VTracker interface C++ library

v1.0.1

Table of contents

Overview

VTracker C++ library provides standard interface as well defines data structures and rules for different video trackers. **VTracker** interface class doesn't do anything, just provides interface and defines data structures. Different video trackers inherit interface form **VTracker** C++ class. **VTracker.h** file contains **VTrackerParams** class, **VTrackerCommand** enum, **VTrackerParam** enum and includes **VTracker** class declaration. **VTrackerParams** class contains video tracker params and includes methods to encode and decode params. **VTrackerCommand** enum contains IDs of commands. **VTrackerParam** enum contains IDs of params. All video trackers should include params and commands listed in **VTracker.h** file. Class dependency: **Frame** class which describes video frame structure and pixel formats, **ConfigReader** class which provides methods to work with JSON structures (read/write).

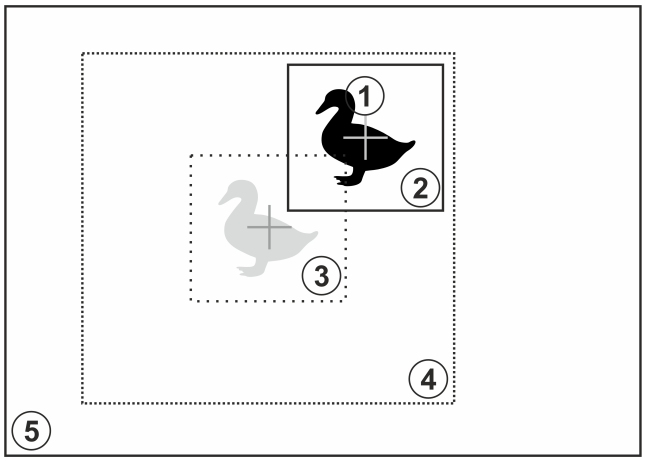
Versions

Table 1 - Library versions.

Version	Release date	What's new
1.0.0	19.07.2023	First version.

Required operating principles

The video tracker shall provide 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.



VTracker interface class description

VTracker class declaration

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

```
class VTracker
{
public:
    /**
     * @brief Get string of current library version.
     * @return String of current library version.
     */
    static std::string getVersion();
    /**
     * @brief Init video tracker. All params will be set.
     * @param params Parameters class.
     * @return TRUE if the video tracker init or FALSE if not.
     */
    virtual bool initVTracker(VTrackerParams& params) = 0;
    /**
     * @brief Set video tracker param.
     * @param id Param ID.
     * @param value Param value to set.
     * @return TRUE if param was set of FALSE.
     */
    virtual bool setParam(VTrackerParam id, float value) = 0;
    /**
     * @brief Get video tracker param value.
     * @param id Param ID.
     * @return Param value or -1.
     */
    virtual float getParam(VTrackerParam id) = 0;
    /**
     * @brief Get video tracker params (results).
     * @return Video tracker params structure.
     */
    virtual VTrackerParams getParams() = 0;
    /**
     * @brief Execute command.
     * @param id Command ID.
     * @param arg1 First argument. Value depends on command ID.
     * @param arg2 Second argument. Value depends on command ID.
     * @param arg3 Third argument. Value depends on command ID.
     * @return TRUE if the command accepted or FALSE if not.
     */
    virtual bool executeCommand(VTrackerCommand id,
                                float arg1 = 0,
                                float arg2 = 0,
                                float arg3 = 0) = 0;
    /**
```

```
     * @brief Process frame. Must be used for each input video frame.
     * @param frame Source video frame.
     * @return TRUE if video frame was processed or FALSE if not.
     */
    virtual bool processFrame(cr::video::Frame& frame) = 0;
    /**
     * @brief Get image of internal surfaces.
     * @param type Type of image to get. Depends on implementation.
     * @param image Pointer to image buffer. Must be 128x128 = 16384 bytes.
     */
    virtual void getImage(int type, cr::video::Frame& image) = 0;
    /**
     * @brief Encode set param command.
     * @param data Pointer to data buffer. Must have size >= 11.
     * @param size Size of encoded data.
     * @param id Param id.
     * @param value Param value.
     */
    static void encodeSetParamCommand(
            uint8_t* data, int& size, VTrackerParam id, float value);
    /**
     * @brief Encode command.
     * @param data Pointer to data buffer. Must have size >= 11.
     * @param size Size of encoded data.
     * @param id Command ID.
     * @param arg1 First argument. Value depends on command ID.
     * @param arg2 Second argument. Value depends on command ID.
     * @param arg3 Third argument. Value depends on command ID.
     */
    static void encodeCommand(
            uint8_t* data, int& size, VTrackerCommand id,
            float arg1 = 0, float arg2 = 0, float arg3 = 0);
    /**
     * @brief Decode command.
     * @param data Pointer to command data.
     * @param size Size of data.
     * @param paramId Output command ID.
     * @param commandId Output command ID.
     * @param value1 Param or command argument 1.
     * @param value2 Command argument 2.
     * @param value3 Command argument 3.
     * @return 0 - command decoded, 1 - set param command decoded, -1 - error.
     */
    static int decodeCommand(uint8_t* data,
                             int size,
                             VTrackerParam& paramId,
                             VTrackerCommand& commandId,
                             float& value1,
                             float& value2,
                             float& value3);
};
```

