12**. For each pixel format the algorithm can use one or more color channels for processing. See section “Supported pixel formats”.
Maximum and minimum video frame sizes to be processed	The maximum size of the video frames for processing is **8192x8192 pixels**, the minimum is **240x240 pixels**.
Calculation time	The library performs calculations for each video frame. Calculation speed does not depend on video frame sizes, but depends on library parameters. The main parameters which determine calculation speed are: **1.** search windows size, **2.** input pixel format, **3.** number of color channels for processing. The library does not perform any background tasks. **The library uses only one physical or logical processor core to perform calculations or number of cores equal to number of color channel if this option set by user.**
Object loss detection	The library automatically detects when an object is lost and switches the algorithm into LOST mode - trajectory prediction mode. When the object detection criteria are met, the library automatically recaptures the object (TRACKING mode). There are some limitations. It is recommended to test this function with demo application.
Adaptation to object shape and size changes	While tracking an object, the library adapts to object shape, size and brightness changes.

Parameter	Value and description
Obstacles processing	If an object is partially (**up to 50%**) blocked by a barrier, there is no tracking loss. The performance of the library in specific situations can be evaluated with the demo application.
Object search window	The size of the object's search area is set by the user in the library's parameters. The library's tracking algorithm searches object in a search area whose center coincides with the center of the tracking rectangle in the previous video frame. The library allows you to set only the following search area widths: **128** or **256** pixels, and possible search area heights of **128** or **256** pixels in any combination. It is recommended that the search area width and height be set to the same value.
Type of tracking algorithm	The calculations are performed using a modified correlation tracking algorithm **CSRM** developed by Constant Robotics Ltd.
STOP-FRAME function and compensation for communication delays	The library allows user to compensate time delays that occur in communication channels when transmitting object capture commands. The library also allows you to implement a STOP-FRAME mode to assist the operator in capturing dynamic objects. The duration of the STOP-FRAME mode can be set in the library parameters.

Note: The values in the table are applied to the concept of video frame(s) and the concept of pixel.

Operating principles

How the tracking algorithm works

The video tracker provides the following principle of operation: each video frame without dropping must be send to the tracker for processing regardless of the current tracker operation mode. If the tracker is not in tracking mode, the tracker does not perform frame processing, but the processing function must be called by user. Figure 1 shows basic principles of object search on video frames.

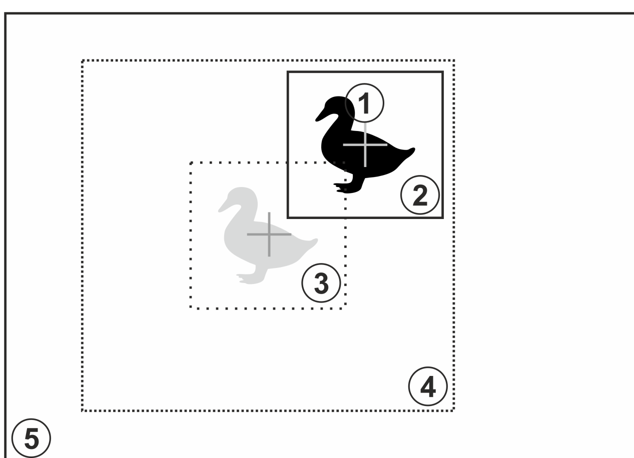

Figure 1 - Basic principles of object search. (**1** - object image on the current frame, **2** - tracking rectangle calculated after processing of the current frame, **3** - position of the tracking rectangle on the previous frame, **4** - object search window on the current frame relative to the position of the tracking rectangle on the previous frame, **5** - current video frame)

At the moment of object capturing, the rectangular area of the video frame (capture rectangle) specified in the capture parameters (position and size) is taken as the object reference image, on the basis of which the pattern is formed. The algorithm then searches an object in each frame of the video in particular search window. Search window is area bounded by the algorithm's parameters with the center coinciding with the calculated center of the tracking rectangle on the previous video frame (or the center of the capture rectangle if the first frame after capture is being processed, position of search window can be change by user for any video frame). The algorithm generates a surface of the spatial distribution of the probability (correlation surface) of the object presence in the search window. Once the surface is formed, it is analyzed to determine the most probable position of the object on the processed video frame (position of maximum value of the correlation surface). The calculated most probable position of the tracking object (with highest value of correlation function) in the current video frame (calculated center of the tracking rectangle) is taken as the coordinates of the object. Calculation of object movement (horizontal and vertical velocity components) is performed for each processed video. For each processed video frame, the algorithm calculates the position of the center of the tracking rectangle, the position and size of the object rectangle (the rectangle describes the size of the object image) in the tracking rectangle, and the speed components of the tracking object on the video frames (pixels per frame). Figure 1 shows a schematic representation of a video frame **(5)** that contains an image of a object **(1)**. Assume that on the previous video frame the object was in the area corresponding to area **(3)**, which is the area of the tracking rectangle (the most probable position of the object) in the previous video frame. The library performs object search in the area **(4)** whose center coincides with the position of the center of the tracking rectangle **(3)** in the previous video frame.

The tracking algorithm does not distinguish between pixels belonging to an object or background within the tracking rectangle immediately after capturing an object. Over time (as several frames are processed), the algorithm estimates whether a pixel within the tracking rectangle belongs to an object or to the background. Based on this information, the algorithm improves the quality of further tracking and estimates the size and position of the object (object image) within the tracking rectangle to enable subsequent automatic parameter adjustments at the operator's command or fully automatic. Tracker support follow modes:

Table 3 - Tracking algorithm operating modes.

Mode	**Description**
FREE - free mode.	In this mode, video tracker does not perform any calculations. Video tracker only adds video frames to the frame buffer. Conditions for entering FREE mode: **1.** Once the video tracker has been initialized. This mode is the default mode. **2.** Automatically when the automatic tracking reset criteria are met. **3.** After command RESET.
TRACKING - tracking mode.	In this mode the video tracker calculates the automatic tracking and updates all calculated (estimated) object parameters. Criteria for entering TRACKING mode: **1.** After the CAPTURE command. **2.** Automatically from LOST mode when object detection criteria are met.

Mode	Description
LOST - object loss mode.	In this mode, the video tracker searches object for automatic recapturing (switching to TRACKING mode) and updates it's coordinates in one of the ways specified in the parameters. LOST mode contains the following additional modes: **0.** Tracking rectangle coordinates are not updated (remain the same as before entering LOST mode). **1.** The tracking rectangle coordinates are updated based on the components of the object's speed calculated before going into LOST mode. When the tracking rectangle reaches any edge of the frame, the coordinate update in the corresponding direction stops. **2.** The tracking rectangle coordinates are updated based on the components of the speed of objects in the video frames calculated before going into LOST mode. When the tracking reset criteria is met, the device switches to FREE mode. Criteria for entering LOST mode: **1.** Automatically when object loss is detected. **2.** On command from TRACKING mode. 3. On command from INERTIAL mode. 4. On command from STATIC mode.
INERTIAL - inertial tracking mode.	In this mode the video tracker does not search for an object to recapture automatically, but only updates the coordinates of the tracking rectangle based on the previously calculated velocity components of the objects. Criteria for entering INERTIAL mode: **1.** On command from TRACKING mode. **2.** On command from LOST mode. **3.** On command from STATIC mode.
STATIC - static mode.	This mode does not perform any calculations and the tracking rectangle coordinates remain the same as before going into this mode. This mode is necessary to “freeze” the tracking algorithm for a certain number of frames. For example, if the tracking system is exposed to strong vibrations, it is possible to “freeze” the tracking algorithm until the vibration ends.

Figure 2 shows the operating mode graph and the possible transitions between them. The words "Auto" in figure 2 indicate the ability to change the mode automatically if the relevant criteria are met. The words "Command" indicated the ability to change mode by user's command.