getVersion method

getVersion() method returns string of current version of **VTracker** class. Particular video tracker class can have it's own **getVersion()** method. Method declaration:

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

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

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

Console output:

```
VTracker class version: 1.0.0
```

initVTracker method

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

```
virtual bool initVTracker(VTrackerParams& params) = 0;
```

Parameter	Value
params	Video tracker params class. Video tracker should initialize all parameters listed in VTrackerParams.

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

setParam method

setParam(...) method designed to set new video tracker parameter value. Method declaration:

```
virtual bool setParam(VTrackerParam id, float value) = 0;
```

Parameter	Description
id	Parameter ID according to **VTrackerParam** enum.
value	Parameter value. Value depends on parameter ID.

Returns: TRUE if the parameter was set or FALSE if not.

getParam method

getParam(...) method designed to obtain video tracker parameter value. Method declaration:

```
virtual float getParam(VTrackerParam id) = 0;
```

Parameter	Description
id	Parameter ID according to **VTrackerParam** enum.

Returns: parameter value or -1 of the parameters doesn't exist in particular object detector class.

getParams method

getParams(...) method designed to obtain video tracker params structures as well as tracking results. Method declaration:

```
virtual VTrackerParams getParams() = 0;
```

Returns: video tracker parameters class (see **VTrackerParams** class description).

executeCommand method

executeCommand(...) method designed to execute video tracker command. Method declaration:

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

Parameter	Description
id	Command ID according to **VTrackerCommand** enum.
arg1	First argument. Value depends on command ID.
arg2	Second argument. Value depends on command ID.
arg3	Third argument. Value depends on command ID.

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

processFrame method

processFrame(...) method designed to process video frame. Method declaration:

```
virtual bool processFrame(cr::video::Frame& frame) = 0;
```

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 (see **VTrackerParam** enum description) the method should return TRUE.

getImage method

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

```
virtual void getImage(int type, cr::video::Frame& image) = 0;
```

Parameter	Description
type	Image type. Depends on implementation.
frame	Output frame. Pixel format depends on implementation.

encodeSetParamCommand method

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

SET_PARAM command format:

Byte	Value	Description
0	0x01	SET_PARAM command header value.
1	0x01	Major version of VTracker class.
2	0x00	Minor version of VTracker class.
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

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).