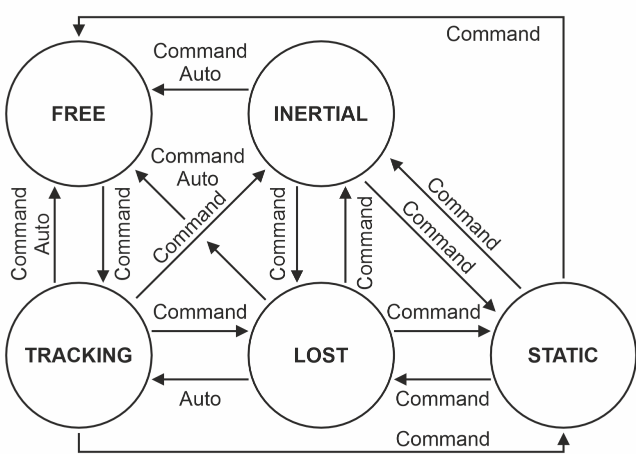

Figure 2 - Operation modes of the tracking algorithm. (Auto – automatic mode change capability)

Figure 2 shows the graph of operation modes. There are the following conditions for automatic mode changes (word “Auto” in figure 2): **1.** Automatic switching from TRACKING to FREE mode is possible only if the tracking rectangle center has touched (coincided in coordinates) any of the video frame edges. **2.** The automatic switching from TRACKING to LOST mode is possible when an object loss is detected – when the calculated object detection probability falls below the threshold. **3.** Automatic switching from LOST to

TRACKING mode is possible when an object is detected again after a loss - when the calculated object detection probability exceeds the threshold. **4.** Automatic reset of tracking in the LOST mode (switch to FREE mode) is possible when the center of the tracking rectangle touches the edge of the video frame (if the LOST mode option set to 2), as well as when the number of frames specified in the parameters has expired, at which the algorithm is continuously in LOST mode. **5.** Automatic reset of tracking in INERTIAL mode (switch to FREE mode) is possible when the center of the tracking rectangle reaches the edge of the frame.

Stop-frame function

If a moving object needs to be captured, this can often be difficult for the user because they need to align the tracking rectangle with an object that keeps changing its position on the video frames. In addition, it is difficult to capture a stationary object in case of camera vibration. To help the user capture an object in challenging dynamic environments, the library has a STOP-FRAME function. This function allows the user to stop the video playback and accurately capture an object on a stopped video frame.

The function works as follows: video frames is put to the library for processing frame-by-frame. The library places frames in a ring buffer of the size specified by user in library parameters. The tracking data contains an index corresponding to the position of the frame added to the ring buffer and is transmitted to the control system via communication channel. The user of the control system sees the video from the cameras on the monitor. Each video frame has its own identifier assigned by the tracking algorithm. The user can stop video playback, move the capture rectangle to an object on a stopped video frame and capture. When a capture command is formed, it includes a video frame identifier corresponding to the displayed video frame. When a capture command is received by the library, the object is captured on the frame in the frame buffer according to the identifier specified in the capture command. The video frame on which the object is captured will be some time behind the current video frame from the camera (the number of frames corresponding to the time the control system operator “stops” the video, with the addition of the delay time in the communication channels). After a capture, the algorithm switches to tracking mode. When processing subsequent frames, the library sequentially processes the frames in the frame buffer, starting with the frame where the object was captured. When a processing method of the library method is called, multiple frames are processed to “catch up” the current video frame. In this way the library “catches up” the current video frame from the camera in a short time and enters normal tracking mode. This function significantly reduces the skills requirements for users of tracking systems. **WARNING:** until the library has “caught up” the current video frame, it is not recommended to turn the pan-tilt systems. This may result in incorrect operation of the tracking systems.

Communication channel time delay compensation

When controlling the tracking system remotely (via communication channels), communication delays negatively affect the quality of the object being captured by the user. Video captured by the tracker device is compressed and transmitted to the control system with some delay. When the operator captures object the generated capture command also arrives at the tracker device with some delay. The capture command contains the coordinates of the capture rectangle center. When capturing a dynamic object, due to time delays in the communication channels, the captured area will not match the object. Figure 2 shows the displacement between capture rectangle and real object position.

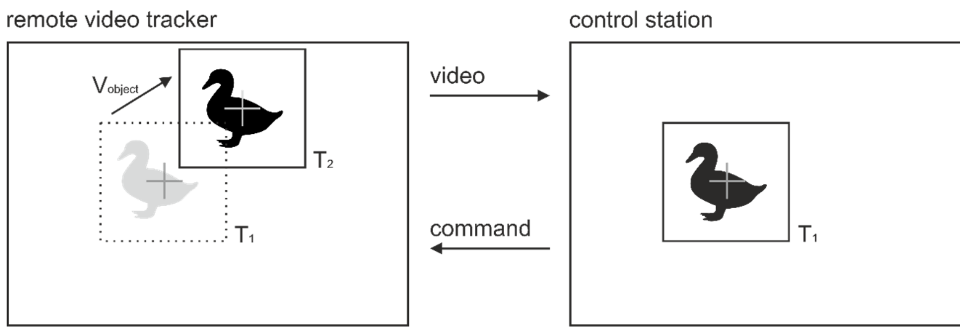

Figure 3 - Position error of the capture rectangle.

Figure 3 shows a video frame coming into the tracker device from the camera (left) and a video frame displayed to a user of the control system (right) at the same point of time. Assume that the video frame captured from the camera is simultaneously put to the tracking algorithm and sent to the control system. At time **T1**, the object was at one frame position. This frame after some time (encoding time + video frame transmission time + decoding time + display time) is displayed to the operator (user), who performs object capture. For operator the object position is corresponding to the time **T1**. Once an object has been captured, the capture command is sent to the tracking algorithm with some delay (command generation time + command transmission time + command decoding time) (time moment **T2**). The algorithm captures an object in the current video frame. If the object is moving, the error (horizontally and vertically) in the position of the capture rectangle will be as follows:

$$\Delta X = (T_2 - T_1) * V_x$$

$$\Delta Y = (T_2 - T_1) * V_y$$

where: ΔX – horizontal position error in pixels; ΔY – vertical position error in pixels; T1 – the point in time corresponding to the frame displayed to the user; T2 – the point in time corresponding to the current video frame; Vx – the horizontal component of the speed of an object in video frames (pixels per frame); Vy – the vertical component of the speed of the object in the video frames (pixels per frame).

To compensate for errors that occur, a video frame identifier must be included in the capture command. The principle for compensating of time delays in communication channels is identical to that of the STOP-FRAME function.

Supported pixel formats

The library uses input video frames in form of class **Frame** which declared in **Frame.h** (included in VTracker interface class). The library supports only **8 bit color depth** RAW pixel formats: **GRAY**, **BGR**, **RGB**, **YUV**, **YUYV**, **UYVY**, **NV12**, **NV21**, **YU12** and **YV12**. Table 4 shows bytes layout for different pixels formats.