Parameter	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:

Byte	Value	Description
0	0x00	COMMAND header value.
1	0x01	Major version of ObjectDetector class.
2	0x00	Minor version of ObjectDetector class.
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);
```

decodeCommand method

decodeCommand(...) static method designed to decode command on video tracker side (edge device). Method declaration:

```
static int decodeCommand(uint8_t* data, int size, VTrackerParam& paramId,
VTrackerCommand& commandId, float& value1, float& value2, float& value3);
```

Parameter	Description
data	Pointer to input command.
size	Size of command. Should be 11 bytes.
paramId	Parameter ID according to **VTrackerParam** enum. After decoding SET_PARAM command the method will return parameter ID.
commandId	Command ID according to **VTrackerCommand** enum. After decoding COMMAND the method will return command ID.
value1	Parameter value or first command argument 1. After decoding SET_PARAM and COMMAND.
value2	Parameter value or second command argument 1. After decoding COMMAND.
value3	Parameter value or third command argument 1. After decoding COMMAND.

Returns: 0 - in case decoding COMMAND, **1** - in case decoding SET_PARAM command or **-1** in case errors.

decodeCommand(...) is static and used without **VTracker** class instance. Command decoding example:

```
// Encode command.
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);

// Decode command.
VTrackerCommand commandId;
VTrackerParam paramId;
float inArg1 = (float)(rand() % 20);
```

```
float inArg2 = (float)(rand() % 20);
float inArg3 = (float)(rand() % 20);
if (VTracker::decodeCommand(data, size, paramId, commandId, inArg1, inArg2, inArg3) != 0)
{
    cout << "Command not decoded" << endl;
    return false;
}
```

Data structures

VTracker.h file defines IDs for parameters (**VTrackerParam** enum) and IDs for commands (**VTrackerCommand** enum).

VTrackerCommand enum

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 rectagle size. Arguments:
    /// arg1 - horizontal size add, pixels.
    /// arg2 - vertical size add, pixels.
    CHANGE_RECT_SIZE
};
```

Table 2 - Video tracker commands description. Some commands maybe unsupported by particular video tracker class.

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	
RESET	
SET_INERTIAL_MODE	
SET_LOST_MODE	
SET_STATIC_MODE	
ADJUST_RECT_SIZE	
ADJUST_RECT_POSITION	
MOVE_RECT	
SET_RECT_POSITION	
SET_RECT_POSITION_PERCENTS	
MOVE_SEARCH_WINDOW	
SET_SEARCH_WINDOW_POSITION	
SET_SEARCH_WINDOW_POSITION_PERCENTS	