Table 4 - Bytes layout of supported pixel formats. Example of 4x4 pixels image.

pixel

R00	G00	B00	R01	G01	B01	R02	G02	B02	R03	G03	B03
R10	G10	B10	R11	G11	B11	R12	G12	B12	R13	G13	B13
R20	G20	B20	R21	G21	B21	R22	G22	B22	R23	G23	B23
R30	G30	B30	R31	G31	B31	R32	G32	B32	R33	G33	B33

RGB24

pixel

B00	G00	R00	B01	G01	R01	B02	G02	R02	B03	G03	R03
B10	G10	R10	B11	G11	R11	B12	G12	R12	B13	G13	R13
B20	G20	R20	B21	G21	R21	B22	G22	R22	B23	G23	R23
B30	G30	R30	B31	G31	R31	B32	G32	R32	B33	G33	R33

BGR24

pixel

Y00	U00	V00	Y01	U01	V01	Y02	U02	V02	Y03	U03	V03
Y10	U10	V10	Y11	U11	V11	Y12	U12	V12	Y13	U13	V13
Y20	U20	V20	Y21	U21	V21	Y22	U22	V22	Y23	U23	V23
Y30	U30	V30	Y31	U31	V31	Y32	U32	V32	Y33	U33	V33

YUV24

pixel

Y00	Y01	Y02	Y03
Y10	Y11	Y12	Y13
Y20	Y21	Y22	Y23
Y30	Y31	Y32	Y33

GRAY

macro pixel

Y00	U00	Y01	V00	Y02	U02	Y03	V02
Y10	U10	Y11	V10	Y12	U12	Y13	V12
Y20	U20	Y21	V20	Y22	U22	Y23	V22
Y30	U30	Y31	V30	Y32	U32	Y33	V32

YUYV

macro pixel

U00	Y00	V00	Y01	U02	Y02	Y03	V02
U10	Y10	V10	Y11	U12	Y12	Y13	V12
U20	Y20	V20	Y21	U22	Y22	Y23	V22
U30	Y30	V30	Y31	U32	Y32	Y33	V32

UYVY

macro pixel

Y00	Y01	Y02	Y03
Y10	Y11	Y12	Y13
Y20	Y21	Y22	Y23
Y30	Y31	Y32	Y33
U00	V00	U02	V02
U20	V20	U22	V22

NV12

macro pixel

Y00	Y01	Y02	Y03
Y10	Y11	Y12	Y13
Y20	Y21	Y22	Y23
Y30	Y31	Y32	Y33
V00	U00	V02	U02
V20	U20	V22	U22

NV21

macro pixel

Y00	Y01	Y02	Y03
Y10	Y11	Y12	Y13
Y20	Y21	Y22	Y23
Y30	Y31	Y32	Y33
U00	U02	U20	U22
V00	V02	V20	V22

YU12

macro pixel

Y00	Y01	Y02	Y03
Y10	Y11	Y12	Y13
Y20	Y21	Y22	Y23
Y30	Y31	Y32	Y33
V00	V02	V20	V22
U00	U02	U20	U22

YV12

CvTracker class description

CvTracker class declaration

CvTracker interface class declared in **CvTracker.h** file. Class declaration:

```
class CvTracker : public VTracker
{
public:

    /// Get string of the library version.
    static std::string getVersion();

    /// Class constructor.
    CvTracker();

    /// Class destructor.
    ~CvTracker();

    /// Init video tracker. All params will be set.
    bool initVTracker(VTrackerParams& params) override;

    /// Set video tracker param.
    bool setParam(VTrackerParam id, float value) override;

    /// Get video tracker parameter value.
    float getParam(VTrackerParam id) override;

    /// Get video tracker params (results).
    void getParams(VTrackerParams& params) override;

    /// Execute command.
    bool executeCommand(VTrackerCommand id,
                        float arg1 = 0,
                        float arg2 = 0,
                        float arg3 = 0) override;

    /// Process frame. Must be used for each input video frame.
    bool processFrame(cr::video::Frame& frame) override;

    /// Get image of internal surfaces.
    void getImage(int type, cr::video::Frame& image) override;

    /// Decode and execute command.
    bool decodeAndExecuteCommand(uint8_t* data, int size) override;
};
```

How to use the library

The library consists of a few source code files. To use the library, the developer must include the library files in C++ project. The **CvTracker.h** file contains the declaration of the **CvTracker** C++ class, which implements the tracking algorithm. The way of use of the library:

1. Connect the library files and OpenCV library to your project.
2. Create an instance of the **CvTracker** C++ class.
3. Set parameters of the library through calling the **setParam(...)** method, if necessary. it is thread-safe.
4. Call the **processFrame(...)** method to process the each video frame.
5. To perform commands use **executeCommand(...)** method. it is thread-safe.
6. The **getParams(...)** method is used to get the tracking results.

getVersion method

getVersion() method returns string of current version of **CvTracker** class. Method declaration:

```
static std::string getVersion();
```

Method can be used without **CvTracker** class instance:

```
cout << "CvTracker class version: " << CvTracker::getVersion() << endl;
```

Console output:

```
CvTracker class version: 9.0.0
```

initVTracker method

initVTracker(...) method initializes video tracker by set of params. Method declaration:

```
bool initVTracker(VTrackerParams& params) override;
```

Parameter	Value
params	Video tracker params class. The library initializes all supported parameters listed in [VTrackerParams class](). **VTrackerParams** class provided by **VTracker** interface class.

Returns: TRUE if the video tracker initialized or FALSE if not.

setParam method

setParam(...) method sets new video tracker parameter value. It is thread-safe method. This means that the **setParam(...)** method can be safely called from any thread. Method declaration:

```
bool setParam(VTrackerParam id, float value) override;
```

Parameter	Description
id	Parameter ID according to VTrackerParam enum. **VTrackerParam enum** provided by **VTracker** interface class. Not all parameters supported by CvTracker.
value	Parameter value. Value depends on parameter ID.

Returns: TRUE if the parameter was set or FALSE if not (not supported or out of valid range).

getParam method

getParam(...) method returns video tracker parameter value. It is thread-safe method. This means that the **getParam(...)** method can be safely called from any thread. Method declaration:

```
float getParam(VTrackerParam id) override;
```

Parameter	Description
id	Parameter ID according to VTrackerParam enum. **VTrackerParam enum** provided by **VTracker** interface class. Not all parameters supported by CvTracker.

Returns: parameter value or -1 if the parameters not supported by CvTracker.

getParams method

getParams(...) method returns video tracker params class and tracking results. It is thread-safe method. This means that the **getParams(...)** method can be safely called from any thread. Method declaration:

```
void getParams(VTrackerParams& params) override;
```

Parameter	Description
params	Video tracker parameters class object (VTrackerParams class). **VTrackerParams** class provided by **VTracker** interface class.

executeCommand method

executeCommand(...) method executes video tracker command. It is thread-safe **executeCommand(...)** method. This means that the **executeCommand(...)** method can be safely called from any thread. Method declaration:

```
bool executeCommand(VTrackerCommand id, float arg1 = 0, float arg2 = 0, float arg3 = 0)
override;
```

Parameter	Description
id	Command ID according to VTrackerCommand enum. **VTrackerCommand enum** provided by **VTracker** interface class.
arg1	First argument. Value depends on command ID (see VTrackerCommand enum description).
arg2	Second argument. Value depends on command ID (see VTrackerCommand enum description).
arg3	Third argument. Value depends on command ID (see VTrackerCommand enum description).

Returns: TRUE is the command was executed or FALSE if not.

processFrame method

processFrame(...) method processes video frame. To get actual tracking results use **getParams(...)** method. Method declaration:

```
bool processFrame(cr::video::Frame& frame) override;
```

Parameter	Description
frame	Video frame for processing. Video tracker processes only RAW pixel formats (BGR24, RGB24, GRAY, YUYV24, YUYV, UYVY, NV12, NV21, YV12, YU12, see Frame class description).

Returns: TRUE is the video frame was processed FALSE if not. If video tracker not in tracking mode the method should return TRUE.

getImage method

getImage(...) method designed to obtain images of internal matrixes. Method declaration:

```
void getImage(int type, cr::video::Frame& image) override;
```

Parameter	Description
type	The type of image to be obtained. The following are available to the user: 0 - reference image of an object. 1 - object mask image. 2 - image of the distribution surface of the probability of object detection.
frame	Output frame (see **Frame** class description). Pixel format depends should be **GRAY**, size of image must be 256x256 pixels.

decodeAndExecuteCommand method

decodeAndExecuteCommand(...) method decodes and executes command on video tracker side. It is thread-safe method. This means that the **decodeAndExecuteCommand(...)** method can be safely called from any thread. The methods only decodes commands encoded by **encodeSetParamCommand(...)** and **encodeCommand(...)** methods of **VTracker** interface class. Method declaration:

```
bool decodeAndExecuteCommand(uint8_t* data, int size) override;
```

Parameter	Description
data	Pointer to input command.
size	Size of command. Must be 11 bytes for SET_PARAM or 19 bytes for COMMAND.