Command	Description
CHANGE_RECT_SIZE	

VTrackerParam enum

Enum declaration:

```
enum class ObjectDetectorParam
{
    /// Logging mode. Values: 0 - Disable, 1 - Only file,
    /// 2 - Only terminal (console), 3 - File and terminal (console).
    LOG_MODE = 1,
    /// Frame buffer size. Depends on implementation.
    FRAME_BUFFER_SIZE,
    /// Minimum object width to be detected, pixels. To be detected object's
    /// width must be >= MIN_OBJECT_WIDTH.
    MIN_OBJECT_WIDTH,
    /// Maximum object width to be detected, pixels. To be detected object's
    /// width must be <= MAX_OBJECT_WIDTH.
    MAX_OBJECT_WIDTH,
    /// Minimum object height to be detected, pixels. To be detected object's
    /// height must be >= MIN_OBJECT_HEIGHT.
    MIN_OBJECT_HEIGHT,
    /// Maximum object height to be detected, pixels. To be detected object's
    /// height must be <= MAX_OBJECT_HEIGHT.
    MAX_OBJECT_HEIGHT,
    /// Minimum object's horizontal speed to be detected, pixels/frame. To be
    /// detected object's horizontal speed must be >= MIN_X_SPEED.
    MIN_X_SPEED,
    /// Maximum object's horizontal speed to be detected, pixels/frame. To be
    /// detected object's horizontal speed must be <= MAX_X_SPEED.
    MAX_X_SPEED,
    /// Minimum object's vertical speed to be detected, pixels/frame. To be
    /// detected object's vertical speed must be >= MIN_Y_SPEED.
    MIN_Y_SPEED,
    /// Maximum object's vertical speed to be detected, pixels/frame. To be
    /// detected object's vertical speed must be <= MAX_Y_SPEED.
    MAX_Y_SPEED,
    /// Probability threshold from 0 to 1. To be detected object detection
    /// probability must be >= MIN_DETECTION_PROPABILITY.
    MIN_DETECTION_PROPABILITY,
    /// Horizontal track detection criteria, frames. By default shows how many
    /// frames the objects must move in any(+/-) horizontal direction to be
    /// detected.
    X_DETECTION_CRITERIA,
    /// Vertical track detection criteria, frames. By default shows how many
    /// frames the objects must move in any(+/-) vertical direction to be
    /// detected.
    Y_DETECTION_CRITERIA,
    /// Track reset criteria, frames. By default shows how many
    /// frames the objects should be not detected to be excluded from results.
    RESET_CRITERIA,
```

```
    /// Detection sensitivity. Depends on implementation. Default from 0 to 1.
    SENSITIVITY,
    /// Frame scaling factor for processing purposes. Reduce the image size by
    /// scaleFactor times horizontally and vertically for faster processing.
    SCALE_FACTOR,
    /// Num threads. Number of threads for parallel computing.
    NUM_THREADS,
    /// Processing time for last frame, mks.
    PROCESSING_TIME_MKS,
    /// Algorithm type. Depends on implementation.
    TYPE,
    /// Mode. Default: 0 - Off, 1 - On.
    MODE,
    /// Custom parameter. Depends on implementation.
    CUSTOM_1,
    /// Custom parameter. Depends on implementation.
    CUSTOM_2,
    /// Custom parameter. Depends on implementation.
    CUSTOM_3
};
```

Table 3 - Object detector params description. Some params maybe unsupported by particular object detector class.

Parameter	Access	Description
LOG_MODE	read / write	Logging mode. Values: 0 - Disable, 1 - Only file, 2 - Only terminal, 3 - File and terminal.
FRAME_BUFFER_SIZE	read / write	Frame buffer size. Depends on implementation. It can be buffer size for image filtering or can be buffer size to collect frames for processing.
MIN_OBJECT_WIDTH	read / write	Minimum object width to be detected, pixels. To be detected object's width must be >= MIN_OBJECT_WIDTH.
MAX_OBJECT_WIDTH	read / write	Maximum object width to be detected, pixels. To be detected object's width must be <= MAX_OBJECT_WIDTH.
MIN_OBJECT_HEIGHT	read / write	Minimum object height to be detected, pixels. To be detected object's height must be >= MIN_OBJECT_HEIGHT.
MAX_OBJECT_HEIGHT	read / write	Maximum object height to be detected, pixels. To be detected object's height must be <= MAX_OBJECT_HEIGHT.
MIN_X_SPEED	read / write	Minimum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be >= MIN_X_SPEED.

Parameter	Access	Description
MAX_X_SPEED	read / write	Maximum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be <= MAX_X_SPEED.
MIN_Y_SPEED	read / write	Minimum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be >= MIN_Y_SPEED.
MAX_Y_SPEED	read / write	Maximum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be <= MAX_Y_SPEED.
MIN_DETECTION_PROPABILITY	read / write	Probability threshold from 0 to 1. To be detected object detection probability must be >= MIN_DETECTION_PROPABILITY. For example: neural networks for each detection result calculates detection probability from 0 to 1. MIN_DETECTION_PROPABILITY parameters used to filter detection results.
X_DETECTION_CRITERIA	read / write	Horizontal track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) horizontal direction to be detected.
Y_DETECTION_CRITERIA	read / write	Vertical track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) vertical direction to be detected.
RESET_CRITERIA	read / write	Track reset criteria, frames. By default shows how many frames the objects should be not detected to be excluded from results.
SENSITIVITY	read / write	Detection sensitivity. Depends on implementation. Default from 0 to 1.
SCALE_FACTOR	read / write	Frame scaling factor for processing purposes. Reduce the image size by scaleFactor times horizontally and vertically for faster processing.
NUM_THREADS	read / write	Num threads. Number of threads for parallel computing.
PROCESSING_TIME_MKS	read only	Processing time for last frame, mks.
TYPE	read / write	Algorithm type. Depends on implementation.
MODE	read / write	Mode. Default: 0 - Off, 1 - On.
CUSTOM_1	read / write	Custom parameter. Depends on implementation.

Parameter	Access	Description
CUSTOM_2	read / write	Custom parameter. Depends on implementation.
CUSTOM_3	read / write	Custom parameter. Depends on implementation.

Object structure

Object structure used to describe detected object. Object structure declared in **ObjectDetector.h** file. Structure declaration:

```
typedef struct Object
{
    /// Object ID. Must be uniques for particular object.
    int id{0};
    /// Object type. Depends on implementation.
    int type{0};
    /// Object rectangle width, pixels.
    int width{0};
    /// Object rectangle height, pixels.
    int height{0};
    /// Object rectangle top-left horizontal coordinate, pixels.
    int x{0};
    /// Object rectangle top-left vertical coordinate, pixels.
    int y{0};
    /// Horizontal component of object velocity, +-pixels/frame.
    float vX{0.0f};
    /// Vertical component of object velocity, +-pixels/frame.
    float vY{0.0f};
    /// Detection probability from 0 (minimum) to 1 (maximum).
    float p{0.0f};
} Object;
```

Table 4 - Object structure fields description.

Field	Type	Description
id	int	Object ID. Must be uniques for particular object. Object detector must assign unique ID for each detected object. This is necessary for control algorithms to distinguish different objects from frame to frame.
type	int	Object type. Depends on implementation. For example detection neural networks can detect multiple type of objects.
width	int	Object rectangle width, pixels. Must be MIN_OBJECT_WIDTH <= width <= MAX_OBJECT_WIDTH (see **ObjectDetectorParam** enum description).
height	int	Object rectangle height, pixels. Must be MIN_OBJECT_HEIGHT <= height <= MAX_OBJECT_HEIGHT (see **ObjectDetectorParam** enum description).

Field	Type	Description
x	int	Object rectangle top-left horizontal coordinate, pixels.
y	int	Object rectangle top-left vertical coordinate, pixels.
vX	float	Horizontal component of object velocity, +-pixels/frame. if it possible object detector should estimate object velocity on video frames.
vY	float	Vertical component of object velocity, +-pixels/frame. if it possible object detector should estimate object velocity on video frames.
p	float	Detection probability from 0 (minimum) to 1 (maximum). Must be p >= MIN_DETECTION_PROPABILITY (see **ObjectDetectorParam** enum description).