Returns: TRUE if command decoded (SET_PARAM or COMMAND) and executed (action command or set param command).

encodeSetParamCommand method of VTracker class

encodeSetParamCommand(...) static method designed to encode command to change any parameter for remote video tracker. To control video tracker remotely, the developer has to design his own protocol and according to it encode the command and deliver it over the communication channel. To simplify this, the **VTracker** class contains static methods for encoding the control command. The **VTracker** class provides two types of commands: a parameter change command (SET_PARAM) and an action command (COMMAND). **encodeSetParamCommand(...)** designed to encode SET_PARAM command. Method declaration:

```
static void encodeSetParamCommand(uint8_t* data, int& size, VTrackerParam id, float
value);
```

Parameter	Description
data	Pointer to data buffer for encoded command. Must have size >= 11.
size	Size of encoded data. Will be 11 bytes.
id	Parameter ID according to **VTrackerParam enum**.

Parameter	Description
value	Parameter value. Value depends on parameter ID (see VTrackerParam enum description).

SET_PARAM command format (11 bytes):

Byte	Value	Description
0	0x01	SET_PARAM command header value.
1	0x01	Major version of VTracker class. Can be vary.
2	0x04	Minor version of VTracker class. Can be vary.
3	id	Parameter ID **int32_t** in Little-endian format.
4	id	Parameter ID **int32_t** in Little-endian format.
5	id	Parameter ID **int32_t** in Little-endian format.
6	id	Parameter ID **int32_t** in Little-endian format.
7	value	Parameter value **float** in Little-endian format.
8	value	Parameter value **float** in Little-endian format.
9	value	Parameter value **float** in Little-endian format.
10	value	Parameter value **float** in Little-endian format.

encodeSetParamCommand(...) is static and used without **VTracker** class instance. This method used on client side (control system). Command encoding example:

```
outValue = (float)(rand() % 20);
VTracker::encodeSetParamCommand(
data, size, VTrackerParam::MULTIPLE_THREADS, outValue);
```

encodeCommand method of VTracker class

encodeCommand(...) static method designed to encode command for remote video tracker. To control video tracker remotely, the developer has to design his own protocol and according to it encode the command and deliver it over the communication channel. To simplify this, the **VTracker** class contains static methods for encoding the control command. The **VTracker** class provides two types of commands: a parameter change command (SET_PARAM) and an action command (COMMAND). **encodeCommand(...)** designed to encode COMMAND (action command). Method declaration:

```
static void encodeCommand(uint8_t* data, int& size, VTrackerCommand id, float arg1 = 0,
float arg2 = 0, float arg3 = 0);
```

Parameter	Description
data	Pointer to data buffer for encoded command. Must have size >= 19.
size	Size of encoded data. Will be 19 bytes.
id	Command ID according to **VTrackerCommand enum**.
arg1	First argument. Value depends on command ID (see **VTrackerCommand enum** description).
arg2	Second argument. Value depends on command ID (see **VTrackerCommand enum** description).
arg3	Third argument. Value depends on command ID (see **VTrackerCommand enum** description).

COMMAND format format (19 bytes):

Byte	Value	Description
0	0x00	COMMAND header value.
1	0x01	Major version of VTracker class. Can be vary.
2	0x04	Minor version of VTracker class. Can be vary.
3	id	Command ID **int32_t** in Little-endian format.
4	id	Command ID **int32_t** in Little-endian format.
5	id	Command ID **int32_t** in Little-endian format.
6	id	Command ID **int32_t** in Little-endian format.
7	arg1	First command argument value **float** in Little-endian format.
8	arg1	First command argument value **float** in Little-endian format.
9	arg1	First command argument value **float** in Little-endian format.
10	arg1	First command argument value **float** in Little-endian format.
11	arg2	Second command argument value **float** in Little-endian format.
12	arg2	Second command argument value **float** in Little-endian format.
13	arg2	Second command argument value **float** in Little-endian format.
14	arg2	Second command argument value **float** in Little-endian format.
15	arg3	Third command argument value **float** in Little-endian format.
16	arg3	Third command argument value **float** in Little-endian format.
17	arg3	Third command argument value **float** in Little-endian format.
18	arg3	Third command argument value **float** in Little-endian format.

encodeCommand(...) is static and used without **VTracker** class instance. This method used on client side (control system). Command encoding example:

```
uint8_t data[1024];
int size = 0;
float outValue = (float)(rand() % 20);
float arg1 = (float)(rand() % 20);
float arg2 = (float)(rand() % 20);
float arg3 = (float)(rand() % 20);
VTracker::encodeCommand(data, size, VTrackerCommand::CAPTURE, arg1, arg2, arg3);
```

Data structures

VTrackerCommand enum

VTrackerCommand enum describes commands supported by **VTracker** interface class. **VTrackerCommand** enum declared in **VTracker.h** file. Enum declaration:

```
enum class VTrackerCommand
{
    /// Object capture. Arguments:
    /// arg1 - Capture rectangle center X coordinate. -1 - default coordinate.
    /// arg2 - Capture rectangle center Y coordinate. -1 - default coordinate.
    /// arg3 - frame ID. -1 - Capture on last frame.
    CAPTURE = 1,
    /// Object capture command. Arguments:
    /// arg1 - Capture rectangle center X coordinate, percents of frame width.
    /// arg2 - Capture rectangle center Y coordinate, percents of frame width.
    CAPTURE_PERCENTS,
    /// Reset command. No arguments.
    RESET,
    /// Set INERTIAL mode. No arguments.
    SET_INERTIAL_MODE,
    /// Set LOST mode. No arguments.
    SET_LOST_MODE,
    /// Set STATIC mode. No arguments.
    SET_STATIC_MODE,
    /// Adjust tracking rectangle size automatically once. No arguments.
    ADJUST_RECT_SIZE,
    /// Adjust tracking rectangle position automatically once. No arguments.
    ADJUST_RECT_POSITION,
    /// Move tracking rectangle (change position). Arguments:
    /// arg1 - add to X coordinate, pixels. 0 - no change.
    /// arg2 - add to Y coordinate, pixels. 0 - no change.
    MOVE_RECT,
    /// Set tracking rectangle position in FREE mode. Arguments:
    /// arg1 - Rectangle center X coordinate.
    /// arg2 - Rectangle center Y coordinate.
    SET_RECT_POSITION,
    /// Set tracking rectangle position in FREE mode in percents of frame size.
```

```
    /// Arguments:
    /// arg1 - Rectangle center X coordinate, percents of frame width.
    /// arg2 - Rectangle center X coordinate, percents of frame height.
    SET_RECT_POSITION_PERCENTS,
    /// Move search window (change position). Arguments:
    /// arg1 - add to X coordinate, pixels. 0 - no change.
    /// arg2 - add to Y coordinate, pixels. 0 - no change.
    MOVE_SEARCH_WINDOW,
    /// Set search window position. Arguments:
    /// arg1 - Search window center X coordinate.
    /// arg2 - Search window center Y coordinate.
    SET_SEARCH_WINDOW_POSITION,
    /// Set search window position in percents of frame size. Arguments:
    /// arg1 - Search window center X coordinate, percents of frame width.
    /// arg2 - Search window center X coordinate, percents of frame height.
    SET_SEARCH_WINDOW_POSITION_PERCENTS,
    /// Change tracking rectangle size. Arguments:
    /// arg1 - horizontal size add, pixels.
    /// arg2 - vertical size add, pixels.
    CHANGE_RECT_SIZE
};
```

Table 5 - Video tracker commands description and description of necessary arguments for **executeCommand(...)** and **encodeCommand(...)** methods described.