ObjectDetectorParams class description

ObjectDetectorParams class declaration

ObjectDetectorParams class used for object detector initialization (**initObjectDetector(...)** method) or to get all actual params (**getParams()** method). Also **ObjectDetectorParams** provide structure to write/read params from JSON files (**JSON_READABLE** macro, see **ConfigReader** class description) and provide methos to encode and decode params. Class declaration:

```
class ObjectDetectorParams
{
public:
    /// Init string. Depends on implementation.
    std::string initString{""};
    /// Logging mode. Values: 0 - Disable, 1 - Only file,
    /// 2 - Only terminal (console), 3 - File and terminal (console).
    int logMode{0};
    /// Frame buffer size. Depends on implementation.
    int frameBufferSize{1};
    /// Minimum object width to be detected, pixels. To be detected object's
    /// width must be >= minObjectWidth.
    int minObjectWidth{4};
    /// Maximum object width to be detected, pixels. To be detected object's
    /// width must be <= maxObjectWidth.
    int maxObjectWidth{128};
    /// Minimum object height to be detected, pixels. To be detected object's
    /// height must be >= minObjectHeight.
    int minObjectHeight{4};
    /// Maximum object height to be detected, pixels. To be detected object's
    /// height must be <= maxObjectHeight.
    int maxObjectHeight{128};
    /// Minimum object's horizontal speed to be detected, pixels/frame. To be
    /// detected object's horizontal speed must be >= minXSpeed.
    float minXSpeed{0.0f};
    /// Maximum object's horizontal speed to be detected, pixels/frame. To be
```

```
    /// detected object's horizontal speed must be <= maxXSpeed.
    float maxXSpeed{30.0f};
    /// Minimum object's vertical speed to be detected, pixels/frame. To be
    /// detected object's vertical speed must be >= minYSpeed.
    float minYSpeed{0.0f};
    /// Maximum object's vertical speed to be detected, pixels/frame. To be
    /// detected object's vertical speed must be <= maxYSpeed.
    float maxYSpeed{30.0f};
    /// Probability threshold from 0 to 1. To be detected object detection
    /// probability must be >= minDetectionProbability.
    float minDetectionProbability{0.5f};
    /// Horizontal track detection criteria, frames. By default shows how many
    /// frames the objects must move in any(+/-) horizontal direction to be
    /// detected.
    int xDetectionCriteria{1};
    /// Vertical track detection criteria, frames. By default shows how many
    /// frames the objects must move in any(+/-) vertical direction to be
    /// detected.
    int yDetectionCriteria{1};
    /// Track reset criteria, frames. By default shows how many
    /// frames the objects should be not detected to be excluded from results.
    int resetCriteria{1};
    /// Detection sensitivity. Depends on implementation. Default from 0 to 1.
    float sensitivity{0.04f};
    /// Frame scaling factor for processing purposes. Reduce the image size by
    /// scaleFactor times horizontally and vertically for faster processing.
    int scaleFactor{1};
    /// Num threads. Number of threads for parallel computing.
    int numThreads{1};
    /// Processing time for last frame, mks.
    int processingTimeMks{0};
    /// Algorithm type. Depends on implementation.
    int type{0};
    /// Mode. Default: false - Off, on - On.
    bool enable{true};
    /// 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};
    /// List of detected objects.
    std::vector<Object> objects;

    JSON_READABLE(ObjectDetectorParams, initString, logMode, frameBufferSize,
                  minObjectWidth, maxObjectWidth, minObjectHeight, maxObjectHeight,
                  minXSpeed, maxXSpeed, minYSpeed, maxYSpeed, minDetectionProbability,
                  xDetectionCriteria, yDetectionCriteria, resetCriteria, sensitivity,
                  scaleFactor, numThreads, type, enable, custom1, custom2, custom3);
    /**
     * @brief operator =
     * @param src Source object.
     * @return ObjectDetectorParams object.
     */
    ObjectDetectorParams& operator= (const ObjectDetectorParams& src);
```

```
    /**
     * @brief Encode params. Method doesn't encode initString.
     * @param data Pointer to data buffer.
     * @param size Size of data.
     * @param mask Pointer to parameters mask.
     */
    void encode(uint8_t* data, int& size,
                ObjectDetectorParamsMask* mask = nullptr);
    /**
     * @brief Decode params. Method doesn't decode initString;
     * @param data Pointer to data.
     * @return TRUE is params decoded or FALSE if not.
     */
    bool decode(uint8_t* data);
};
```

Table 5 - ObjectDetectorParams class fields description.

Field	Type	Description
initString	string	Init string. Depends on implementation.
logMode	int	Logging mode. Values: 0 - Disable, 1 - Only file, 2 - Only terminal, 3 - File and terminal.
frameBufferSize	int	Frame buffer size. Depends on implementation. It can be buffer size for image filtering or can be buffer size to collect frames for processing.
minObjectWidth	int	Minimum object width to be detected, pixels. To be detected object's width must be >= minObjectWidth.
maxObjectWidth	int	Maximum object width to be detected, pixels. To be detected object's width must be <= maxObjectWidth.
minObjectHeight	int	Minimum object height to be detected, pixels. To be detected object's height must be >= minObjectHeight.
maxObjectHeight	int	Maximum object height to be detected, pixels. To be detected object's height must be <= maxObjectHeight.
minXSpeed	float	Minimum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be >= minXSpeed.
maxXSpeed	float	Maximum object's horizontal speed to be detected, pixels/frame. To be detected object's horizontal speed must be <= maxXSpeed.
minYSpeed	float	Minimum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be >= minYSpeed.

Field	Type	Description
maxYSpeed	float	Maximum object's vertical speed to be detected, pixels/frame. To be detected object's vertical speed must be <= maxYSpeed.
minDetectionProbability	float	Probability threshold from 0 to 1. To be detected object detection probability must be >= minDetectionProbability. For example: neural networks for each detection result calculates detection probability from 0 to 1. minDetectionProbability parameters used to filter detection results.
xDetectionCriteria	int	Horizontal track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) horizontal direction to be detected.
yDetectionCriteria	int	Vertical track detection criteria, frames. By default shows how many frames the objects must move in any(+/-) vertical direction to be detected.
resetCriteria	int	Track reset criteria, frames. By default shows how many frames the objects should be not detected to be excluded from results.
sensitivity	float	Detection sensitivity. Depends on implementation. Default from 0 to 1.
scaleFactor	int	Frame scaling factor for processing purposes. Reduce the image size by scaleFactor times horizontally and vertically for faster processing.
numThreads	int	Num threads. Number of threads for parallel computing.
processingTimeMks	int	Processing time for last frame, mks.
type	int	Algorithm type. Depends on implementation.
enable	bool	Mode: false - Off, true - On.
custom1	float	Custom parameter. Depends on implementation.
custom2	float	Custom parameter. Depends on implementation.
custom3	float	Custom parameter. Depends on implementation.
objects	std::vector	List of detected objects.

None: *ObjectDetectorParams class fields listed in Table 5* ***must*** *reflect params set/get by methods setParam(...) and getParam(...).*

Serialize object detector params

ObjectDetectorParams class provides method **encode(...)** to serialize object detector params (fields of ObjectDetectorParams class, see Table 5). Serialization of object detector 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 (3 bytes) where each bit represents particular parameter and **decode(...)** method recognizes it. Method doesn't encode initString. Method declaration:

```
void encode(uint8_t* data, int& size, ObjectDetectorParamsMask* mask = nullptr);
```

Parameter	Value
data	Pointer to data buffer. Buffer size should be at least **43** bytes.
size	Size of encoded data. 43 bytes by default.
mask	Parameters mask - pointer to **ObjectDetectorParamsMask** structure. **ObjectDetectorParamsMask** (declared in ObjectDetector.h file) determines flags for each field (parameter) declared in **ObjectDetectorParams** 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 ObjectDetectorParamsMask structure.

ObjectDetectorParamsMask structure declaration:

```
typedef struct ObjectDetectorParamsMask
{
    bool logMode{true};
    bool frameBufferSize{true};
    bool minObjectWidth{true};
    bool maxObjectWidth{true};
    bool minObjectHeight{true};
    bool maxObjectHeight{true};
    bool minXSpeed{true};
    bool maxXSpeed{true};
    bool minYSpeed{true};
    bool maxYSpeed{true};
    bool minDetectionProbability{true};
    bool xDetectionCriteria{true};
    bool yDetectionCriteria{true};
    bool resetCriteria{true};
    bool sensitivity{true};
    bool scaleFactor{true};
    bool numThreads{true};
    bool processingTimeMks{true};
    bool type{true};
    bool enable{true};
    bool custom1{true};
    bool custom2{true};
    bool custom3{true};
    bool objects{true};
} ObjectDetectorParamsMask;
```