Command	Description
CAPTURE	Object capture. **Arguments:** arg1 - Capture rectangle center X coordinate. -1 - default coordinate, arg2 - Capture rectangle center Y coordinate. -1 - default coordinate, arg3 - frame ID. -1 - Capture on last frame.
CAPTURE_PERCENTS	Object capture command. **Arguments:** arg1 - Capture rectangle center X coordinate, percents of frame width, arg2 - Capture rectangle center Y coordinate, percents of frame width.
RESET	Reset command. **No arguments.**
SET_INERTIAL_MODE	Set INERTIAL mode. **No arguments.**
SET_LOST_MODE	Set LOST mode. **No arguments.**
SET_STATIC_MODE	Set STATIC mode. **No arguments.**
ADJUST_RECT_SIZE	Adjust tracking rectangle size automatically once. **No arguments.**
ADJUST_RECT_POSITION	Adjust tracking rectangle position automatically once. **No arguments.**

Command	Description
MOVE_RECT	Move tracking rectangle (change position). **Arguments:** arg1 - add to X coordinate, pixels. 0 - no change, arg2 - add to Y coordinate, pixels. 0 - no change.
SET_RECT_POSITION	Set tracking rectangle position in FREE mode. **Arguments:** arg1 - Rectangle center X coordinate, arg2 - Rectangle center Y coordinate.
SET_RECT_POSITION_PERCENTS	Set tracking rectangle position in FREE mode in percent of frame size. **Arguments:** arg1 - Rectangle center X coordinate, percent of frame width, arg2 - Rectangle center X coordinate, percent of frame height.
MOVE_SEARCH_WINDOW	Move search window (change position). **Arguments:** arg1 - add to X coordinate, pixels. 0 - no change, arg2 - add to Y coordinate, pixels. 0 - no change.
SET_SEARCH_WINDOW_POSITION	Set search window position. **Arguments:** arg1 - Search window center X coordinate, arg2 - Search window center Y coordinate.
SET_SEARCH_WINDOW_POSITION_PERCENTS	Set search window position in percent of frame size. **Arguments:** arg1 - Search window center X coordinate, percent of frame width, arg2 - Search window center X coordinate, percent of frame height.
CHANGE_RECT_SIZE	Change tracking rectangle size. **Arguments:** arg1 - horizontal size add, pixels, arg2 - vertical size add, pixels.

VTrackerParam enum

VTrackerParam enum describes parameters supported by **VTracker** interface. **VTrackerParam** enum declared in **VTracker.h** file. Enum declaration:

```
enum class VTrackerParam
{
    /// Width of search window, pixels. Set by user.
    SEARCH_WINDOW_WIDTH = 1,
    /// Height of search window, pixels. Set by user.
    SEARCH_WINDOW_HEIGHT,
    /// Tracking rectangle width, pixels. Set by user or can be changed by
    /// tracking algorithm if rectAutoSize == true.
    RECT_WIDTH,
    /// Tracking rectangle height, pixels. Set by user or can be changed by
    /// tracking algorithm if rectAutoSize == true.
    RECT_HEIGHT,
    /// Option for LOST mode. Parameter that defines the behavior of the
```

```
    /// tracking  algorithm in LOST mode. Default is 0. Possible values:
    /// 0. In LOST mode, the coordinates of the center of the  tracking
    ///    rectangle are not updated and remain the  same as before entering
    ///    LOST mode.
    /// 1. The coordinates of the center of the tracking  rectangle are updated
    ///    based on the components  of the object's speed calculated before
    ///    going into  LOST mode. If the tracking rectangle "touches" the  edge
    ///    of the video frame, the coordinate updating  for this component
    ///    (horizontal or vertical) will stop.
    /// 2. The coordinates of the center of the tracking  rectangle are
    ///    updated based on the components  of the object's speed calculated
    ///    before going into  LOST mode. The tracking is reset if the center of
    ///    the tracking rectangle touches any of the edges of  the video frame.
    LOST_MODE_OPTION,
    /// Size of frame buffer (number of frames to store). Set by user.
    FRAME_BUFFER_SIZE,
    /// Maximum number of frames in LOST mode to auto reset of algorithm. Set
    /// by user.
    MAX_FRAMES_IN_LOST_MODE,
    /// Use tracking rectangle auto size flag: 0 - no use, 1 - use. Set by user.
    RECT_AUTO_SIZE,
    /// Use tracking rectangle auto position: 0 - no use, 1 - use. Set by user.
    RECT_AUTO_POSITION,
    /// Use multiple threads for calculations. 0 - one thread, 1 - multiple. Set
    /// by user.
    MULTIPLE_THREADS,
    /// Number of channels for processing. E.g., number of color channels. Set
    /// by user.
    NUM_CHANNELS,
    /// Tracker type. Depends on implementation. Set by user.
    TYPE,
    /// Custom parameter. Depends on implementation.
    CUSTOM_1,
    /// Custom parameter. Depends on implementation.
    CUSTOM_2,
    /// Custom parameter. Depends on implementation.
    CUSTOM_3
};
```

Table 6 - Video tracker params description. Some params maybe unsupported by particular video tracker.

Parameter	Access	Description
SEARCH_WINDOW_WIDTH	read / write	Width of search window, pixels. Set by user.
SEARCH_WINDOW_HEIGHT	read / write	Height of search window, pixels. Set by user.
RECT_WIDTH	read / write	Tracking rectangle width, pixels. Set by user or can be changed by tracking algorithm if RECT_AUTO_SIZE == 1.
RECT_HEIGHT	read / write	Tracking rectangle height, pixels. Set by user or can be changed by tracking algorithm if RECT_AUTO_SIZE == 1.

Parameter	Access	Description
LOST_MODE_OPTION	read / write	Option for LOST mode. Parameter that defines the behavior of the tracking algorithm in LOST mode. Default is 0. Possible values: **0.** In LOST mode, the coordinates of the center of the tracking rectangle are not updated and remain the same as before entering LOST mode. **1.** The coordinates of the center of the tracking rectangle are updated based on the components of the object’s speed calculated before going into LOST mode. If the tracking rectangle “touches” the edge of the video frame, the coordinate updating for this component (horizontal or vertical) will stop. **2.** The coordinates of the center of the tracking rectangle are updated based on the components of the object’s speed calculated before going into LOST mode. The tracking is reset if the center of the tracking rectangle touches any of the edges of the video frame.
FRAME_BUFFER_SIZE	read / write	Size of frame buffer (number of frames to store). Set by user.
MAX_FRAMES_IN_LOST_MODE	read / write	Maximum number of frames in LOST mode to auto reset of algorithm. Set by user.
RECT_AUTO_SIZE	read / write	Use tracking rectangle auto size flag: 0 - no use, 1 - use. Set by user.
RECT_AUTO_POSITION	read / write	Use tracking rectangle auto position: 0 - no use, 1 - use. Set by user.
MULTIPLE_THREADS	read / write	Use multiple threads for calculations. 0 - one thread, 1 - multiple. Set by user.
NUM_CHANNELS	read / write	Number of channels for processing. E.g., number of color channels. Set by user.
TYPE	read / write	Not supported by CvTracker.
CUSTOM_1	read / write	Not supported by CvTracker.
CUSTOM_2	read / write	Not supported by CvTracker.
CUSTOM_3	read / write	Not supported by CvTracker.