Example without parameters mask:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.objects.clear();
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vX = rand() % 255;
    obj.vY = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

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

Example with parameters mask:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.objects.clear();
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vX = rand() % 255;
    obj.vY = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

// Prepare mask.
ObjectDetectorParamsMask mask;
mask.logMode = false;

// Encode data.
uint8_t data[1024];
int size = 0;
```

```
in.encode(data, size, &mask)
cout << "Encoded data size: " << size << " bytes" << endl;
```

Deserialize object detector params

ObjectDetectorParams class provides method **decode(...)** to deserialize params (fields of ObjectDetectorParams 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 doesn't decode initString. Method declaration:

```
bool decode(uint8_t* data);
```

Parameter	Value
data	Pointer to encode data buffer.

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

Example:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vX = rand() % 255;
    obj.vY = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

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

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

Read params from JSON file and write to JSON file

ObjectDetector 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.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.objects.clear();
for (int i = 0; i < 5; ++i)
{
    Object obj;
    obj.id = rand() % 255;
    obj.type = rand() % 255;
    obj.width = rand() % 255;
    obj.height = rand() % 255;
    obj.x = rand() % 255;
    obj.y = rand() % 255;
    obj.vX = rand() % 255;
    obj.vY = rand() % 255;
    obj.p = rand() % 255;
    in.objects.push_back(obj);
}

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

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

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

ObjectDetectorParams.json will look like:

```
{
    "ObjectDetectorParams": {
        "custom1": 57.0,
        "custom2": 244.0,
        "custom3": 68.0,
        "enable": false,
```

```
        "frameBufferSize": 200,
        "initString": "sfsfsfsfsf",
        "logMode": 111,
        "maxObjectHeight": 103,
        "maxObjectWidth": 199,
        "maxXSpeed": 104.0,
        "maxYSpeed": 234.0,
        "minDetectionProbability": 53.0,
        "minObjectHeight": 191,
        "minObjectWidth": 149,
        "minXSpeed": 213.0,
        "minYSpeed": 43.0,
        "numThreads": 33,
        "resetCriteria": 62,
        "scaleFactor": 85,
        "sensitivity": 135.0,
        "type": 178,
        "xDetectionCriteria": 224,
        "yDetectionCriteria": 199
    }
}
```


Build and connect to your project

Typical commands to build **ObjectDetector** library:

```
git clone https://github.com/ConstantRobotics-Ltd/ObjectDetector.git
cd ObjectDetector
git submodule update --init --recursive
mkdir build
cd build
cmake ..
make
```

If you want connect **ObjectDetector** 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
```

You can add repository **ObjectDetector** as submodule by commands:

```
cd <your respository folder>
git submodule add https://github.com/ConstantRobotics-Ltd/ObjectDetector.git
3rdparty/ObjectDetector
git submodule update --init --recursive
```

In you repository folder will be created folder **3rdparty/ObjectDetector** which contains files of **ObjectDetector** repository with subrepositories **Frame** and **ConfigReader**. New structure of your repository:

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

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(${PARENT}_SUBMODULE_OBJECT_DETECTOR                 ON  CACHE BOOL "" FORCE)
if (${PARENT}_SUBMODULE_OBJECT_DETECTOR)
    SET(${PARENT}_OBJECT_DETECTOR                       ON  CACHE BOOL "" FORCE)
    SET(${PARENT}_OBJECT_DETECTOR_TEST                  OFF CACHE BOOL "" FORCE)
endif()

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

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

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

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 ObjectDetector library in your **src/CMakeLists.txt** file:

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

Done!