VTrackerParams class description

VTrackerParams class declaration

VTrackerParams class used for video tracker initialization (**initVTracker(...)** method) or to get all actual params and tracking results (**getParams()** method). Also **VTrackerParams** provide structure to write/read params from JSON files (**JSON_READABLE** macro, see **ConfigReader** class description) and provides methos to encode and decode params. Class declaration:

```
class VTrackerParams
{
public:
    /// Tracker mode index: 0 - FREE, 1 - TRACKING, 2 - INERTIAL, 3 - STATIC.
    /// Set by video tracker according to processing results or after command
    /// execution.
    int mode{0};
    /// Tracking rectangle horizontal center position. Calculated by tracking
    /// algorithm.
    int rectX{0};
    /// Tracking rectangle vertical center position. Calculated by tracking
    /// algorithm.
    int rectY{0};
    /// Tracking rectangle width, pixels. Set by user or can be changed by
    /// tracking algorithm if rectAutoSize == true.
    int rectWidth{72};
    /// Tracking rectangle height, pixels. Set by user or can be changed by
    /// tracking algorithm if rectAutoSize == true.
    int rectHeight{72};
    /// Estimated horizontal position of object center, pixels. Calculated by
    /// video tracker.
    int objectX{0};
    /// Estimated vertical position of object center, pixels.Calculated by
    /// video tracker.
    int objectY{0};
    /// Estimated object width, pixels. Calculated by video tracker.
    int objectWidth{72};
    /// Estimated object height, pixels. Calculated by video tracker.
    int objectHeight{72};
    /// Frame counter in LOST mode. After switching in LOST mode the video
    /// tracker start counting from 0. After switching to another mode from
    /// LOST mode the video tracker will reset this counter.
    int lostModeFrameCounter{0};
    /// Counter for processed frames after capture object. After reset tracking
    /// the video tracker will reset counter.
    int frameCounter{0};
    /// Width of processed video frame. Set by video tracker after processing
    /// first video frame.
    int frameWidth{0};
    /// Height of processed video frame. Set by video tracker after processing
    /// first video frame.
```

```
    int frameHeight{0};
    /// Width of search window, pixels. Set by user.
    int searchWindowWidth{256};
    /// Height of search window, pixels. Set by user.
    int searchWindowHeight{256};
    /// Horizontal position of search window center. This position will be used
    /// for next video frame. Usually it coincides with data tracking rectangle
    /// center but can be set by user to move search window for new video frame.
    int searchWindowX{0};
    /// Vertical position of search window center. This position will be used
    /// for next video frame. Usually it coinsides with data tracking rectangle
    /// center but can be set by user to move search window for new video frame.
    int searchWindowY{0};
    /// Option for LOST mode. Parameter that defines the behavior of the
    /// tracking  algorithm in LOST mode. Default is 0. Possible values:
    /// 0. In LOST mode, the coordinates of the center of the  tracking
    ///    rectangle are not updated and remain the  same as before entering
    ///    LOST mode.
    /// 1. The coordinates of the center of the tracking  rectangle are updated
    ///    based on the components  of the object’s speed calculated before
    ///    going into  LOST mode. If the tracking rectangle “touches” the  edge
    ///    of the video frame, the coordinate updating  for this component
    ///    (horizontal or vertical) will stop.
    /// 2. The coordinates of the center of the tracking  rectangle are
    ///    updated based on the components  of the object’s speed calculated
    ///    before going into  LOST mode. The tracking is reset if the center of
    ///    the tracking rectangle touches any of the edges of  the video frame.
    int lostModeOption{0};
    /// Size of frame buffer (number of frames to store). Set by user.
    int frameBufferSize{128};
    /// Maximum number of frames in LOST mode to auto reset of algorithm. Set
    /// by user.
    int maxFramesInLostMode{128};
    /// ID of last processed frame in frame buffer. Set by video tracker.
    int processedFrameId{0};
    /// ID of last added frame to frame buffer. Set by video tracker.
    int frameId{0};
    /// Horizontal velocity of object on video frames (pixel/frame). Calculated
    /// by video tracker.
    float velX{0.0f};
    /// Vertical velocity of object on video frames (pixel/frame). Calculated
    /// by video tracker.
    float velY{0.0f};
    /// Estimated probability of object detection. Calculated by video tracker.
    float detectionProbability{0.0f};
    /// Use tracking rectangle auto size flag: false - no use, true - use. Set
    /// by user.
    bool rectAutoSize{false};
    /// Use tracking rectangle auto position: false - no use, true - use. Set
    /// by user.
    bool rectAutoPosition{false};
    /// Use multiple threads for calculations. Set by user.
    bool multipleThreads{false};
    /// Number of channels for processing. E.g., number of color channels. Set
    /// by user.
```

```
    int numChannels{3};
    /// Tracker type. Depends on implementation. Set by user.
    int type{0};
    /// Processing time for last frame, mks. Calculated by video tracker.
    int processingTimeMks{0};
    /// Custom parameter. Depends on implementation.
    float custom1{0.0f};
    /// Custom parameter. Depends on implementation.
    float custom2{0.0f};
    /// Custom parameter. Depends on implementation.
    float custom3{0.0f};

    JSON_READABLE(VTrackerParams, rectWidth, rectHeight, searchWindowWidth,
                  searchWindowHeight, lostModeOption, frameBufferSize,
                  maxFramesInLostMode, rectAutoSize, rectAutoPosition,
                  multipleThreads, numChannels, type, custom1, custom2, custom3);

    /// operator =
    VTrackerParams& operator= (const VTrackerParams& src);

    /// Encode params.
    bool encode(uint8_t* data, int bufferSize,
                int& size, VTrackerParamsMask* mask = nullptr);

    /// Decode params.
    bool decode(uint8_t* data, int dataSize);
};
```

Table 7 - VTrackerParams class fields description.

Field	Type	Description
mode	int	Tracker mode index: **0** - FREE. **1** - TRACKING. **2** - INERTIAL. **3** - STATIC. Set by video tracker according to processing results or after command execution.
rectX	int	Tracking rectangle horizontal center position, pixels. Calculated by tracking algorithm.
rectY	int	Tracking rectangle vertical center position, pixels. Calculated by tracking algorithm.
rectWidth	int	Tracking rectangle width, pixels. Set by user or can be changed by tracking algorithm if rectAutoSize == true.
rectHeight	int	Tracking rectangle height, pixels. Set by user or can be changed by tracking algorithm if rectAutoSize == true.
objectX	int	Estimated horizontal position of object center, pixels. Calculated by video tracker.

Field	Type	Description
objectY	int	Estimated vertical position of object center, pixels. Calculated by video tracker.
objectWidth	int	Estimated object width, pixels. Calculated by video tracker.
objectHeight	int	Estimated object height, pixels. Calculated by video tracker.
lostModeFrameCounter	int	Frame counter in LOST mode. After switching in LOST mode the video tracker start counting from 0. After switching to another mode from LOST mode the video tracker will reset this counter.
frameCounter	int	Counter for processed frames after capture object. After reset tracking the video tracker will reset counter.
frameWidth	int	Width of processed video frame. Set by video tracker after processing first video frame.
frameHeight	int	Height of processed video frame. Set by video tracker after processing first video frame.
searchWindowWidth	int	Width of search window, pixels. Set by user.
searchWindowHeight	int	Height of search window, pixels. Set by user.
searchWindowX	int	Horizontal position of search window center. This position will be used for next video frame. Usually it coincides with data tracking rectangle center but can be set by user to move search window for new video frame.
searchWindowY	int	Vertical position of search window center. This position will be used for next video frame. Usually it coincides with data tracking rectangle center but can be set by user to move search window for new video frame.
lostModeOption	int	Option for LOST mode. Parameter that defines the behavior of the tracking algorithm in LOST mode. Default is 0. Possible values: **0.** In LOST mode, the coordinates of the center of the tracking rectangle are not updated and remain the same as before entering LOST mode. **1.** The coordinates of the center of the tracking rectangle are updated based on the components of the object's speed calculated before going into LOST mode. If the tracking rectangle "touches" the edge of the video frame, the coordinate updating for this component (horizontal or vertical) will stop. **2.** The coordinates of the center of the tracking rectangle are updated based on the components of the object's speed calculated before going into LOST mode. The tracking is reset if the center of the tracking rectangle touches any of the edges of the video frame.
frameBufferSize	int	Size of frame buffer (number of frames to store). Set by user.

Field	Type	Description
maxFramesInLostMode	int	Maximum number of frames in LOST mode to auto reset of algorithm. Set by user.
processedFrameId	int	ID of last processed frame in frame buffer. Set by video tracker.
frameId	int	ID of last added frame to frame buffer. Set by video tracker.
velX	float	Horizontal velocity of object on video frames (pixel/frame). Calculated by video tracker.
velY	float	Vertical velocity of object on video frames (pixel/frame). Calculated by video tracker.
detectionProbability	float	Estimated probability of object detection. Calculated by video tracker.
rectAutoSize	bool	Use tracking rectangle auto size flag: false - no use, true - use. Set by user.
rectAutoPosition	bool	Use tracking rectangle auto position: false - no use, true - use. Set by user.
multipleThreads	bool	Use multiple threads for calculations. Set by user.
numChannels	int	Number of channels for processing. E.g., number of color channels. Set by user.
type	int	Not supported by CvTracker.
processingTimeMks	int	Processing time for last frame, mks. Calculated by video tracker.
custom1	float	Not supported by CvTracker.
custom2	float	Not supported by CvTracker.
custom3	float	Not supported by CvTracker.

Serialize video tracker params

VTrackerParams class provides method **encode(...)** to serialize video tracker params (fields of VTrackerParams class, see Table 7). Serialization of params necessary in case when you need to send params via communication channels. Method provides options to exclude particular parameters from serialization. To do this method inserts binary mask (5 bytes) where each bit represents particular parameter and **decode(...)** method recognizes it. Method doesn't encode initString. Method declaration:

```
bool encode(uint8_t* data, int bufferSize, int& size, VTrackerParamsMask* mask =
nullptr);
```

Parameter	Value
data	Pointer to data buffer. Buffer size should be at least **135** bytes.

Parameter	Value
bufferSize	Data buffer size. Buffer size should be at least **135** bytes.
size	Size of encoded data.
mask	Parameters mask - pointer to **VrackerParamsMask** structure. **VTrackerParamsMask** (declared in VTracker.h file) determines flags for each field (parameter) declared in [VTrackerParams class](). If the user wants to exclude any parameters from serialization, he can put a pointer to the mask. If the user wants to exclude a particular parameter from serialization, he should set the corresponding flag in the **VTrackerParamsMask** structure.

VTrackerParamsMask structure declaration:

```
typedef struct VTrackerParamsMask
{
    bool mode{true};
    bool rectX{true};
    bool rectY{true};
    bool rectWidth{true};
    bool rectHeight{true};
    bool objectX{true};
    bool objectY{true};
    bool objectWidth{true};
    bool objectHeight{true};
    bool lostModeFrameCounter{true};
    bool frameCounter{true};
    bool frameWidth{true};
    bool frameHeight{true};
    bool searchWindowWidth{true};
    bool searchWindowHeight{true};
    bool searchWindowX{true};
    bool searchWindowY{true};
    bool lostModeOption{true};
    bool frameBufferSize{true};
    bool maxFramesInLostMode{true};
    bool processedFrameId{true};
    bool frameId{true};
    bool velX{true};
    bool velY{true};
    bool detectionProbability{true};
    bool rectAutoSize{true};
    bool rectAutoPosition{true};
    bool multipleThreads{true};
    bool numChannels{true};
    bool type{true};
    bool processingTimeMks{true};
    bool custom1{true};
    bool custom2{true};
    bool custom3{true};
} VTrackerParamsMask;
```

Example without parameters mask:

```
// Prepare random params.
VTrackerParams in;
in.mode = rand() % 255;
in.rectX = rand() % 255;
in.rectY = rand() % 255;

// Encode data.
uint8_t data[1024];
int size = 0;
in.encode(data, 1024, size);
cout << "Encoded data size: " << size << " bytes" << endl;
```

Example with parameters mask:

```
// Prepare random params.
VTrackerParams in;
in.mode = rand() % 255;
in.rectX = rand() % 255;
in.rectY = rand() % 255;

// Prepare mask.
VTrackerParamsMask mask;
mask.mode = true;
mask.rectX = false;
mask.rectY = true;

// Encode data.
uint8_t data[1024];
int size = 0;
in.encode(data, 1024, size, &mask);
cout << "Encoded data size: " << size << " bytes" << endl;
```

Deserialize video tracker params

[VTrackerParams class](#) provides method **decode(...)** to deserialize params (fields of VTrackerParams class, see Table 5). Deserialization of params necessary in case when you need to receive params via communication channels. Method automatically recognizes which parameters were serialized by **encode(...)** method. Method declaration:

```
bool decode(uint8_t* data, int dataSize);
```

Parameter	Value
data	Pointer to encode data buffer.

Returns: TRUE if data decoded (deserialized) or FALSE if not.

Example:

```
// Prepare random params.
VTrackerParams in;
```

```
in.mode = rand() % 255;
in.rectX = rand() % 255;
in.rectY = rand() % 255;

// Encode data.
uint8_t data[1024];
int size = 0;
in.encode(data, 1024, size);
cout << "Encoded data size: " << size << " bytes" << endl;

// Decode data.
VTrackerParams out;
if (!out.decode(data, size))
{
    cout << "Can't decode data" << endl;
    return false;
}
```

Read params from JSON file and write to JSON file

VTracker library depends on **ConfigReader** library which provides method to read params from JSON file and to write params to JSON file. Example of writing and reading params to JSON file:

```
// Prepare random params.
VTrackerParams in;
in.mode = rand() % 255;
in.rectX = rand() % 255;
in.rectY = rand() % 255;

// Write params to file.
cr::utils::ConfigReader inConfig;
inConfig.set(in, "VTrackerParams");
inConfig.writeToFile("VTrackerParams.json");

// Read params from file.
cr::utils::ConfigReader outConfig;
if(!outConfig.readFromFile("VTrackerParams.json"))
{
    cout << "Can't open config file" << endl;
    return false;
}

// Obtain params from config reader.
VTrackerParams out;
if(!outConfig.get(out, "VTrackerParams"))
{
    cout << "Can't read params from file" << endl;
    return false;
}
```

VTrackerParams.json will look like:

```json
{
    "VTrackerParams": {
        "custom1": 155.0,
        "custom2": 239.0,
        "custom3": 79.0,
        "frameBufferSize": 130,
        "lostModeOption": 65,
        "maxFramesInLostMode": 111,
        "multipleThreads": true,
        "numChannels": 47,
        "rectAutoPosition": true,
        "rectAutoSize": true,
        "rectHeight": 18,
        "rectWidth": 2,
        "searchWindowHeight": 19,
        "searchWindowWidth": 195,
        "type": 213
    }
}
```

Build and connect to your project

Before build you have to install OpenCv libraries (version >= 4.5) and build environment. Installation commands for Linux (Ubuntu):

```
sudo apt-get install build-essential libopencv-dev cmake
```

Typical commands to build **CvTracker** library:

```
cd CvTracker
git submodule update --init --recursive
mkdir build
cd build
cmake ..
make
```

If you want connect **CvTracker** library to your CMake project as source code you can make follow. For example, if your repository has structure:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
```

Create folder **3rdparty** in your repository and copy CvTracker repository folder in **3rdparty** folder. New structure of your repository:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    CvTracker
```

Create CMakeLists.txt file in **3rdparty** folder. CMakeLists.txt should contain:

```
cmake_minimum_required(VERSION 3.13)

################################################################################
## 3RD-PARTY
## dependencies for the project
################################################################################
project(3rdparty LANGUAGES CXX)

################################################################################
## SETTINGS
## basic 3rd-party settings before use
################################################################################
# To inherit the top-level architecture when the project is used as a submodule.
SET(PARENT ${PARENT}_YOUR_PROJECT_3RDPARTY)
# Disable self-overwriting of parameters inside included subdirectories.
SET(${PARENT}_SUBMODULE_CACHE_OVERWRITE OFF CACHE BOOL "" FORCE)

################################################################################
## CONFIGURATION
## 3rd-party submodules configuration
################################################################################
SET(${PARENT}_SUBMODULE_CVTRACKER                     ON  CACHE BOOL "" FORCE)
if (${PARENT}_SUBMODULE_CVTRACKER)
    SET(${PARENT}_CVTRACKER                           ON  CACHE BOOL "" FORCE)
    SET(${PARENT}_CVTRACKER_BENCHMARK                 OFF CACHE BOOL "" FORCE)
    SET(${PARENT}_CVTRACKER_DEMO                      OFF CACHE BOOL "" FORCE)
    SET(${PARENT}_CVTRACKER_EXAMPLE                   OFF CACHE BOOL "" FORCE)
endif()

################################################################################
## INCLUDING SUBDIRECTORIES
## Adding subdirectories according to the 3rd-party configuration
################################################################################
if (${PARENT}_SUBMODULE_CVTRACKER)
    add_subdirectory(CvTracker)
endif()
```

File **3rdparty/CMakeLists.txt** adds folder **CvTracker** to your project and excludes test application (VTracker class test applications) from compiling. Your repository new structure will be:

```
CMakeLists.txt
src
    CMakeList.txt
    yourLib.h
    yourLib.cpp
3rdparty
    CMakeLists.txt
    CvTracker
```

Next you need include folder 3rdparty in main **CMakeLists.txt** file of your repository. Add string at the end of your main **CMakeLists.txt**:

```
add_subdirectory(3rdparty)
```

Next you have to include CvTracker library in your **src/CMakeLists.txt** file:

```
target_link_libraries(${PROJECT_NAME} CvTracker)
```

Done!

Simple example

Simple example shows how to build basic application with user interface (based on OpenCV). Simple application opens video source (it can be video file, video stream or camera) and provides basic functions to control tracker (capture, reset, changing parameters). Source code:

```
#include <opencv2/opencv.hpp>
#include "CvTracker.h"

// Link namespaces.
using namespace cv;
using namespace std;
using namespace cr::video;
using namespace cr::vtracker;

/// Video tracker object.
CvTracker g_tracker;
/// Frame width.
int g_frameWidth = 0;
/// Frame height.
int g_frameHeight = 0;

// Prototype of mouse callback function.
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata);

// Entry point.
int main(void)
{
    cout << "=======================================================" << endl;
    cout << "Simpe Demo Application v" << CvTracker::getVersion() << endl;
    cout << "=======================================================" << endl;
```

```
// Set video source.
string initString = "0";
cout << "Enter video source init string "
<< "(camera num, rtsp string, video file): ";
cin >> initString;
cout << initString << endl;

// Open video source.
VideoCapture videoSource;
if (initString.size() < 4)
{
    // Open camera.
    if (!videoSource.open(stoi(initString)))
    {
        cout << "ERROR: Camera not open" << endl;
        return -1;
    }
}
else
{
    // Open video source.
    if (!videoSource.open(initString))
    {
        cout << "ERROR: Video source not open" << endl;
        return -1;
    }
}

// Init variables.
Mat frame;
VTrackerParams trackerData;

// Set initial tracking rectangle size.
g_tracker.setParam(VTrackerParam::RECT_WIDTH, 72);
g_tracker.setParam(VTrackerParam::RECT_HEIGHT, 72);
g_tracker.setParam(VTrackerParam::LOST_MODE_OPTION, 0);
g_tracker.setParam(VTrackerParam::NUM_CHANNELS, 3);
g_tracker.setParam(VTrackerParam::MULTIPLE_THREADS, 1);

// Init OpenCV window.
namedWindow("Simple demo application", WINDOW_AUTOSIZE);
// Set mouse callback.
setMouseCallback("Simple demo application", MouseCallBackFunc, nullptr);

// Main loop.
while (true)
{
    // Capture next video frame.
    videoSource >> frame;
    if (frame.empty())
    {
        // If we have video file we can set initial position to replay.
        videoSource.set(CAP_PROP_POS_FRAMES, 0);
        continue;
```

```
    }

    // Update frame size.
    g_frameWidth = frame.size().width;
    g_frameHeight = frame.size().height;

    // Create frame object.
    Frame trackerFrame(g_frameWidth, g_frameHeight, Fourcc::YUV24);
    Mat yuvFrame(g_frameHeight, g_frameWidth, CV_8UC3, trackerFrame.data);

    // Convert to YUV format.
    cvtColor(frame, yuvFrame, COLOR_BGR2YUV);

    // Process video frame.
    g_tracker.processFrame(trackerFrame);

    // Get current tracker data.
    g_tracker.getParams(trackerData);

    // Choose tracking rectangle color according to tracker mode.
    Scalar rectColor;
    switch (trackerData.mode) {
    case 0: rectColor = Scalar(255, 255, 255); break;
    case 1: rectColor = Scalar(0, 0, 255); break;
    case 2: rectColor = Scalar(255, 0, 0); break;
    default: rectColor = Scalar(255, 255, 255); break; }

    // Draw tracking rectangle.
    Rect rect(trackerData.rectX - trackerData.rectWidth / 2,
              trackerData.rectY - trackerData.rectHeight / 2,
              trackerData.rectWidth, trackerData.rectHeight);
    rectangle(frame, rect, rectColor, 2);

    // Show video with tracker result information.
    imshow("Simple demo application", frame);

    // Wait keyboard events.
    switch (waitKey(25))
    {
    // ESC - Exit.
    case 27:
        destroyAllWindows();
        return 1;
    // W - Increase tracking rectangle height.
    case 119:
        g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, 0, 8);
        break;
    // S - Decrease tracking rectangle height.
    case 115:
        g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, 0, -8);
        break;
    // D - Increase tracking rectangle width.
    case 100:
        g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, 8, 0);
        break;
```

```
        // A - Decrease tracking rectangle width.
        case 97:
            g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, -8, 0);
            break;
        // T - Move strobe UP (change position in TRACKING mode).
        case 116:
            g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, 0, 4);
            break;
        // G - Move strobe DOWN (change position in TRACKING mode).
        case 103:
            g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, 0, -4);
            break;
        // H - Move strobe RIGHT (change position in TRACKING mode).
        case 104:
            g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, -4, 0);
            break;
        // F - Move strobe LEFT (change position in TRACKING mode).
        case 102:
            g_tracker.executeCommand(VTrackerCommand::CHANGE_RECT_SIZE, 4, 0);
            break;
        }
    }
}

// Mouse callback function.
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata)
{
    // Set mouse position in any case.
    g_tracker.executeCommand(VTrackerCommand::SET_RECT_POSITION, x, y);

    switch (event)
    {
    /// Capture object.
    case cv::EVENT_LBUTTONDOWN:
        g_tracker.executeCommand(VTrackerCommand::CAPTURE, x, y);
        break;
    /// Reset tracker.
    case cv::EVENT_RBUTTONDOWN:
        g_tracker.executeCommand(VTrackerCommand::RESET);
        break;
    case cv::EVENT_MBUTTONDOWN:
        break;
    case cv::EVENT_MOUSEMOVE:
        break;
    }
}
```